CS229 2018 Lecture Notes

Machine Learning — Stanford, Autumn 2018, with Andrew Ng

Murat Raimbekov

Table of contents

CS229 2018 Lecture Notes **1**
Lectures . . . 1
What these notes are . . . 2
Roadmap . . . 2
Contributing . . . 3
Disclaimer . . . 3

I. Supervised Learning **5**

1. Introduction **7**
1.1. What the course assumes . . . 7
1.2. Two definitions . . . 7
1.3. Supervised learning: regression . . . 8
1.4. Supervised learning: classification . . . 9
1.5. More features than you can draw . . . 9
1.6. Learning to steer: ALVINN . . . 11
1.7. Machine learning strategy . . . 12
1.8. Unsupervised learning . . . 12
1.9. Reinforcement learning . . . 13
1.10. The map of the course . . . 14

2. Linear Regression and Gradient Descent **15**
2.1. The housing problem and the hypothesis . . . 15
2.2. The least-squares cost . . . 16
2.3. Gradient descent . . . 16
2.4. The LMS update . . . 17
2.5. A bowl, and the learning rate . . . 18
2.6. Stochastic gradient descent . . . 20
2.7. Matrix derivatives and the trace . . . 21
2.8. The normal equations . . . 22

3. Locally Weighted and Logistic Regression **25**
3.1. Features, or a different algorithm . . . 25
3.2. Locally weighted regression . . . 25
3.3. Least squares as maximum likelihood . . . 27
3.4. Why not linear regression for classification . . . 28
3.5. The logistic model . . . 29
3.6. Maximum likelihood and gradient ascent . . . 30
3.7. Newton's method . . . 31

4. Perceptron and Generalized Linear Models **33**
4.1. The perceptron . . . 33
4.2. The exponential family . . . 35

4.3. Why the exponential family is convenient . . . 36
4.4. Generalized linear models . . . 37
4.5. Three parametrisations . . . 38
4.6. What the assumptions say about the data . . . 39
4.7. Softmax regression . . . 39

5. GDA and Naive Bayes **43**
5.1. Two ways to build a classifier . . . 43
5.2. The multivariate Gaussian . . . 44
5.3. The GDA model and its fit . . . 45
5.4. GDA's boundary is logistic . . . 46
5.5. Which one to use . . . 48
5.6. Naive Bayes . . . 49

6. Support Vector Machines **53**
6.1. Laplace smoothing . . . 53
6.2. Discretising a continuous feature . . . 54
6.3. The multinomial event model . . . 54
6.4. Build something quick and dirty first . . . 56
6.5. Why support vector machines . . . 57
6.6. The functional margin . . . 58
6.7. The geometric margin . . . 58
6.8. The optimal margin classifier . . . 59

7. Kernels **61**
7.1. From the largest margin to the smallest w . . . 61
7.2. The representer theorem . . . 62
7.3. Everything is an inner product . . . 62
7.4. The kernel trick . . . 63
7.5. Polynomial kernels . . . 64
7.6. Seeing the lift . . . 64
7.7. Which functions are kernels . . . 65
7.8. Kernelising other algorithms . . . 67
7.9. The soft margin . . . 67
7.10. Kernels for inputs that are not vectors . . . 68

II. Generalization **71**

8. Data Splits, Models and Cross-Validation **73**
8.1. Bias and variance . . . 73
8.2. Regularisation . . . 75
8.3. Regularisation as a prior on . . . 76
8.4. Choosing a model: train, dev and test sets . . . 78
8.5. k-fold and leave-one-out cross-validation . . . 80
8.6. Feature selection . . . 81

Learning Theory **83**
The learning algorithm as an estimator . . . 83
Bias and variance as moments of the sampling distribution . . . 84
Approximation error and estimation error . . . 86
Empirical risk minimisation and two tools . . . 87

Uniform convergence for a finite class . . . 88
How good is the hypothesis ERM picks? . . . 90
Infinite classes and the VC dimension . . . 90

III. Trees and Neural Networks **93**

9. Decision Trees and Ensemble Methods **95**
9.1. A model that cuts the space into boxes . . . 95
9.2. Choosing a split . . . 96
9.3. Regression trees and categorical features . . . 97
9.4. Regularising a tree . . . 98
9.5. Runtime . . . 98
9.6. What trees cannot do . . . 99
9.7. Why averaging helps . . . 99
9.8. Bagging . . . 100
9.9. Random forests . . . 101
9.10. Boosting . . . 102

10. Introduction to Neural Networks **105**
10.1. Logistic regression as one neuron . . . 105
10.2. Three animals, three neurons . . . 106
10.3. Softmax: one class per image . . . 107
10.4. Predicting a number . . . 108
10.5. Hidden layers . . . 109
10.6. Forward propagation and shapes . . . 110
10.7. Vectorising over a batch . . . 110
10.8. The cost, and why backpropagation starts at the end . . . 111

11. Backprop and Improving Neural Networks **113**
11.1. The gradient at the output . . . 113
11.2. The chain rule, one layer down . . . 114
11.3. What the forward pass must keep . . . 115
11.4. Activation functions . . . 115
11.5. Why the network needs a non-linearity . . . 116
11.6. Normalising the inputs . . . 117
11.7. Vanishing and exploding gradients . . . 118
11.8. Initialising the weights . . . 118
11.9. Mini-batch gradient descent . . . 119
11.10Momentum . . . 120

IV. Applying Machine Learning **123**

12. Debugging ML Models and Error Analysis **125**
12.1. What to try next . . . 125
12.2. Learning curves . . . 126
12.3. Matching the fix to the diagnosis . . . 127
12.4. Optimisation algorithm or objective . . . 127
12.5. The helicopter . . . 129
12.6. Error analysis . . . 131
12.7. Ablative analysis . . . 132

V. Unsupervised Learning 135

13. Expectation-Maximization Algorithms 137
13.1. k-means . . . 137
13.2. The cost k-means descends . . . 138
13.3. Density estimation and anomalies . . . 139
13.4. The mixture of Gaussians . . . 139
13.5. EM for the mixture: soft k-means . . . 139
13.6. Jensen's inequality . . . 140
13.7. EM as maximising a lower bound . . . 142

14. EM Algorithm and Factor Analysis 145
14.1. From the general algorithm to the mixture updates . . . 145
14.2. Fewer examples than dimensions . . . 146
14.3. Restricting the covariance . . . 147
14.4. The factor analysis model . . . 148
14.5. Marginals and conditionals of Gaussians . . . 149
14.6. EM for factor analysis . . . 150

15. PCA and ICA 153
15.1. One-dimensional data living in two dimensions . . . 153
15.2. Two definitions of the best direction . . . 154
15.3. The principal direction is an eigenvector . . . 155
15.4. Pre-processing and the choice of k . . . 155
15.5. When to use PCA, and when not to . . . 156
15.6. Why individual eigenvectors should not be interpreted . . . 157
15.7. A map of the unsupervised algorithms . . . 158
15.8. Independent components analysis: the cocktail party . . . 158

VI. Reinforcement Learning 161

16. Independent Component Analysis and RL 163
16.1. Gaussian sources cannot be unmixed . . . 163
16.2. The density of a linear transformation . . . 164
16.3. A prior for the sources: the sigmoid as a CDF . . . 164
16.4. Maximum likelihood for the unmixing matrix . . . 165
16.5. Where ICA is used, and where it breaks . . . 167
16.6. Why reinforcement learning . . . 167
16.7. Markov decision processes . . . 168
16.8. Payoff, discounting and policies . . . 168

17. MDPs and Value/Policy Iteration 171
17.1. The value of a policy . . . 171
17.2. The optimal value function and the optimal policy . . . 172
17.3. Value iteration . . . 173
17.4. Reading the policy off V^* . . . 174
17.5. Policy iteration . . . 174
17.6. Learning the model . . . 175
17.7. Exploration versus exploitation . . . 176

18. Continuous State MDPs and Model Simulation 179
18.1. States that are vectors . . . 179

18.2. Discretization, and what it costs . . . 179
18.3. Getting a model . . . 180
18.4. Fitted value iteration . . . 182
18.5. From a value function to an action . . . 184

19. Finite-Horizon MDPs and the Linear Quadratic Regulator **185**
19.1. Rewards that depend on the action . . . 185
19.2. Finite horizons and policies that watch the clock . . . 186
19.3. Solving it backwards . . . 186
19.4. The LQR model . . . 187
19.5. Linearizing a nonlinear model . . . 188
19.6. The value function is exactly quadratic . . . 188
19.7. The noise does not change the policy . . . 190

20. RL Debugging and Direct Policy Search **193**
20.1. Debugging the helicopter, in RL notation . . . 193
20.2. Searching for the policy directly . . . 194
20.3. Stochastic policies . . . 194
20.4. The objective . . . 195
20.5. REINFORCE . . . 196
20.6. Why the update follows the gradient . . . 196
20.7. Why REINFORCE is slow, and what helps . . . 197
20.8. Partial observability . . . 199
20.9. Policy or value function? . . . 199

CS229 2018 Lecture Notes

Machine Learning — Stanford University, Autumn 2018

> **How to read these notes**
>
> Each lecture is a standalone page written to be read straight through. The recording stays one click away: the chapter links its video under the title, and every section carries a timestamp that opens the lecture at the moment the idea is introduced, so you can watch the derivation being built when the prose is not enough. Sections, equations and figures are numbered and individually linkable.

Lectures

The course runs to twenty lectures and one discussion section, and every one has a chapter; the roadmap groups them by theme.

Table 1.: Lectures and their status.

#	Topic	Status
1	Introduction	Complete
2	Linear Regression and Gradient Descent	Complete
3	Locally Weighted and Logistic Regression	Complete
4	Perceptron and Generalized Linear Models	Complete
5	GDA and Naive Bayes	Complete
6	Support Vector Machines	Complete
7	Kernels	Complete
8	Data Splits, Models and Cross-Validation	Complete
—	Learning Theory	Complete
9	Decision Trees and Ensemble Methods	Complete
10	Introduction to Neural Networks	Complete
11	Backprop and Improving Neural Networks	Complete
12	Debugging ML Models and Error Analysis	Complete

#	Topic	Status
13	Expectation-Maximization Algorithms	Complete
14	EM Algorithm and Factor Analysis	Complete
15	PCA and ICA	Complete
16	Independent Component Analysis and RL	Complete
17	MDPs and Value/Policy Iteration	Complete
18	Continuous State MDPs and Model Simulation	Complete
19	Finite-Horizon MDPs and the Linear Quadratic Regulator	Complete
20	RL Debugging and Direct Policy Search	Complete

What these notes are

The official CS229 notes are the formal account of the course, and a good one. What they cannot carry is the lecture itself: the order in which Andrew Ng motivates an idea before he formalises it, the picture he draws to make an update rule obvious, the question from the room that exposes the step everyone was about to skip.

These notes are written from the Autumn 2018 recordings rather than from the official notes, and they are meant to sit beside both. Each page aims to be:

- **Faithful to the lecture** — it follows what was taught and why, and links each section to the moment in the video where it happens.
- **Complete in its mathematics** — the derivations done at the board are written out in full, with every symbol defined.
- **A bridge, not a copy** — where the official notes go further, the chapter says which section to read next rather than reproducing it.

The whole book is also a single PDF. For an AI assistant, llms.txt maps every chapter to a Markdown copy with the maths in TeX, and llms-full.txt holds the book in one file. The same treatment of Stanford's deep learning for computer vision course is in the companion CS231n 2025 Lecture Notes.

Roadmap

1. **Supervised learning** — linear and logistic regression, generalized linear models, generative learning, support vector machines and kernels
2. **Generalization** — bias and variance, model selection, learning theory
3. **Trees and neural networks** — decision trees and ensembles, neural networks and backpropagation
4. **Applying machine learning** — debugging learning algorithms and error analysis
5. **Unsupervised learning** — k-means, EM, factor analysis, PCA and ICA
6. **Reinforcement learning** — MDPs, value and policy iteration, continuous states, LQR

Contributing

Found an error? Corrections are genuinely welcome — there is an **Edit this page** link at the bottom of every lecture, which opens a pull request in your browser. For anything larger, open an issue.

Disclaimer

This is an **unofficial** resource, not affiliated with Stanford University. For official course materials see cs229.stanford.edu and the 2018 lecture videos.

The notes are released under the MIT licence, which covers the writing and the diagrams made for this project. Any figure reproduced from elsewhere remains the property of its original authors and is credited in its caption — see NOTICE.

Part I.

Supervised Learning

1. Introduction

Lecture 1

Lecture video: Lecture 1: Introduction · 1:15:20

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

The first lecture of CS229 does two things. It places the course among Stanford's other machine-learning courses, and it draws, in about forty minutes at the board, the map the remaining nineteen lectures fill in: learning from labelled examples, learning to apply learning algorithms well, deep learning, learning from data with no labels, and learning from reward. No algorithm is derived. What the lecture supplies instead is the set of pictures — a scatter of house prices, a row of noughts and crosses, two clouds of unlabelled points — that every later derivation is drawn on top of, and it is worth having them clearly in mind before the mathematics arrives.

1.1. What the course assumes

10:22 The stated prerequisites are modest on paper: familiarity with basic probability and statistics, basic linear algebra, and enough programming to write non-trivial code. In 2018 the assignments moved from MATLAB and Octave to Python and NumPy, which Ng presented as following the field rather than leading it.

The more useful guidance came from a question at 28:37 about how CS229 differs from CS229A and CS230. Ng's answer was not a list of topics but a statement about depth. CS229A, taught as a flipped classroom built on his online course, covers a similarly broad set of algorithms with much less mathematics; CS230 is narrower, deep learning only, and also more applied. CS229 is the one that derives things, and he was candid that the derivations make it a heavy course. His advice to anyone unsure whether they were ready was to take CS229A first. He also pointed out that the three overlap less than their titles suggest — students who took two at once saw the same algorithms from different angles rather than the same lectures twice — and closed with the remark that frames the rest of the course: AI is larger than machine learning, and machine learning is larger than deep learning.

1.2. Two definitions

36:17 Ng offers two definitions of machine learning, an informal one and an operational one.

The informal one is usually credited to Arthur Samuel: machine learning is the field of study that gives computers the ability to learn without being explicitly programmed. That exact sentence does not appear in Samuel's paper, Some Studies in Machine Learning Using the Game of Checkers (1959), and is better read as a later paraphrase of it. Samuel's own wording describes programming a computer "to behave in a way which, if done by human beings or animals, would

be described as involving the process of learning". The result the lecture attaches to it is in the paper, and it is what made the work famous. Samuel's program, given the rules of checkers and a list of board features with unknown weights, learned to play a better game than Samuel himself, partly by playing against a copy of itself and keeping whichever set of weights won. The abstract puts the training time at eight to ten hours of machine-playing time.

Ng's point about this is historical. At the time the open question was whether a computer could do anything it had not been explicitly told to do, and a program that beat its own author was a direct answer. Today it is routine for a learning algorithm to outperform people on a *narrow* task — a particular kind of speech recognition, or Go given enormous amounts of computation and self-play — and he stresses the word narrow. Samuel's checkers player was among the first cases of that pattern.

The operational definition is Tom Mitchell's, from his 1997 textbook *Machine Learning*, which calls something a *well-posed learning problem* when three things are named. A program learns from experience E with respect to a task T and a performance measure P if its performance on T, as measured by P, improves with E. 38:43 Applied to checkers, T is playing the game, E is the large number of games the program plays against itself, and P is the probability of winning its next game against a new opponent. The value of the definition is that it forces all three to be stated. A good deal of what goes wrong in applied machine learning, which the course returns to in its lectures on strategy, is a project where one of them was never pinned down: the team is measuring a P that does not track the T they care about, or gathering an E that does not resemble the situations the system will face.

1.3. Supervised learning: regression

40:12 The most widely used kind of machine learning, and the subject of the first third of the course, is supervised learning. Ng introduces it with the example the next lecture formalises. Plot the size of each house in a dataset on the horizontal axis and its sale price on the vertical, call the first x and the second y, and ask for the relationship between them. If you own a house of 1,250 square feet and want to know what to ask for it, one answer is to fit a straight line to the data and read the price off the line above 1,250.

That is the whole shape of supervised learning. You are given a training set of inputs x paired with the correct outputs y, and the job is to learn a mapping from x to y good enough to apply to an x you have not seen. The problem is called *regression* when y is continuous. Prices are rounded to the cent and so are not literally real numbers, Ng notes, but nobody prices a house at π million dollars and for every practical purpose they are continuous.

A straight line is close to the simplest learning algorithm there is, and Figure 1.1 shows why it is not the only candidate. A quadratic follows the flattening of prices at the top end and has lower error on the training points (an RMSE of 25.4 against 29.6 on this data). That comparison cannot settle the question, though. A quadratic can never fit its training data worse than a line, because every line is a quadratic whose x^2 coefficient is zero, and this particular quadratic peaks at about 2,400 square feet and predicts that every house larger than that is worth less. How to choose between models — the line, the quadratic, or something richer, whether by hand or automatically — is one of the questions Ng flags as recurring all quarter. It becomes the bias–variance discussion of lecture 8.

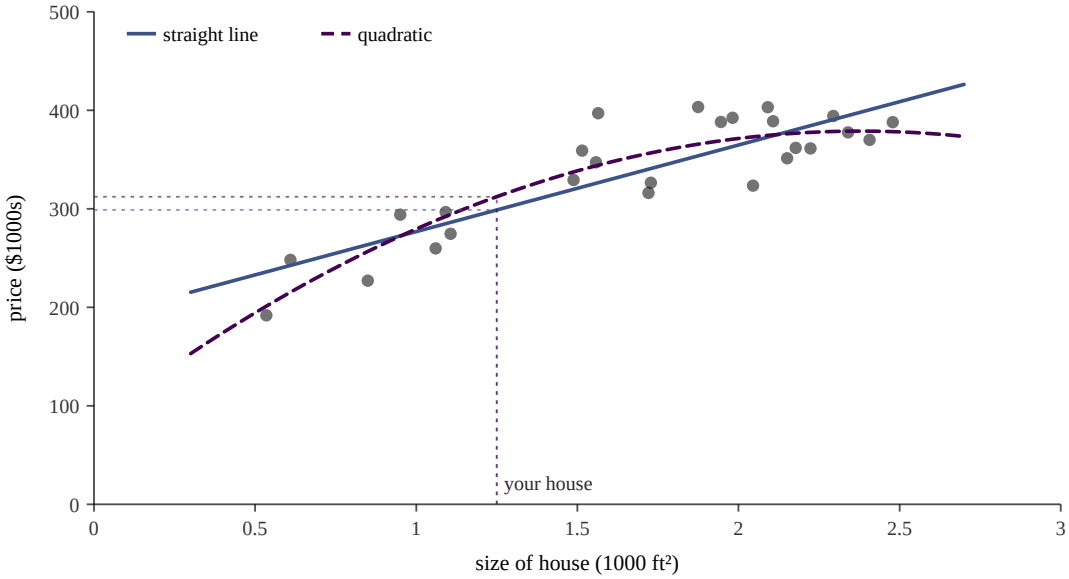


Figure 1.1.: House size against price, with a straight line and a quadratic each fitted to the same points by least squares, and the price each predicts for a 1,250 ft² house. The data are synthetic, drawn to resemble the board sketch. *Computed by* `figures/01-introduction/make_diagrams.py`.

1.4. Supervised learning: classification

43:04 The second example comes from a medical problem some of Ng's collaborators worked on: deciding from a breast tumour's size whether it is malignant or benign. Plot size on the horizontal axis and, on the vertical, $y = 1$ for malignant and $y = 0$ for benign. When a new patient arrives with a tumour of a given size, the learning algorithm should say how likely it is to be malignant. Because y now takes one of a small number of discrete values, this is *classification*. With two values it is binary classification, and with, say, five types of cancer it is still classification, just with five classes.

45:37 Ng then redraws the same data in a form the course uses from then on. Every example is pushed up onto a single horizontal line, and the label is carried by the symbol rather than by the height: a nought for a negative example ($y = 0$), a cross for a positive one ($y = 1$). Nothing is lost — Figure 1.2 shows both views of identical data — but the second frees the vertical axis to hold a second input.

47:07 That is the step he takes next. Real problems rarely give you one number to predict another; with the patient's age as a second feature, x becomes a two-dimensional vector of tumour size and age, and each example becomes a nought or a cross in the plane. The task is now to separate the two kinds of mark, and the algorithm the course reaches for first draws a straight line between them. That algorithm is logistic regression, taken up in lecture 3, and it is what fitted the boundary in Figure 1.3.

1.5. More features than you can draw

48:41 The two-feature picture is a teaching device. The group Ng describes used many more measurements of each tumour: clump thickness, uniformity of cell size, uniformity of cell shape,

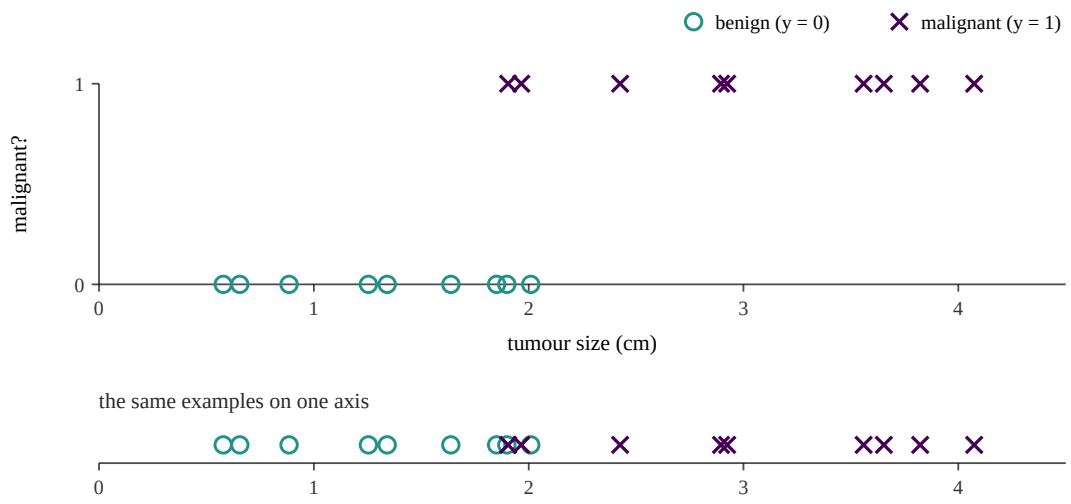


Figure 1.2.: The tumour problem twice. Above, size against the label plotted as a height; below, the same examples on one axis with the label carried by the symbol. Synthetic data. *Computed by* `figures/01-introduction/make_diagrams.py`.

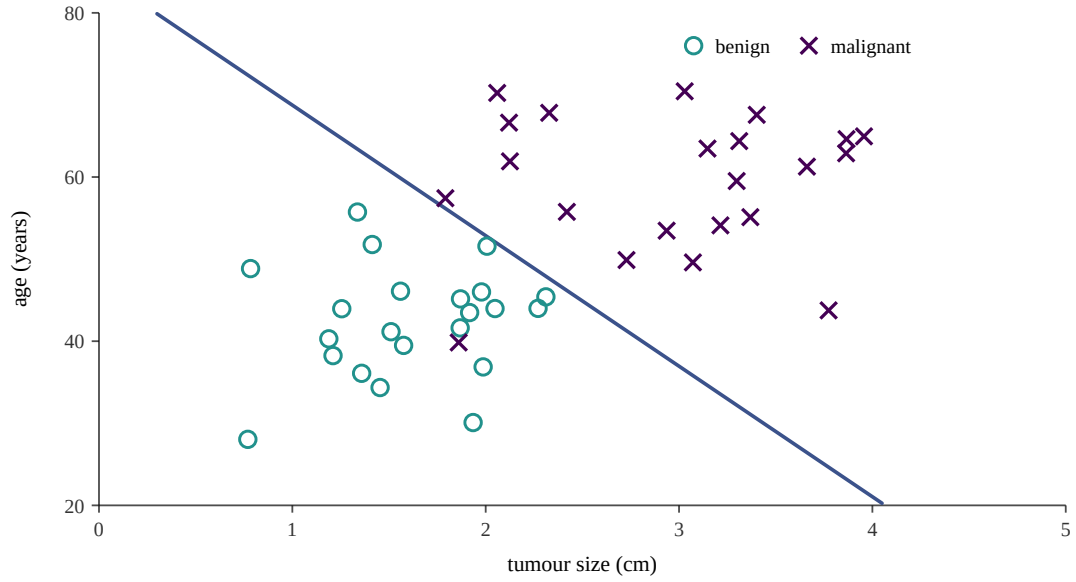


Figure 1.3.: Two features, tumour size and patient age. The line is the decision boundary logistic regression fits to these points by Newton's method; one malignant example falls on the benign side of it. Synthetic data. *Computed by* `figures/01-introduction/make_diagrams.py`.

marginal adhesion. Those are attributes of the Wisconsin breast cancer data (Breast Cancer Wisconsin (Original)), which scores each of nine cytological features on a scale of one to ten. Nine dimensions cannot be plotted. The algorithms of the coming weeks are therefore written for an input vector $x \in \mathbb{R}^n$ of any length, and the board pictures are the $n = 2$ case drawn to build intuition.

50:11 Ng then makes the remark of the lecture that most deserves a second look. The support vector machine, he says, can represent each patient not by two numbers or a million, but by an *infinite* list of features. His question is the one a reader should ask: a computer cannot store infinitely many numbers, so how can that work?

The answer, which lecture 7 develops, is that the algorithm never needs the features themselves. It only ever needs inner products between pairs of feature vectors, and for well-chosen feature maps that inner product has a closed form. A one-dimensional example shows it. Map a scalar x to the infinite vector

$$\phi(x) = e^{-x^2/2}\left(1,\; x,\; \frac{x^2}{\sqrt{2!}},\; \frac{x^3}{\sqrt{3!}},\; \ldots\right). \tag{1.1}$$

The inner product of two such vectors is a power series that sums exactly:

$$\phi(x)^{\top}\phi(z) = e^{-x^2/2}\,e^{-z^2/2}\sum_{k=0}^{\infty}\frac{(xz)^k}{k!} = e^{-x^2/2-z^2/2+xz} = e^{-(x-z)^2/2}. \tag{1.2}$$

The left-hand side is an inner product in infinitely many dimensions; the right-hand side is one subtraction and one exponential. The generator for this chapter checks the identity numerically. For $x = 0.7$ and $z = -0.4$, the first five coordinates of ϕ already match the exact value $e^{-(1.1)^2/2} \approx 0.5461$ to within 10^{-5}, and ten match it to about 10^{-12}. A function like the right-hand side of Equation 1.2 is called a *kernel*, and this one is the Gaussian kernel. The SVM is written so that features appear only inside such inner products, and that is the whole of the trick Ng is foreshadowing.

1.6. Learning to steer: ALVINN

52:05 The idea underneath all of this, Ng says, is that during training the algorithm sees inputs x together with their labels y, and its job is to find a mapping that produces the right y for a new x. To show how far that idea goes, he plays an old video of Dean Pomerleau's ALVINN at Carnegie Mellon, a neural network that learned to steer a vehicle.

The set-up is supervised learning with a human as the labeller. While a person drives, the vehicle repeatedly digitises the camera image of the road ahead, downsampled to a small greyscale picture, and records the steering direction the driver chose at that moment. The image is x and the driver's steering is y. On screen the label is a bright blob on a bar, a little left of centre when the driver is turning slightly left, and the network's output is drawn on a second bar beneath it. At the start of training the network's output is a grey smear across the whole bar, its way of saying it has no idea which way to steer. As training proceeds the smear contracts to a sharp blob that tracks the driver's. Then the driver takes their hands off the wheel and the network steers. A later version in the video trains separate networks for one-lane and two-lane roads, with an arbitrator choosing between them, and switches from one to the other as the road changes.

Ng is clear that this is not how self-driving cars are built now. The example is in the lecture because it is a supervised learning problem whose labels cost nothing except someone driving, and because the techniques that make it work — gradient descent and backpropagation — are the ones the course teaches in its first weeks and in lectures 10 and 11.

1.7. Machine learning strategy

58:01 The second area of the course, which the class notes file under learning theory, is what Ng calls machine learning strategy, and his motivation for it is drawn from practice rather than from theory. Visiting machine-learning teams at large technology companies, he finds an enormous difference in how effectively two teams apply *the same* algorithm. The failure he describes is a team that has spent six months on an approach, where a look at the data shows that it was never going to work, and that could have been seen at the start.

The decisions behind that are the everyday ones: whether to collect more data and which data, whether to try a different algorithm, whether to rent faster GPUs and train for longer, which architecture to use. 1:01:08 A learning algorithm, he says, almost never works the first time, so what separates teams is how they debug. His two analogies are both about debugging by principle rather than by reflex. One is a friend who dealt with compiler errors by deleting every line that had one. The other is the difference between an inexperienced engineer who starts optimising code at once and an experienced one who runs a profiler first and changes only the bottleneck. The aim is to turn what has been "black magic", tribal knowledge held by people with decades of experience, into an engineering discipline that can be taught. That material comes in lecture 8, the learning-theory discussion section, and lecture 12 on debugging and error analysis.

Deep learning is the third area. Ng describes it as the part of the field moving fastest and treats it as one tool among many. CS229 covers enough to train a neural network from scratch in lectures 10 and 11, and leaves the depth to CS230.

1.8. Unsupervised learning

1:04:57 Ng introduces unsupervised learning by redrawing the classification picture and deleting the labels. You are given only the inputs x, no y, and asked to find interesting structure in them. In the data on the board the structure is visible at a glance — two clusters — and an algorithm the course covers, k-means clustering, finds it. Figure 1.4 runs that algorithm on synthetic points like the board's. Starting from two arbitrary points as centres, it reaches a stable split in three rounds of reassignment.

The applications Ng lists show how much of clustering is grouping things that nobody has named. Google News gathers thousands of articles a day by different reporters into a single story, such as the BP oil-well spill. Genetic microarray data group individuals by shared characteristics. Clustering groups machines in a data centre by related workloads, finds the cohesive communities in a social network, and splits a customer database into market segments that can be addressed separately. A colleague of his has used it to group galaxies in astronomical data.

1:08:12 A student asked the natural question: is unsupervised learning just clustering? It is not. Unsupervised learning means any use of unlabelled data to find something interesting, and clustering is one instance. Ng's second example is the *cocktail party problem*. Put several microphones in a room where several people are talking at once. Each microphone records an overlapping mixture of every voice, and the task is to recover each speaker's voice cleanly.

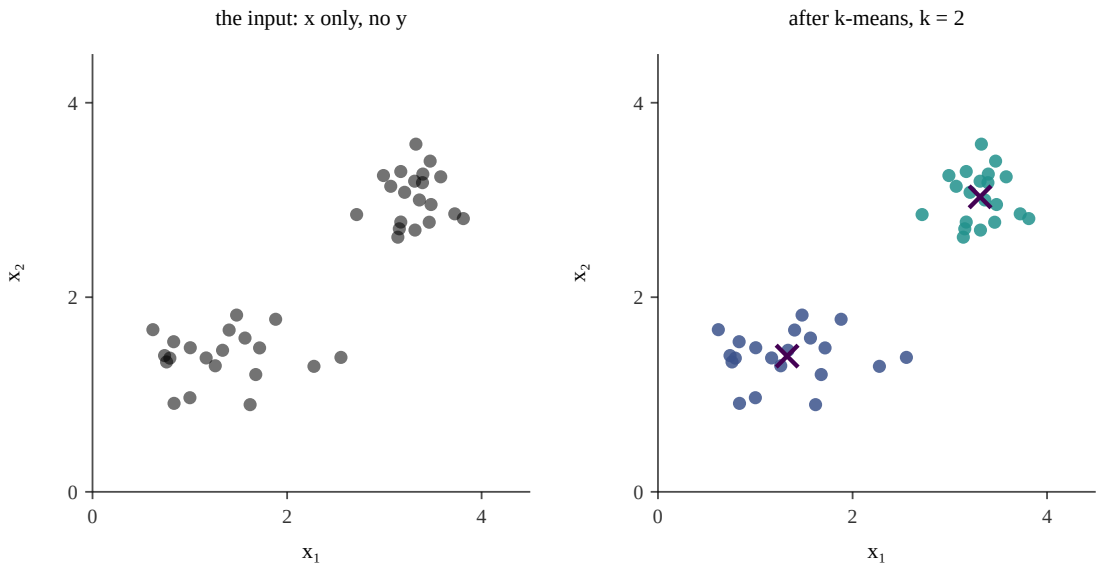


Figure 1.4.: Unsupervised learning on the same kind of picture as Figure 1.3, but without labels. Left, all the algorithm receives; right, the two groups k-means assigns, with each group's centre marked. Synthetic data. *Computed by* `figures/01-introduction/make_diagrams.py`.

Nothing labels which sound came from whom, which is what makes the problem unsupervised. The algorithm that solves it, independent component analysis, is in lectures 15 and 16 and is a programming exercise later in the course. His third example is text. From unlabelled text alone, word representations can be learned that capture analogies such as *man is to woman as king is to queen* and *Tokyo is to Japan as Washington, D.C. is to the United States*, the result popularised by Mikolov et al. (2013).

He also offered a judgement on the economics. Most of the value machine learning was creating in 2018 came from supervised learning, while unsupervised learning had important uses and was where much of the research frontier lay.

1.9. Reinforcement learning

1:11:19 The fifth area starts from a problem that defeats supervised learning: if you were handed the keys to Stanford's autonomous helicopter, how would you write a program to fly it? The difficulty is that *nobody knows* the optimal way to move a helicopter's two control sticks, so there is no y to label each state with, and no expert whose every action is worth imitating as ALVINN imitated its driver.

1:12:18 Ng's analogy is training a dog. You cannot show a dog the optimal way to behave either. You let it do what it wants and say "good dog" when it does something you like and "bad dog" when it does not, and over time it does more of the first and less of the second. Reinforcement learning replaces the label with that signal, called the *reward*. The helicopter flies however it likes, gets "good helicopter" for an accurate manoeuvre and "bad helicopter" when it crashes, and the algorithm's job is to find a way of acting that collects more reward over time. The helicopter in the video is the Stanford autonomous helicopter, whose aerobatic flight is described in Abbeel et al., 2006, and a second clip applies the same reward-driven learning to a legged robot dog learning to climb over obstacles.

The best-known successes of reinforcement learning were in games, Atari and AlphaGo. Ng said he was as excited, or more, by its growing traction in robotics and in optimising logistics systems. The course's last five lectures build it up from Markov decision processes.

1.10. The map of the course

1:11:11 Ng closes the overview with the list of the five areas. They map onto this book as follows.

Table 1.1.: The five areas of CS229 and the lectures that cover them.

Area	What it is about	Lectures
Supervised learning	learning a mapping from labelled pairs (x, y)	2–7, 9
Learning theory and strategy	why learning works, and how to make it work in practice	8, learning theory, 12
Deep learning	training neural networks	10–11
Unsupervised learning	finding structure in x alone	13–16
Reinforcement learning	learning to act from a reward signal	16–20

The order is deliberate. Supervised learning comes first because it is where both the mathematics and the economic value are most developed, and every later area borrows its notation. The next chapter starts there, with the house-price scatter of Figure 1.1 and the first algorithm for fitting it.

2. Linear Regression and Gradient Descent

Lecture 2

Lecture video: Lecture 2: Linear Regression and Gradient Descent · 1:18:17

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

The second lecture builds the first complete learning algorithm of the course and then solves the same problem a second way. Linear regression predicts a number as a weighted sum of features; the weights are chosen to make the squared error on the training set as small as possible; and that minimum can be found either by walking downhill on the error surface, one gradient step at a time, or by solving a linear system for it directly. Gradient descent is the half that matters later: the normal equations work for linear regression and its weighted variant and nothing else in the course, while the walk downhill reappears in logistic regression, generalised linear models and neural networks. Ng uses this lecture to set up the vocabulary all of them share — hypothesis, parameters, cost function, learning rate — and it is worth getting that vocabulary straight here, where the mathematics is still simple enough to check by hand.

2.1. The housing problem and the hypothesis

1:55 The running example is the one that closed lecture 1: predict the asking price of a house from its size. Ng's data are real, a few dozen listings from Portland, Oregon collected off Craigslist — a 2,104 ft² house listed at \$400,000, a 1,416 ft² one at \$232,000, and so on — and the task is to fit a straight line through them.

Before writing any formula he describes the shape every supervised algorithm in the course will have. A *training set* goes into a *learning algorithm*, and what comes out is a function, by convention called the *hypothesis* h, which takes the size of a house it has never seen and returns an estimated price. The learning algorithm's job is to produce h; h's job is to make predictions. 4:43 The first design decision is therefore not how to fit, but how to *represent* h, and for linear regression the answer is

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2, \tag{2.1}$$

with x_1 the size and x_2 the number of bedrooms. Ng concedes the point a mathematician in the room would raise: with the θ_0 term this is an affine function, not a linear one, and machine learning calls it linear anyway.

The intercept is awkward to carry around, so it is absorbed into the sum by inventing a feature that is always one, $x_0 = 1$. Then x and θ are both vectors in $\mathbb{R}^{n+1}$, and

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^\top x. \tag{2.2}$$

The rest of the notation is fixed here for the whole quarter. The θ_j are the *parameters*, and choosing them is all that learning means for this model. m is the number of training examples (rows of the table), n the number of features (two here, so x and θ are three-dimensional), x the input features and y the output or *target*. The i-th training example is written $(x^{(i)}, y^{(i)})$, and the superscript in parentheses is an index into the table, not a power: $x_1^{(1)} = 2104$ is the size of the first house. Ng is open that some of this terminology is an accident of history — he does not think "hypothesis" is a good name, but someone used it in a paper decades ago and the field is stuck with it.

2.2. The least-squares cost

14:41 "Choose θ so that $h(x)$ is close to y on the training examples" needs a number that measures closeness, and Ng builds it one piece at a time on the board: the squared difference between prediction and truth for one house, summed over all m houses, with a factor of one half in front:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 . \tag{2.3}$$

J is the *cost function*, and linear regression — ordinary least squares — is the choice of θ that minimises it. The one half changes nothing about where the minimum is; it is there so that the 2 brought down by differentiating a square cancels it, as it will in Section 2.4.

16:39 The question Ng says he is asked most often is why the error is squared rather than taken in absolute value or raised to the fourth power. His answer here is a deferral with a hint: squared error corresponds to assuming the noise in y is Gaussian, and linear regression turns out to be one member of a family, the generalised linear models, in which that choice is derived rather than assumed. The next lecture makes the Gaussian argument precise.

2.3. Gradient descent

18:27 The algorithm for minimising J starts from some initial θ — the zero vector is a reasonable default, a random start is also fine — and keeps changing θ in a way that lowers J.

19:40 Ng's picture for it is a landscape. Put θ_0 and θ_1 on the two horizontal axes and J as the height (with three parameters the surface lives in four dimensions, which is why the picture uses two). Stand at the starting point, turn through 360 degrees, and ask: if I took one tiny step, which direction would take me downhill fastest? Take that step, look around again from the new spot, and repeat.

22:03 The picture also exposes the method's weakness. On a landscape with more than one valley, where you end up depends on where you start: begin a few steps to one side and the same procedure walks you into a different local minimum (Figure 2.1). For linear regression this turns out not to matter, for a reason Section 2.5 gives, but it matters a great deal for most of what comes later in the course.

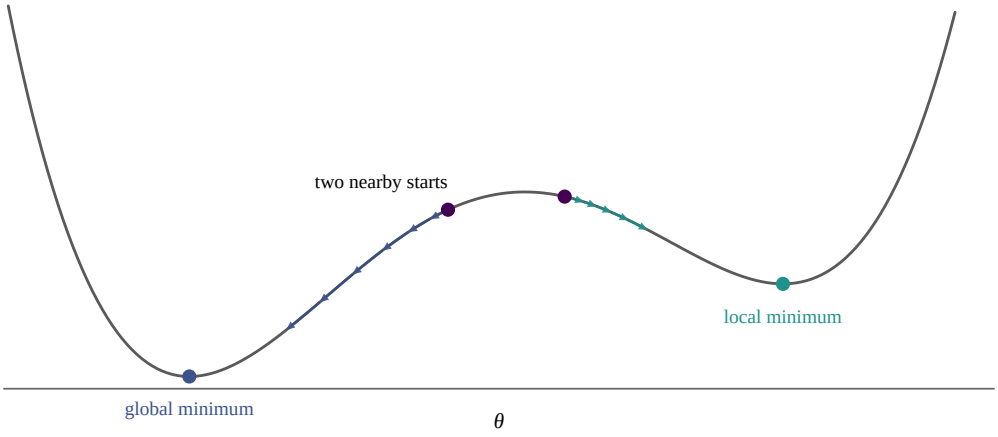


Figure 2.1.: Gradient descent on a one-parameter cost with two valleys. Started just left of the hump, it runs into the deeper valley; started just right, it stops in the shallower one and has no way of knowing a better minimum exists. *Computed by* `figures/02-linear-regression/make_diagrams.py`.

23:54 Formally, one step of gradient descent updates every parameter at once:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta), \qquad j = 0, 1, \dots, n. \tag{2.4}$$

The partial derivative gives the direction of steepest ascent, so subtracting a multiple of it goes downhill. The multiple α is the *learning rate*, which sets the size of the step. The `:=` is assignment, and Ng is deliberate about it: $a := a + 1$ increments a, whereas $a = b$ asserts a fact, and he promises never to write $a = a + 1$. Throughout the update, the training set is fixed and J is a fixed function of θ; only θ moves.

26:04 A student asked at once how to choose α. "There's a theory and there's a practice," and the practice, to laughter, is 0.01. The serious version followed: if the features are scaled to roughly $[-1, 1]$, start at 0.01 and try larger and smaller values to see which drives J down fastest. Section 2.5 says why the scaling clause is not optional.

2.4. The LMS update

27:27 To run Equation 2.4 the partial derivative has to be computed, and Ng does it for a training set of a single example (x, y) first, so that the sum does not get in the way. The chain rule brings the 2 down from the square, where it cancels the half, and leaves the derivative of what was inside:

$$\frac{\partial}{\partial \theta_j} \frac{1}{2} \left(h_\theta(x) - y\right)^2 = \left(h_\theta(x) - y\right) \frac{\partial}{\partial \theta_j} \left(\theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n - y\right) = \left(h_\theta(x) - y\right) x_j. \tag{2.5}$$

Of all the terms in the expanded sum only $\theta_j x_j$ depends on θ_j, so every other term differentiates to zero and this one to x_j.

31:43 With m examples, the derivative of a sum is the sum of the derivatives, and the full algorithm is

$$\text{repeat until convergence:} \quad \theta_j := \theta_j - \alpha \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)} \quad \text{for every } j. \tag{2.6}$$

This is the least-mean-squares or Widrow–Hoff rule, and it reads well once written out. Each example pushes θ_j in proportion to its own error: a house whose price is already predicted almost exactly barely moves the parameters, one that is badly mispredicted moves them a lot, and the push on θ_j is weighted by how large the feature x_j was for that house. The generator behind this chapter checks Equation 2.6's sum against finite differences of J on the housing data, and it agrees.

> **In the official notes**
>
> The same derivation, and the remark that the update is proportional to the error, are in cs229-notes1, section 1.

2.5. A bowl, and the learning rate

33:52 The local-minimum worry of Figure 2.1 disappears for this cost. J is a sum of squares of functions linear in θ, so it is a quadratic in θ, and a quadratic of this kind is a single bowl: its only local minimum is the global one. Looked at from above, its contours are nested ellipses, and Ng adds a fact he half-remembers from school — "I think it's true" — that each step of steepest descent crosses the contour it starts on at right angles. It is true: the gradient is normal to the level set through any point, and the generator confirms it on the housing cost to within a cosine of 0.003.

What the board picture does not show is how *thin* the bowl can be. Figure 2.2 runs gradient descent on synthetic housing data shaped like Ng's, with size in thousands of square feet, starting from $\theta = 0$. The first step drops almost straight across the contours into the floor of a long valley; every step after that inches along the floor towards the minimum, and it takes more than four hundred of them to bring the intercept within \$500 of its final value. The ratio of the bowl's steepest to its shallowest curvature — the eigenvalues of $X^\top X$, in the notation of Section 2.8 — is about a hundred here, and gradient descent's pace is set by the shallow direction while its step size is capped by the steep one.

That cap is the precise version of Ng's 36:08 advice on the learning rate. Too small and the algorithm needs very many iterations; too large and each step overshoots the minimum, lands higher on the far side, and diverges (Figure 2.3). For a quadratic the boundary is exact: with $\lambda_{\max}$ the largest eigenvalue of $X^\top X$, gradient descent converges for $\alpha < 2/\lambda_{\max}$ and diverges above it, which the generator checks on both sides of the line. His practical rules follow from watching J rather than computing eigenvalues. If J goes *up* over iterations, the learning rate is almost certainly too large. And rather than search finely, try values on a roughly doubling scale — 0.01, 0.02, 0.04, 0.08 — and keep whichever drives J down fastest.

The same bound explains the feature-scaling clause in "set it to 0.01". On the synthetic data with size measured in raw square feet, $\lambda_{\max}$ is so large that the stable range ends at $\alpha \approx 4 \times 10^{-9}$, and 0.01 diverges immediately; with size rescaled to about $[-1, 1]$ the limit is about 0.04, and 0.01 is comfortably safe. One further caveat follows from writing J as a sum. $\lambda_{\max}$ grows with m, so the safe learning rate shrinks as the training set grows, and 0.01 that works on 47 houses

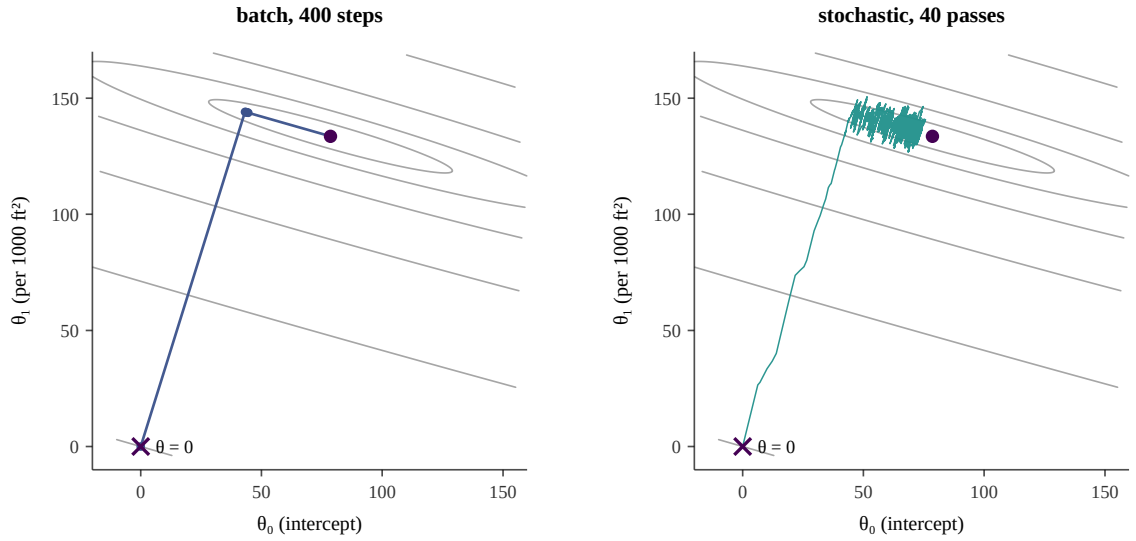


Figure 2.2.: Batch and stochastic gradient descent on the contours of the least-squares cost, both from $\theta = 0$, on 47 synthetic houses shaped after the Portland data. Batch descent (left) reaches the valley floor in one step and then crawls along it; the stochastic version (right) takes the same route noisily and keeps moving near the minimum, marked by the dot. *Computed by* `figures/02-linear-regression/make_diagrams.py`.

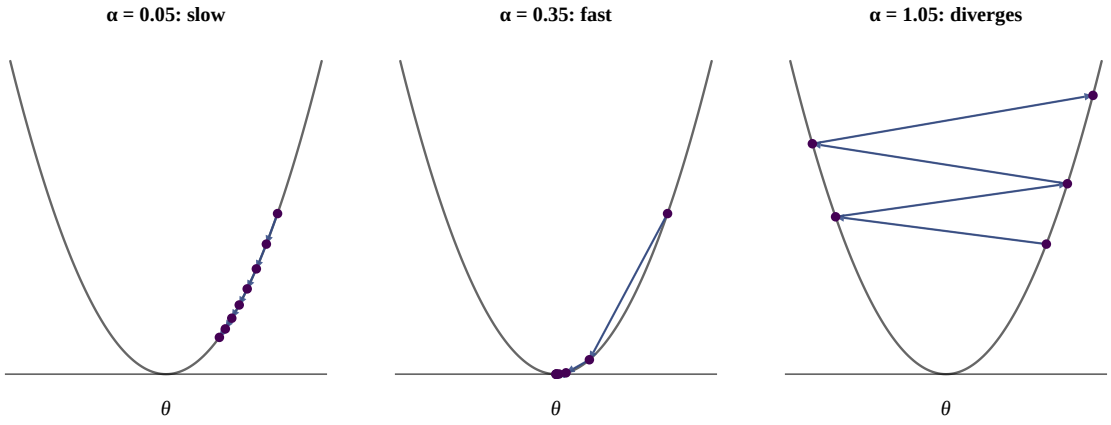


Figure 2.3.: Gradient descent on $J(\theta) = \theta^2$, where the update multiplies θ by $1 - 2\alpha$. A small learning rate creeps down one side; a moderate one reaches the bottom in a few steps; above $\alpha = 1$ every step overshoots to the other side, higher than before. *Computed by* `figures/02-linear-regression/make_diagrams.py`.

will not on 47,000. Many implementations average the squared error instead of summing it, which removes that dependence; the minimum is unchanged either way.

37:14 Ng's second view of the same run is in the data rather than in parameter space. With $\theta = 0$ the hypothesis is the horizontal line $h(x) = 0$, predicting every house free; each step of gradient descent replaces it with a line that fits a little better, until it settles on the least-squares fit (Figure 2.4).

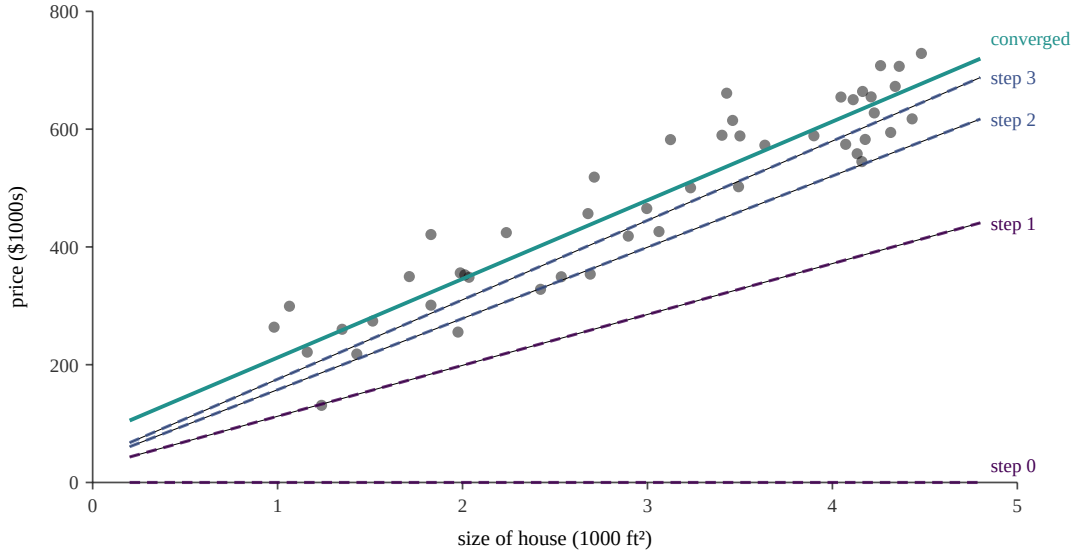


Figure 2.4.: The hypothesis after successive steps of batch gradient descent from $\theta = 0$, drawn over the synthetic housing data; the solid line is the least-squares fit the steps converge to. *Computed by* `figures/02-linear-regression/make_diagrams.py`.

39:30 A student asked why the update *subtracts* α times the gradient rather than adding it. Ng's suggestion was to work one case by hand, and it settles the matter: take $J(\theta) = \theta^2$ at $\theta = 1$. The derivative there is 2 (Ng said 1 at the board; the sign is what matters), positive, and to reach the minimum at zero θ has to decrease — so the step must go against the gradient. Adding it would climb.

2.6. Stochastic gradient descent

41:05 Equation 2.6 is called *batch* gradient descent because every single update looks at the whole training set as one batch. That is its weakness. If m is the size of the US census rather than of a Craigslist scrape — Ng mentions training on hundreds of millions of examples — then one step requires a full pass over the data, possibly terabytes read from disk, before θ moves at all, and convergence can take hundreds or thousands of such steps.

44:13 *Stochastic* gradient descent updates after every example instead:

$$\text{loop: for } i = 1, \dots, m: \quad \theta_j := \theta_j - \alpha \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \quad \text{for every } j. \tag{2.7}$$

46:02 In Ng's words it looks at one house and adjusts the parameters to predict that house better, then the next house, then the next. No single house's gradient points straight at the minimum, so the path is noisy, but on average it heads downhill, and it starts making progress

after the first example rather than after the millionth. The price is that it never quite converges: because each update chases a different house, θ keeps wandering around the minimum, as on the right of Figure 2.2. For a large dataset that trade is almost always worth making.

The questions from the room fill in how it is used in practice. Asked 47:58 whether one could start stochastic and switch to batch near the end, Ng said yes, and that it does no harm — but that people rarely bother, because batch steps on a large dataset are too slow to be worth it, and because the parameters stochastic gradient descent ends near are usually perfectly good even if they are not exactly the global minimum. The middle ground, which he names and leaves to CS230, is *mini-batch* gradient descent, which updates on a hundred or so examples at a time. What is common instead is 50:03 to decrease the learning rate slowly over time: the oscillations shrink with the step size, so θ ends up bouncing around a smaller region closer to the minimum. Asked when to stop, his answer was to plot J over time and stop when it stops going down — easy here, where there are no local optima to confuse the picture, and much harder for the neural networks later in the course.

51:35 The last question was whether stochastic gradient descent's learning rate should be $1/m$ times batch's. No, Ng said, it is usually much bigger, since otherwise it would be as slow as batch. The arithmetic behind that answer is worth having. Because J is a sum, a full pass of Equation 2.7 at rate α adds up, to first order in α, to one batch step at the same α; the generator confirms the gap between them shrinking as α does. Dividing by m would therefore make a whole pass of stochastic updates worth only $1/m$ of a batch step. And a single-example update tolerates a far larger rate in the first place: its stability limit is $2/\|x^{(i)}\|^2$ rather than $2/\lambda_{\max}$ of the whole of $X^\top X$, which on the synthetic housing data is a factor of about 25 more room.

51:51 Ng's own rule of thumb closes the topic. With a dataset of hundreds or thousands of examples, where a pass is cheap, he would almost always use batch gradient descent, because it is one less thing to fiddle with — no oscillating parameters to worry about. When the data are too large for that, almost everyone uses stochastic gradient descent or something like it.

2.7. Matrix derivatives and the trace

53:25 For linear regression, and for exactly linear regression — none of the models of the following lectures beyond its weighted variant in lecture 3 — there is a way to skip the iterations altogether and jump to the minimum in one step. Deriving it means setting the gradient of J to zero and solving, and Ng's aim is to do that "in roughly four lines of linear algebra rather than pages and pages", which takes a little notation first. In his experience, he says, the right notation is what makes such problems tractable.

56:02 For a function $J : \mathbb{R}^{n+1} \to \mathbb{R}$, the derivative with respect to the vector θ is the vector of partial derivatives,

$$\nabla_\theta J = \begin{bmatrix} \partial J/\partial \theta_0 \\ \vdots \\ \partial J/\partial \theta_n \end{bmatrix}, \tag{2.8}$$

and more generally, for a function f that maps an $m \times n$ matrix A to a real number, $\nabla_A f(A)$ is the $m \times n$ matrix whose (i, j) entry is $\partial f/\partial A_{ij}$. Ng's example 58:32 is a made-up function of a 2×2 matrix, $f(A) = A_{11} + A_{12}^2$, so that f applied to the matrix with rows $(5, 6)$ and $(7, 8)$ is $5 + 36 = 41$, and

$$\nabla_A f(A) = \begin{bmatrix} 1 & 2A_{12} \\ 0 & 0 \end{bmatrix}. \tag{2.9}$$

A student asked whether the vector definition was a special case of the matrix one; it is — a vector is an $(n+1) \times 1$ matrix, and the two agree.

1:03:35 The second tool is the *trace* of a square matrix, the sum of its diagonal entries, $\operatorname{tr} A = \sum_i A_{ii}$. Transposing a matrix leaves its diagonal where it was, so $\operatorname{tr} A = \operatorname{tr} A^\top$. The trace of a product is unchanged by moving the last factor to the front, $\operatorname{tr} AB = \operatorname{tr} BA$ and $\operatorname{tr} ABC = \operatorname{tr} CAB$, and its derivative has a clean form:

$$\nabla_A \operatorname{tr} AB = B^\top. \tag{2.10}$$

1:06:23 Ng then gives one identity as the matrix analogue of $\frac{d}{da} a^2 c = 2ac$, a rule to be used without re-deriving it every time, the way one uses the derivative of x^2. The analogy is a good way to remember the shape of the result, but the generator caught that the shorthand needs a condition the board left out:

$$\nabla_A \operatorname{tr}(AA^\top C) = (C + C^\top) A, \tag{2.11}$$

which equals $2CA$ only when C is symmetric. For a random non-symmetric C, the numerical gradient matches Equation 2.11 and misses $2CA$ by a visible margin; for the symmetric part of the same C the two coincide. In the derivation that follows A is θ and the matrix playing the role of C is $X^\top X$, which is symmetric, so nothing goes wrong, but the general rule is the one with $C + C^\top$. The official notes state the fully general version, $\nabla_A \operatorname{tr} ABA^\top C = CAB + C^\top AB^\top$, which reduces to Equation 2.11 when B is the identity.

2.8. The normal equations

1:08:39 To write J with matrices, stack the training inputs as the rows of the *design matrix* X, an $m \times (n+1)$ matrix whose i-th row is $(x^{(i)})^\top$, and stack the targets into a vector $\vec{y} \in \mathbb{R}^m$. Then $X\theta$ is the vector of every prediction at once, since its i-th entry is $(x^{(i)})^\top \theta = h_\theta(x^{(i)})$, and $X\theta - \vec{y}$ is the vector of every error. A vector dotted with itself is the sum of its squared entries, $z^\top z = \sum_i z_i^2$, so

$$J(\theta) = \frac{1}{2} (X\theta - \vec{y})^\top (X\theta - \vec{y}). \tag{2.12}$$

1:13:08 Now take the gradient and set it to zero. Expanding the product the way one expands $(ax - b)^2$ gives four terms, $\theta^\top X^\top X\theta - \theta^\top X^\top \vec{y} - \vec{y}^\top X\theta + \vec{y}^\top \vec{y}$. The last does not depend on θ; the middle two are the same scalar, each the transpose of the other; and the trace rules of Section 2.7 differentiate the first to $2X^\top X\theta$ and the middle pair to $2X^\top \vec{y}$. Halving,

$$\nabla_\theta J(\theta) = X^\top X\theta - X^\top \vec{y}. \tag{2.13}$$

This is Equation 2.6's sum written in matrix form, which is a useful check on both. 1:16:19 Setting it to zero gives the *normal equations*,

$$X^\top X\theta = X^\top \vec{y}, \qquad \theta = \left(X^\top X\right)^{-1} X^\top \vec{y}. \tag{2.14}$$

A few matrix products land on the global minimum of the bowl in one step, the same point that gradient descent spends hundreds of steps approaching in Figure 2.2; the generator runs both and confirms they agree to six decimal places. On the real Portland data the official notes report the fit $\theta_0 = 71.27$, $\theta_1 = 0.1345$: about \$71,000 plus \$134.50 per square foot.

1:17:01 The common question is what to do if $X^\top X$ is not invertible. That usually means the features are linearly dependent — redundant — and the matrix pseudo-inverse will still return a sensible answer. But Ng's preferred fix is diagnostic: a singular $X^\top X$ most often means the same feature is in the data twice (size in square feet and size in square metres, say), and the right move is to find the duplicate and drop it. With a feature duplicated, every way of splitting its weight between the two copies gives exactly the same predictions and the same J, which is why no single θ can be picked out by the equations; the generator checks this on a copy of the housing data with the size column repeated.

His last answer, to a final question on how to choose the learning rate, came back to where Section 2.3 left it: it is empirical, and most people try several values and keep the one that works.

> **In the official notes**
>
> The matrix-derivative identities and the full derivation of the normal equations, including the trace manipulation Ng skips, are in cs229-notes1, sections 2.1 and 2.2. Section 3 justifies the least-squares cost as maximum likelihood under Gaussian noise, which is where the next lecture begins.

3. Locally Weighted and Logistic Regression

Lecture 3

Lecture video: Lecture 3: Locally Weighted and Logistic Regression · 1:19:34

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

This lecture takes the linear regression of lecture 2 in three directions. Locally weighted regression keeps the least-squares machinery but refits it around every query point, which lets a straight-line method trace a curve no single set of features would fit. A probabilistic reading of linear regression then shows that least squares is not an arbitrary choice: it is what maximum likelihood gives when the noise is Gaussian. And the same two-step recipe — assume a distribution for y given x, then maximise the likelihood — produces logistic regression, the first classification algorithm of the course, along with Newton's method to fit it in a handful of iterations. Ng ranks linear and logistic regression as the two learning algorithms he uses most.

3.1. Features, or a different algorithm

2:20 Lecture 2 fitted a straight line to house prices, but the data may bend. One answer is to change the features: fit $\theta_0 + \theta_1 x + \theta_2 x^2$ by defining $x_2 = x^2$, and the linear-regression machinery runs unchanged, because "linear" refers to the parameters, not to x. The trouble is choosing. A quadratic eventually curves back down, which for house prices says a bigger house is cheaper; $\sqrt{x}$ or $\log x$ might bend the right way; and for data that rise, dip and rise again, Ng says frankly that finding the right set of features — square root, log, cube, $x^{2/3}$ — is hard. Feature-selection algorithms that automate the choice come later in the course.

5:53 Locally weighted regression sidesteps the question, and it introduces a distinction worth having early. A *parametric* learning algorithm fits a fixed set of parameters, after which the training set can be erased: linear regression keeps only θ. A *non-parametric* algorithm has to keep the data, and the amount it stores grows — linearly here — with the size of the training set. That is the cost of not having to choose features, and it is why Ng warns that such methods may not suit a really massive dataset.

3.2. Locally weighted regression

8:33 Linear regression makes a prediction at x by fitting θ once to all the data and returning $\theta^\top x$. Locally weighted regression postpones the fit until it knows where the prediction is wanted. Given the query point x, it fits a straight line that pays most attention to the training examples with inputs near x, glances at the ones further away, and uses that local line to predict at x; a different query point gets a different line. Formally it chooses θ to minimise a weighted cost,

$$\sum_{i=1}^{m} w^{(i)} \left(y^{(i)} - \theta^{\top} x^{(i)}\right)^2, \qquad w^{(i)} = \exp\left(-\frac{\left(x^{(i)} - x\right)^2}{2\tau^2}\right), \tag{3.1}$$

and returns $\theta^{\top} x$. 12:09 When $x^{(i)}$ is close to the query the weight is near $e^0 = 1$ and that example's error counts in full; when it is far the weight is near zero and its error term all but disappears from the sum. (Ng first wrote the exponent without the square on τ; a student caught it, and it is τ^2.) The left panel of Figure 3.1 shows one such fit.

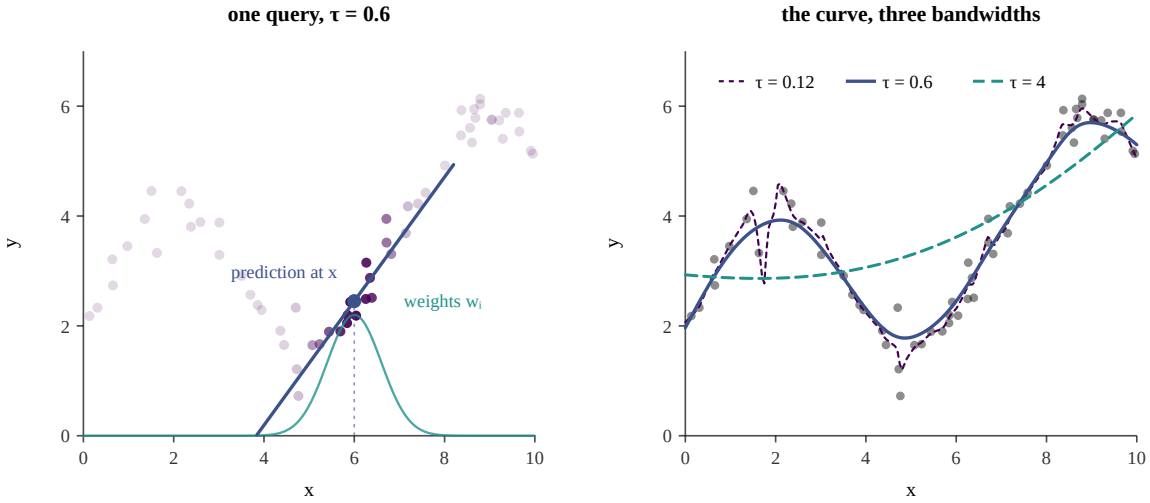


Figure 3.1.: Locally weighted regression on synthetic data. Left: at the query $x = 6$, points are shaded by their weight, the bump below is the weight function, and the line is the weighted least-squares fit whose value at $x = 6$ is the prediction. Right: repeating that at every x traces the whole curve, for three bandwidths. *Computed by* `figures/03-locally-weighted-logistic/make_diagrams.py`.

15:27 The weight function is shaped like a Gaussian bell, and Ng heads off the obvious confusion: it has nothing to do with a Gaussian density. Its peak is 1, it does not integrate to 1, and it is not a probability of anything — just a convenient bump. The bump's width is the *bandwidth* τ, a hyperparameter, and 16:46 it decides how large a neighbourhood each local fit sees. Ng leaves its effect for the problem set, saying only that it controls overfitting and underfitting; the right panel of Figure 3.1 shows both. With $\tau = 0.12$ only a couple of points carry weight in each fit, and the curve chases the noise into spikes; with $\tau = 4$ nearly every point carries weight everywhere, and the curve flattens towards a single straight line. In the limit it *is* that line: the generator checks that a bandwidth of 10^4 reproduces the ordinary least-squares fit to four decimal places. At $\tau = 0.6$ the curve follows the rise, the dip and the plateau without being told what features describe them.

The questions that followed are the practical ones. Asked 18:53 about predicting outside the range of the data, Ng said the formulas still work but the results may be poor — locally weighted regression is not good at extrapolation, but then most learning algorithms are not. Asked about letting τ vary across the input space, he said yes: there is a large literature on choosing it from the local density of points, and on replacing the bell with other shapes, such as a triangle that reaches zero outside a window. His own rule 19:58 is to reach for locally weighted regression when the number of features is small, two or three, there is a lot of data, and you do not want to think about which features to use.

20:29 On cost, his answer at the board needs one correction. Each prediction solves a weighted version of the normal equations, $X^\top W X\theta = X^\top W\vec{y}$ with W the diagonal matrix of weights, and that is a system of dimension $n+1$ — the number of parameters — not of the number of training examples, as the lecture puts it. What does grow with m is the work of *forming* the system, since every training example contributes a weighted term, and that work is repeated for every query. So a prediction costs time proportional to m, which is why a few thousand examples are no problem and millions call for data structures such as k-d trees that find the nearby points without visiting all of them.

> **i** In the official notes
>
> Locally weighted linear regression, with the same weight function, is cs229-notes1, section 4; the problem set explores the bandwidth.

3.3. Least squares as maximum likelihood

22:24 Lecture 2 promised a reason for squaring the error. Ng's answer starts by assuming how the data were generated: each house's price is a linear function of its features plus an error,

$$y^{(i)} = \theta^\top x^{(i)} + \epsilon^{(i)}, \qquad \epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2), \tag{3.2}$$

where $\epsilon^{(i)}$ gathers everything the model leaves out — the seller's mood that day, unmodelled features — and random noise. The density of the error is the Gaussian $\frac{1}{\sqrt{2\pi}\sigma}\exp(-\epsilon^2/2\sigma^2)$, which, unlike the bump of Section 3.2, does integrate to one. 25:45 The large assumption is that the errors are IID — independent and identically distributed across houses — and Ng is candid that it is false: if one house on a street is priced unusually high, its neighbours probably are too. It is one of those assumptions that is probably not true and is good enough to give a useful model.

Together the assumptions say that, given x and θ, the price is Gaussian around the line:

$$p\left(y^{(i)} \mid x^{(i)}; \theta\right) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{\left(y^{(i)} - \theta^\top x^{(i)}\right)^2}{2\sigma^2}\right). \tag{3.3}$$

28:16 The semicolon is read "parameterised by". Writing a comma would mean conditioning on θ, which is not a random variable in this frequentist view; Ng keeps the distinction, he says, mostly to avoid upsetting statistician friends, and admits it changes nothing about the code one writes.

30:57 Because the errors are independent, the probability of all the observed prices is the product of the individual densities, and viewed as a function of θ with the data fixed that product is the *likelihood*,

$$L(\theta) = \prod_{i=1}^{m} p\left(y^{(i)} \mid x^{(i)}; \theta\right). \tag{3.4}$$

32:44 Ng's vocabulary rule is worth adopting. The likelihood of the parameters and the probability of the data are the same number; which name you use says which argument you are varying.

So one speaks of the likelihood *of the parameters* and the probability *of the data*, never the reverse.

37:49 Maximum likelihood estimation chooses the θ that makes the observed data most probable. The logarithm is strictly increasing, so it is easier and equivalent to maximise the log-likelihood $\ell(\theta) = \log L(\theta)$, which turns the product into a sum:

$$\ell(\theta) = m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^\top x^{(i)} \right)^2 . \tag{3.5}$$

The first term does not involve θ, and σ^2 is a positive constant, so maximising ℓ is exactly minimising the least-squares cost $J(\theta)$ of lecture 2 — and the answer does not depend on σ at all. Least squares is maximum likelihood under IID Gaussian noise.

Asked 35:31 why the noise should be Gaussian, Ng pointed to the central limit theorem: an error made of many small, not-too-correlated contributions — the seller's mood, the school district, the weather, transport — sums to something close to Gaussian, which is why it became the default. The same derivation also answers lecture 2's question about absolute error, which Ng deferred. Assume instead that the errors follow a Laplace distribution, with density proportional to $\exp(-|\epsilon|/b)$, and the log-likelihood becomes a constant minus a multiple of $\sum_i |y^{(i)} - \theta^\top x^{(i)}|$: maximum likelihood is then least *absolute* deviations. The difference is not cosmetic. With an intercept alone the Gaussian answer is the mean and the Laplace answer is the median, and the generator checks both against a grid search on a sample of twenty values around 3 plus one at 25: the Gaussian estimate is pulled to 4.09 by the single outlier, the Laplace one stays at 2.93. The choice of noise model is a choice of how much to trust the extreme points.

Asked 41:07 whether some other cost would ever be better, Ng said that under these assumptions none is — the derivation makes least squares the answer. If you know the errors are badly non-IID, more sophisticated models exist, but in practice he rarely bothers: assumptions known to be not quite true are often kept because they make the algorithm efficient.

> **In the official notes**
>
> The same derivation is cs229-notes1, section 3, which also notes that the conclusion does not depend on σ.

3.4. Why not linear regression for classification

43:42 The recipe — assume $p(y \mid x; \theta)$, then maximise the likelihood — now goes to a problem where y is 0 or 1: binary classification, with lecture 1's tumours as the example. The tempting shortcut is to fit a straight line to the 0/1 labels and call anything above 0.5 a 1. Ng's objection 44:43 is a single extra example. Add one malignant tumour far to the right, a point whose label nobody would dispute, and the least-squares line tilts to reach it; the place it crosses 0.5 slides right, and malignant tumours that were correctly classified are now called benign. In Figure 3.2 the threshold moves from 3.17 to 4.11, and the number of misclassified tumours goes from one to three, because of a point that contained no new information about where the boundary is. There is also the unnatural fact that a line outputs values below 0 and above 1 for something that can only be 0 or 1. Ng says some people use linear regression for classification and sometimes get lucky; he never does.

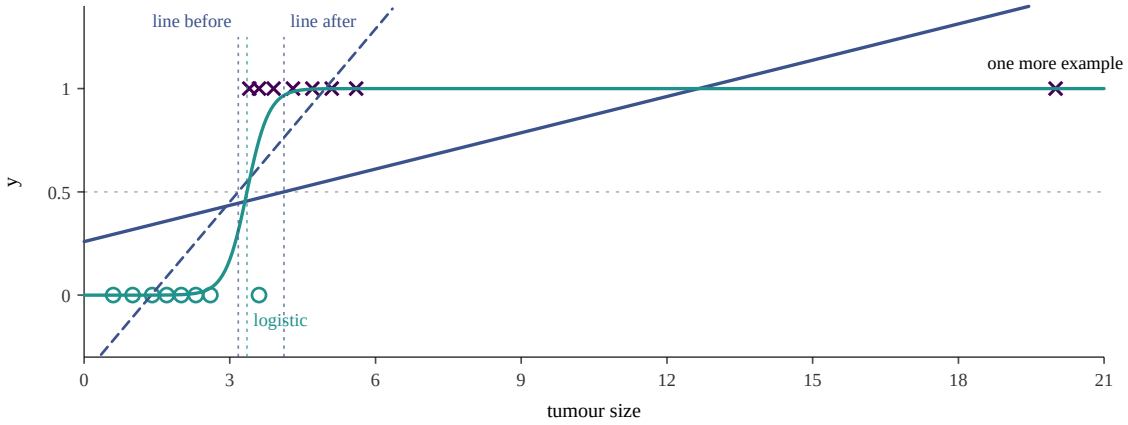


Figure 3.2.: One malignant tumour at size 20 drags the least-squares line (dashed without it, solid with it), moving the size at which it crosses 0.5 past three malignant tumours. The logistic regression curve, fitted with the extra example, keeps its boundary at the same place. Synthetic data. *Computed by* `figures/03-locally-weighted-logistic/make_diagrams.py`.

3.5. The logistic model

46:46 A classifier's hypothesis should output a number between 0 and 1, so logistic regression passes the linear function through the *sigmoid* or *logistic* function — the two names mean the same thing:

$$h_\theta(x) = g(\theta^\top x) = \frac{1}{1 + e^{-\theta^\top x}}, \qquad g(z) = \frac{1}{1 + e^{-z}}. \tag{3.6}$$

g starts near 0 for very negative z, passes through $\frac{1}{2}$ at $z = 0$ and approaches 1. Ng is open that he is pulling it "out of a hat" for now: many functions have that shape, and lecture 4 derives this one from the generalised linear model framework. Section 3.6 adds a reason of its own.

50:23 The probabilistic assumption is that the hypothesis *is* the probability of the positive class — the chance that the tumour is malignant:

$$P(y = 1 \mid x; \theta) = h_\theta(x), \qquad P(y = 0 \mid x; \theta) = 1 - h_\theta(x). \tag{3.7}$$

A student asked whether the roles could be swapped; they could, and nothing would change except which class is called positive. 52:46 Since y is 0 or 1, the two lines compress into one,

$$p(y \mid x; \theta) = h_\theta(x)^{\,y} \left(1 - h_\theta(x)\right)^{1-y}, \tag{3.8}$$

because whichever factor carries a zero exponent becomes 1 and drops out, leaving the line that applies. The trick is only notation, but it is what makes the likelihood a single product.

3.6. Maximum likelihood and gradient ascent

56:03 With Equation 3.8 the likelihood is $\prod_i h(x^{(i)})^{y^{(i)}}(1-h(x^{(i)}))^{1-y^{(i)}}$, and its log is

$$\ell(\theta) = \sum_{i=1}^{m} y^{(i)} \log h_\theta\big(x^{(i)}\big) + \big(1-y^{(i)}\big) \log\big(1-h_\theta\big(x^{(i)}\big)\big)\,. \tag{3.9}$$

To fit, 58:54 Ng uses *batch gradient ascent*: the same walk as lecture 2, but uphill, since the goal is now to maximise ℓ rather than minimise J, so the minus sign in the update becomes a plus. Working out the derivative, which Ng leaves to the notes, gives

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \big(y^{(i)} - h_\theta\big(x^{(i)}\big)\big)\, x_j^{(i)}. \tag{3.10}$$

The generator checks this gradient against finite differences of Equation 3.9.

1:03:06 Ng then put a question to the room. Flip the sign and swap the two terms in the bracket, and Equation 3.10 is character for character the LMS rule of lecture 2 — yet the lecture had just argued that linear regression is wrong for classification. What changed? The hypothesis: h_θ is now $g(\theta^\top x)$, not $\theta^\top x$, so the same-looking update does something different. The resemblance is not a coincidence; it holds for the whole family of generalised linear models, and Ng adds that the likeness caused priority disputes in the field's early history, between people who thought the algorithms were one and people who saw they were not.

1:02:15 Asked whether ℓ has local maxima, Ng said no: for logistic regression the log-likelihood is concave, so the only maximum is the global one. The Hessian makes that checkable, $\nabla^2\ell = -\sum_i h(1-h)\,x^{(i)}x^{(i)\top}$, a negative sum of outer products, and the generator confirms it is negative semidefinite at two hundred random parameter values. He then gave this as *another reason* to choose the logistic function over other S-shaped functions — that it guarantees a single maximum. That is true but not unique to it, and the generator says where the property actually comes from. ℓ is concave because $\log g(z)$ and $\log(1-g(z))$ are both concave in z, and that is a property of the S-curve, not of logistic regression. The normal CDF, which gives *probit* regression, has it too. The Cauchy CDF, $\frac{1}{2} + \arctan(z)/\pi$, an S-curve with heavy tails, does not: its log bends upward in the left tail (Figure 3.3), and on a three-example dataset the resulting log-likelihood has positive curvature at $\theta = -0.9$, so the argument that rules out local maxima no longer applies. The logistic function is one of a class of choices with a single maximum, not the only one; its claim to being *the* choice is the generalised-linear-model derivation of lecture 4.

Asked 1:04:53 whether logistic regression has an equivalent of the normal equations, Ng's short answer was no: there is no known closed form for the maximising θ, so an iterative method is always needed.

> **In the official notes**
>
> The derivative of the sigmoid, $g'(z) = g(z)(1-g(z))$, and the derivation of Equation 3.10 from it are in cs229-notes1, section 5.

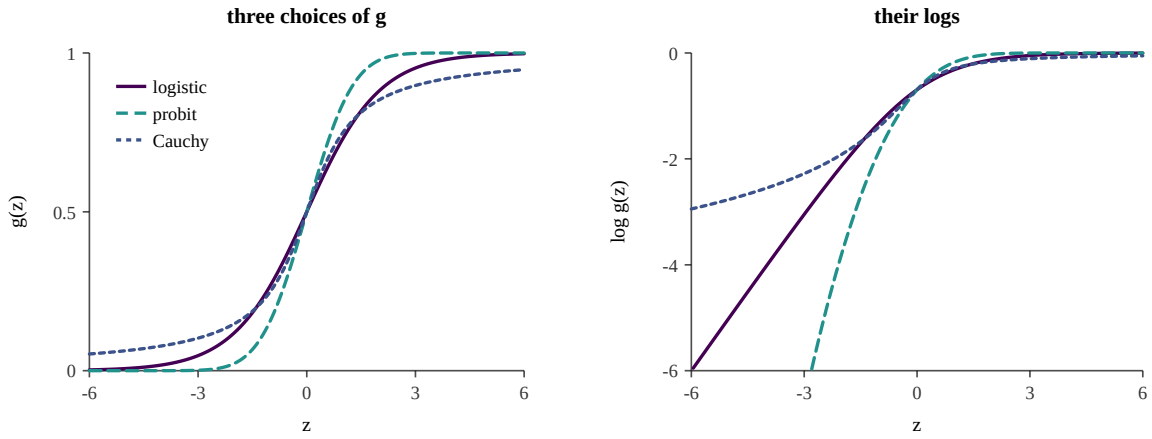


Figure 3.3.: Left: three S-shaped choices of g — logistic, the normal CDF (probit) and the Cauchy CDF. Right: their logarithms. The first two bend downward everywhere, which is what makes the log-likelihood concave; the Cauchy curve bends upward in its left tail. *Computed by* `figures/03-locally-weighted-logistic/make_diagrams.py`.

3.7. Newton's method

1:05:36 Gradient ascent takes baby steps. Ng's comparison is a problem that needs a hundred or a thousand gradient iterations and only ten of Newton's method, each of which costs more. To explain it he solves a different problem first: given a function f of a single real θ, find where $f(\theta) = 0$. Maximising ℓ is such a problem, since at the maximum $\ell'(\theta) = 0$, so in the end f will be ℓ'.

1:08:33 The picture is a sequence of tangents. From the current guess, draw the line tangent to f and follow it down to where it crosses zero; that crossing is the next guess. Repeat, and within a few iterations the guesses sit on the root (Figure 3.4). If the tangent at θ_0 meets the axis a horizontal distance Δ away, its slope is the height over the run, $f'(\theta_0) = f(\theta_0)/\Delta$, so $\Delta = f(\theta_0)/f'(\theta_0)$ and

$$\theta^{(t+1)} := \theta^{(t)} - \frac{f\left(\theta^{(t)}\right)}{f'\left(\theta^{(t)}\right)} \quad \Longrightarrow \quad \theta^{(t+1)} := \theta^{(t)} - \frac{\ell'\left(\theta^{(t)}\right)}{\ell''\left(\theta^{(t)}\right)}. \tag{3.11}$$

The update uses only where ℓ' vanishes, so it is the same whether the goal is a maximum or a minimum — the official notes leave this as a question, and that is the answer. What makes it the right tool here is that ℓ is concave, so its only stationary point is the maximum.

1:14:22 The speed has a name, *quadratic convergence*, which Ng calls "not a great name". Informally, once Newton's method is close, the error is roughly squared at each step: an error of 0.01 becomes about 0.0001 and then about 0.00000001, so the number of correct digits doubles per iteration. On the problem of Figure 3.4 the generator's errors run 0.053, then 0.0014, then 9×10^{-7}. The doubling only starts near the root; from the start at zero the first few steps shrink the error more slowly.

1:16:03 With θ a vector, the first derivative becomes the gradient and the second the *Hessian* H, the $(n+1) \times (n+1)$ matrix of second partial derivatives $H_{ij} = \partial^2 \ell / \partial\theta_i \partial\theta_j$:

$$\theta := \theta - H^{-1} \nabla_\theta \ell(\theta). \tag{3.12}$$

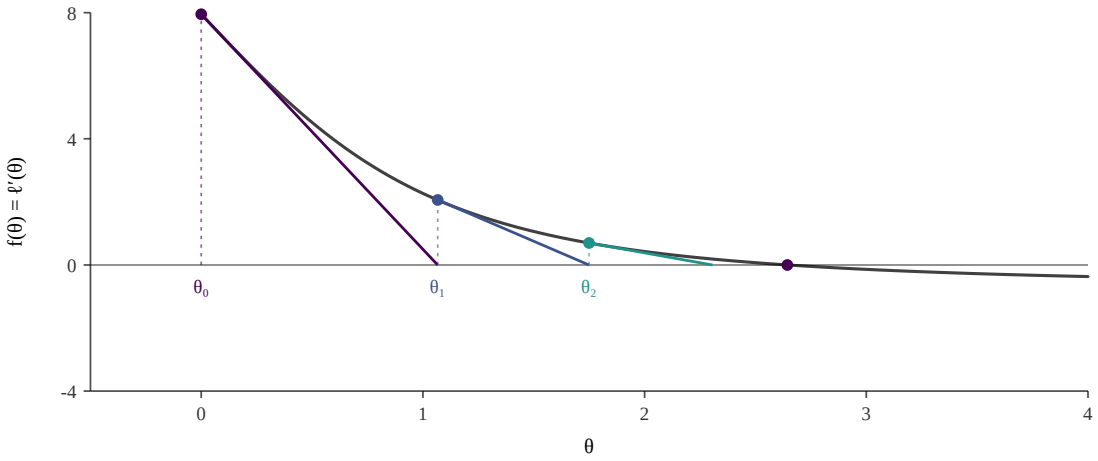


Figure 3.4.: Newton's method on $f(\theta) = \ell'(\theta)$, the derivative of a one-parameter logistic log-likelihood on synthetic data. Each tangent from the current guess meets the axis at the next guess; the gap to the root shrinks faster at every step, and by the fifth the guess has all but reached it, marked by the dot. *Computed by* `figures/03-locally-weighted-logistic/make_diagrams.py`.

1:17:22 That matrix is the catch. Each iteration solves a linear system in it, or inverts it, which is nothing for ten parameters and prohibitive for a hundred thousand. Ng's rule of thumb follows: with ten or fifty parameters he would very likely use Newton's method and be done in a few iterations; with ten thousand or more he would use gradient descent. The gap in iteration counts can be far larger than his hundred-to-ten. On the tumour data of Figure 3.2, which are nearly separable and so put the maximum far out at $\theta \approx (-14.9, 4.4)$, Newton's method reaches it to six decimal places in 9 iterations; gradient ascent with the safe fixed step $\alpha = 4/\lambda_{\max}(X^\top X)$ needs about 117,000.

> **In the official notes**
>
> Newton's method and its vector form are cs229-notes1, section 7, which adds that Newton's method applied to the logistic log-likelihood is also called Fisher scoring.

4. Perceptron and Generalized Linear Models

Lecture 4

Lecture video: Lecture 4: Perceptron and Generalized Linear Model · 1:22:02

Based on the lecture by Anand Avati — CS229, Stanford University, Autumn 2018.

Lectures 2 and 3 derived linear regression and logistic regression separately, and both ended with the same update, $\theta_j := \theta_j + \alpha\,(y - h_\theta(x))\,x_j$. This lecture, given by Anand Avati while Ng was away, explains why. Both models are instances of a *generalized linear model*: pick a distribution for y from the exponential family, let its natural parameter be a linear function of x, and the hypothesis, the likelihood and the update all follow — the same update every time, with only h_θ changing. The sigmoid, which lecture 3 introduced without justification, falls out of this recipe once y is assumed Bernoulli. The lecture opens with the perceptron, which has the same update for a different reason, and closes with softmax regression, which extends logistic regression to many classes.

4.1. The perceptron

1:35 The perceptron replaces logistic regression's sigmoid with a hard threshold, $g(z) = 1$ if $z \geq 0$ and 0 otherwise, so $h_\theta(x) = g(\theta^\top x)$ outputs a label rather than a probability. The update is the one from lecture 3, written for one example at a time:

$$\theta := \theta + \alpha\left(y^{(i)} - h_\theta(x^{(i)})\right)x^{(i)}. \tag{4.1}$$

Avati reads it from the inside out. Since y and $h_\theta(x)$ are both 0 or 1, the bracket can take only three values: 0 when the prediction is right, so nothing happens; $+1$ when a positive example was called negative; and -1 in the opposite case. A correctly classified example therefore leaves θ alone, and a misclassified one adds or subtracts a multiple of itself.

7:27 The geometric picture makes that rule look inevitable rather than arbitrary. The decision boundary $\theta^\top x = 0$ is the line perpendicular to θ, and $\theta^\top x$ is positive for inputs on the side θ points toward. We want θ to have a positive dot product with the positive examples and a negative one with the negative examples — to be *similar* to the first and dissimilar to the second. When a positive x lands on the wrong side, the most naive way to make θ more like x is to add a bit of x to it. That rotates θ toward x, and the boundary rotates with it (Figure 4.1).

The lecture calls α "some small learning rate", but for the perceptron as usually run it has no effect at all. Starting from $\theta = 0$, every update adds $\pm\alpha x^{(i)}$, so θ is always α times a sum of examples, and the sign of $\theta^\top x$ — the only thing the prediction depends on — does not depend on α. The generator checks this: on a separable set of 40 points, $\alpha = 1$ and $\alpha = 0.001$ make the same 8 mistakes in the same order and finish with parameter vectors that differ by exactly the

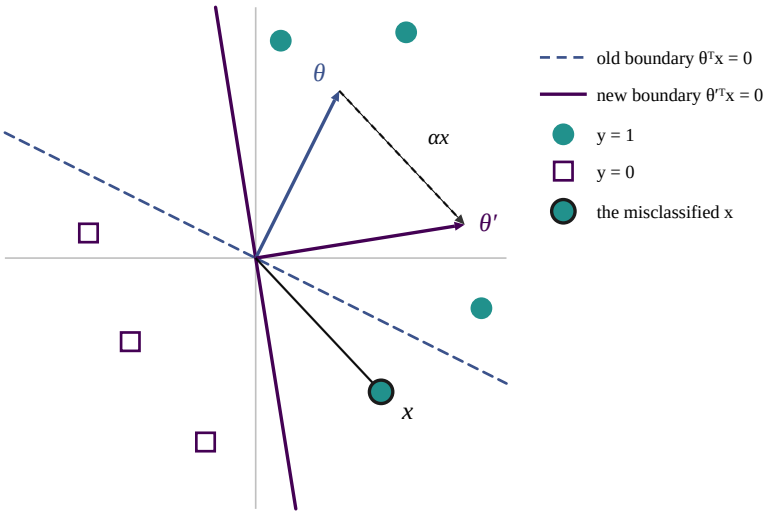


Figure 4.1.: One perceptron step. The positive example x lies on the wrong side of the dashed boundary $\theta^\top x = 0$; adding αx (here $\alpha = 1$, for visibility) gives θ', whose boundary puts x on the positive side while every other point keeps its side. *Computed by* `figures/04-perceptron-glm/make_diagrams.py`.

factor of 1000. The learning rate starts to matter only if it changes over time, which is where a student question led.

12:39 Asked why the perceptron is not used in practice, Avati gives two reasons. It has no probabilistic interpretation: it draws a boundary but says nothing about how confident it is. And it has a famous limitation. The perceptron was celebrated in the late 1950s as a model of the brain (Rosenblatt, 1958); the 1969 book *Perceptrons* by Marvin Minsky and Seymour Papert (the lecture remembers it as a paper by Minsky) set out what a single linear threshold unit cannot compute, and the smallest example is XOR — two positive and two negative points at the corners of a square, diagonally opposite, which no line separates. Interest in the model collapsed afterwards.

13:39 A second question asked when to stop training. If the data are linearly separable, the answer is a theorem the lecture does not state: the perceptron stops making mistakes after finitely many updates. If every $\|x^{(i)}\| \le R$ and some unit vector u separates the classes with margin γ, the number of mistakes is at most $(R/\gamma)^2$, whatever the order of the examples (Novikoff, 1962). On the generator's 40 points the bound is 55 and the perceptron makes 8. If the data are not separable, it never settles — on XOR it is still making mistakes on all four points in the thousandth pass — so Avati's practical advice is to anneal: shrink α with every example until θ stops moving much. That is exactly the case where α is no longer irrelevant, since later updates then count for less than earlier ones.

13:31 Avati also offers the cleanest way to place the perceptron beside the previous lecture: logistic regression is a soft version of it. The step function is the limit of $g(kz) = 1/(1 + e^{-kz})$ as the slope k grows (Figure 4.2); at $k = 12$ the two already differ by less than 0.003 everywhere outside $|z| < 0.5$. What the sigmoid buys in exchange for that softness is a likelihood, and with it everything that follows.

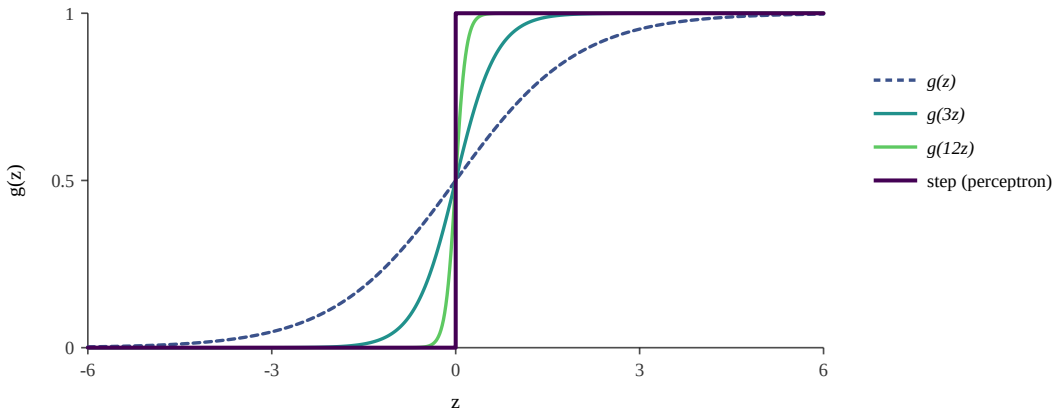


Figure 4.2.: The perceptron's threshold as a hard sigmoid: $g(kz)$ for $k = 1, 3, 12$ steepens toward the step function. *Computed by* `figures/04-perceptron-glm/make_diagrams.py`.

> **i** In the official notes
>
> The perceptron is the short digression of cs229-notes1, section 6, which presents it only as an algorithm; the mistake bound above is not in the 2018 notes.

4.2. The exponential family

14:53 The exponential family is a class of distributions with convenient mathematics, and it is the bridge to generalized linear models. A distribution belongs to it if its density (or, for discrete y, its probability mass function) can be written

$$p(y;\eta) = b(y)\exp\!\left(\eta^{\top}T(y) - a(\eta)\right). \tag{4.2}$$

Avati names each piece. The variable is called y, not x, on purpose: these distributions will model the *output* of a supervised problem, and x enters only when the model is built. η is the *natural parameter*. $T(y)$ is the *sufficient statistic*, in the statistician's sense, but for every distribution in this course $T(y) = y$, and the lecture tells the reader to make that substitution and move on. $b(y)$ is the *base measure*, which may depend only on y; and $a(\eta)$ is the *log-partition function*, which may depend only on η. In most of the course y and η are scalars; softmax will need a vector η.

18:48 The name of a explains its job. Moving it out of the exponent gives $p(y;\eta) = b(y)\,e^{\eta T(y)}/e^{a(\eta)}$, so $e^{a(\eta)}$ is the normalising constant — the *partition function* — that makes the density integrate to one, and $a(\eta)$ is its log. That has a consequence worth making explicit, because the lecture says one can choose a, b and T freely as long as the result integrates to one. The condition is not a side constraint on a; it determines it. Once b and T are fixed,

$$a(\eta) = \log \int b(y)\,e^{\eta T(y)}\,dy,$$

with a sum in place of the integral for discrete y. Choosing b and T chooses the family; a comes with it. The generator confirms, for the three families below, that the $a(\eta)$ read off by pattern matching equals this normaliser computed by brute-force summation.

21:44 To show that a familiar distribution is a member, write its density in the usual form and massage it until it matches Equation 4.2. For the Bernoulli with mean ϕ, the density $\phi^y(1-\phi)^{1-y}$ — an "if–else written in maths", as Avati puts it, since it equals ϕ when $y=1$ and $1-\phi$ when $y=0$ — becomes, after wrapping it in exp log,

$$p(y;\phi) = \exp\!\left(y \log \frac{\phi}{1-\phi} + \log(1-\phi)\right).$$

Matching term by term gives $b(y)=1$, $T(y)=y$, the natural parameter $\eta = \log\frac{\phi}{1-\phi}$, and $a(\eta) = -\log(1-\phi)$. 26:14 Inverting the expression for η gives

$$\phi = \frac{1}{1+e^{-\eta}}, \tag{4.3}$$

the sigmoid, which arrives here without being asked for. Substituting it into a gives $a(\eta) = \log(1+e^{\eta})$, now a function of η alone as the definition requires.

28:41 The Gaussian works the same way once its variance is fixed. With $\sigma^2 = 1$ — and lecture 3 showed that the variance has no effect on the fitted θ in least squares, so nothing is lost —

$$\frac{1}{\sqrt{2\pi}} \exp\!\left(-\tfrac{1}{2}(y-\mu)^2\right) = \underbrace{\frac{1}{\sqrt{2\pi}} e^{-y^2/2}}_{b(y)} \exp\!\Big(\mu\, y - \underbrace{\tfrac{1}{2}\mu^2}_{a(\eta)}\Big),$$

so $\eta = \mu$ and $a(\eta) = \eta^2/2$. A student asked what happens if the variance is unknown: the Gaussian is still in the family, but η becomes a two-dimensional vector and $T(y) = (y, y^2)$. The Poisson, which the lecture uses later without deriving, fits the same mould: $\lambda^y e^{-\lambda}/y!$ has $b(y) = 1/y!$, $\eta = \log\lambda$ and $a(\eta) = e^{\eta}$.

> **i** In the official notes
>
> The Bernoulli and Gaussian derivations are cs229-notes1, section 8, which also lists the multinomial, Poisson, gamma, exponential, beta and Dirichlet as members.

4.3. Why the exponential family is convenient

33:10 Three properties make the family worth the trouble, and Avati states them without proof — they are an exercise in the first problem set. First, maximum likelihood in the natural parameter is a concave problem, so it has no spurious local maxima; the negative log-likelihood, its cost-function form, is convex. Second and third, the log-partition function produces the moments by differentiation:

$$\mathbb{E}[y;\eta] = \frac{\partial a}{\partial \eta}, \qquad \mathrm{Var}[y;\eta] = \frac{\partial^2 a}{\partial \eta^2}. \tag{4.4}$$

35:08 The point Avati makes about these is practical: computing a mean or a variance normally means integrating against the density, and here it means differentiating a single function, which is far easier. For the Bernoulli, $a'(\eta) = 1/(1+e^{-\eta}) = \phi$ and $a''(\eta) = \phi(1-\phi)$; for the Gaussian, $a' = \eta = \mu$ and $a'' = 1$; for the Poisson, both derivatives are $e^{\eta} = \lambda$, the textbook fact that a Poisson's mean equals its variance. The generator checks all three by computing a' and a'' by

finite differences and the mean and variance by summing over the support. The first property follows from the third: a variance is never negative, so a is convex, and the log-likelihood $\log b(y) + \eta y - a(\eta)$ is linear minus convex, which is concave. 36:06 When η is a vector, as for a Gaussian with unknown variance, a student asked how this generalises; the gradient of a gives the mean of $T(y)$ and its Hessian the covariance.

4.4. Generalized linear models

36:35 A generalized linear model attaches the inputs to an exponential-family distribution, and the lecture frames it as three design choices.

1. $y \mid x; \theta$ is distributed according to some member of the exponential family, with natural parameter η.
2. The natural parameter is linear in the input: $\eta = \theta^\top x$, with $\theta, x \in \mathbb{R}^n$.
3. At test time, the prediction for x is the mean of that distribution: $h_\theta(x) = \mathbb{E}[y \mid x; \theta]$.

38:36 The first choice is the modelling decision, and it is dictated by what kind of quantity y is. Real-valued outputs such as house prices call for a Gaussian; binary labels for a Bernoulli; counts — non-negative integers, such as the number of visitors to a website — for a Poisson; positive reals such as a volume or the time until an event for a gamma or an exponential. Avati warns that the *exponential distribution* and the *exponential family* are different things, though the first is a member of the second, and the list ends with distributions over probabilities, the beta and the Dirichlet, which mostly appear in Bayesian statistics. Whether a GLM is a classifier or a regressor is therefore not a property of GLMs at all: it is decided by the distribution put in the first slot.

42:55 The picture to keep in mind has two halves. On the left is a linear model: x goes in, a learnable θ multiplies it, and out comes a number. On the right is a distribution from the exponential family, and that number becomes its natural parameter. 45:40 Only θ is learned. A student asked whether training adjusts the distribution's parameters, and the answer separates two uses of the word "parameter": η is not a variable that gradient descent touches, it is the output of the linear model for one particular x. Training is maximum likelihood over θ,

$$\max_\theta \sum_{i=1}^{m} \log p\big(y^{(i)}; \eta = \theta^\top x^{(i)}\big)\,,$$

which is concave in θ because it is concave in η and η is linear in θ.

48:46 Here the recipe pays off. One would expect each new distribution to need its own algebra: write p as a function of $\theta^\top x$, differentiate, simplify. It does not, because for every GLM built this way the stochastic gradient-ascent update is

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}, \tag{4.5}$$

with $h_\theta(x) = a'(\theta^\top x)$ the only thing that changes. The derivation is one line: the log-likelihood of one example is $\log b(y) + \theta^\top x\, y - a(\theta^\top x)$, and its gradient is $(y - a'(\theta^\top x))\, x$, which by Equation 4.4 is $(y - h_\theta(x))\, x$. This is why linear regression, logistic regression and Poisson regression share an update: the first is the Gaussian GLM with $h = \theta^\top x$, the second the Bernoulli with $h = 1/(1+e^{-\theta^\top x})$, the third the Poisson with $h = e^{\theta^\top x}$. The generator checks the expression against a numerical gradient of the log-likelihood for all three.

"Every GLM" carries a qualification the lecture leaves implicit: the identity holds because the natural parameter itself is set to $\theta^\top x$. A model that instead connects $\theta^\top x$ to the mean through some other function does not inherit it. The probit model of lecture 3, $P(y = 1 \mid x) = \Phi(\theta^\top x)$ with Φ the standard normal CDF, is a Bernoulli model with exactly the same shape, and the update that looks right for it — Equation 4.5 with $h = \Phi(\theta^\top x)$ — is not its gradient: at a test point on synthetic data it has the right direction but a length off by 41 percent. That would be harmless if it only rescaled the step, but it also moves the place where the step vanishes. Iterated to convergence, the look-alike update stops at $\theta = (-0.200, 0.697)$, while the probit maximum-likelihood estimate is $(-0.210, 0.707)$. The two links are close, so the error is small, but the algorithm is now solving a different equation from the one it claims to.

50:43 For batch training, sum the update over the examples. Asked about Newton's method, Avati calls it probably the most common way to fit a GLM, provided the number of features is below a few thousand, so that the Hessian is affordable — the same rule of thumb Ng gave in lecture 3. Concavity is what makes it safe: on a Poisson regression with 200 synthetic counts, Newton's method reaches the maximum in 7 iterations, at the same θ that plain gradient ascent eventually finds.

4.5. Three parametrisations

52:03 The lecture pauses on terminology, because a GLM has three sets of numbers called parameters and they are easy to confuse (Figure 4.3). The *model parameters* θ are what learning adjusts. The *natural parameter* η is the linear model's output, one per example. The *canonical parameters* are the ones the distribution is usually written in: ϕ for a Bernoulli, μ (and σ^2) for a Gaussian, λ for a Poisson. The step from θ to η is linear by design choice 2. The step from η to the mean has a name: $g(\eta) = \mathbb{E}[y; \eta] = a'(\eta)$ is the *canonical response function*, and its inverse g^{-1} is the *canonical link function*.

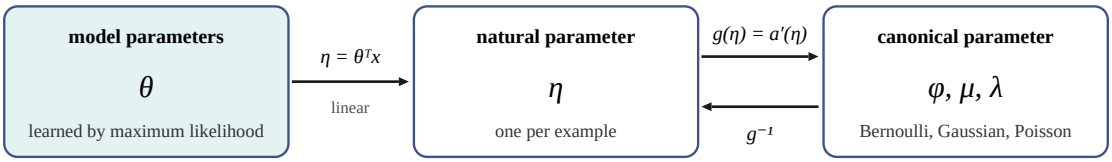


Figure 4.3.: The three parametrisations of a GLM. Only θ is learned; $\eta = \theta^\top x$ is the natural parameter of the distribution for one input; the response function $g = a'$ maps it to the mean, and the canonical link g^{-1} maps back.

56:23 With the vocabulary in place, logistic regression stops being a choice and becomes a consequence. For a binary label the natural distribution is the Bernoulli; design choice 3 says the hypothesis is its mean, $h_\theta(x) = \phi$; and Equation 4.3 says $\phi = 1/(1 + e^{-\eta})$ with $\eta = \theta^\top x$. Lecture 3 pulled the logistic function "out of thin air" because it squashes the real line into $(0, 1)$. Here, as Avati puts it, it simply pops out: the sigmoid is the canonical response function of the Bernoulli. The same steps applied to the Gaussian return $h_\theta(x) = \theta^\top x$, ordinary least squares.

The idea and the name are due to Nelder and Wedderburn (1972), who showed that one fitting procedure covered regression models that had been treated separately.

> **In the official notes**
>
> cs229-notes1, section 9, constructs the GLM from the same three assumptions and derives ordinary least squares (9.1) and logistic regression (9.2) from them.

4.6. What the assumptions say about the data

1:00:03 Avati's last pass over GLMs draws the assumptions as a story about how the data were generated. For the Gaussian, take a line $\theta^\top x$; at every x, stand a unit-variance Gaussian on the line; and draw each training y from the Gaussian above its x (Figure 4.4, left). Learning runs the story backwards: each choice of θ is a different line, and maximum likelihood picks the line from whose Gaussians the observed points were most likely to have come.

1:04:50 For classification the story has one more step. The line now gives η, which can be any real number; passing it through the sigmoid gives ϕ, the probability that $y = 1$ at that x; and each label is a coin flip with that bias, so labels are mostly 0 where the curve is low and mostly 1 where it is high (Figure 4.4, right). Logistic regression asks which θ would have produced a sigmoid from which these labels were most likely drawn.

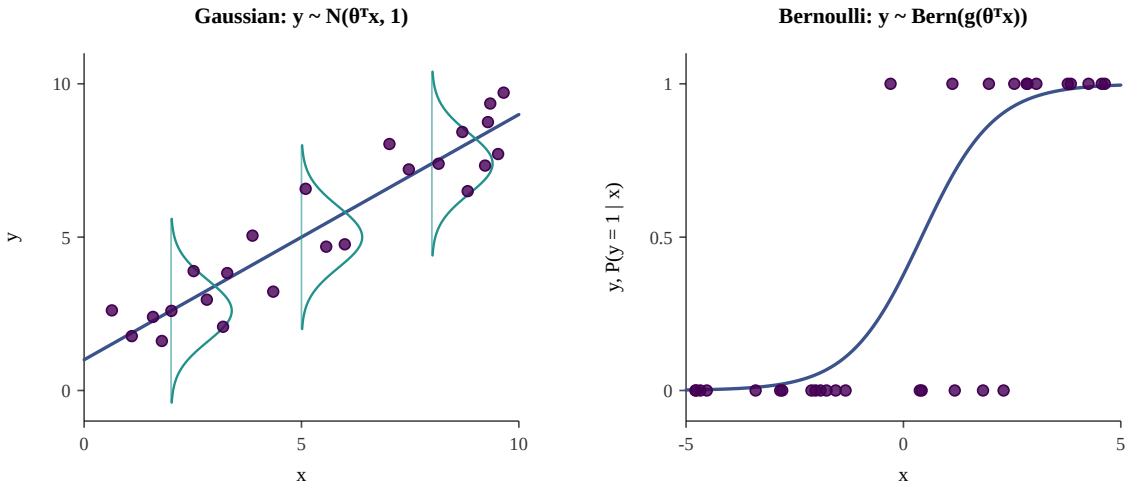


Figure 4.4.: The data-generating process each GLM assumes, sampled from the model. Left: Gaussian y around the line $\theta^\top x$, with the density drawn sideways at three inputs. Right: Bernoulli labels whose probability of being 1 follows $g(\theta^\top x)$. *Computed by* `figures/04-perceptron-glm/make_diagrams.py`.

4.7. Softmax regression

1:07:58 Softmax regression extends logistic regression to k classes. The notes present it as one more GLM, built on the multinomial distribution; the lecture deliberately takes another route and presents it as *cross-entropy minimisation*, an interpretation worth having because it is how classifiers with thousands of classes are usually described.

1:11:04 The labels are *one-hot* vectors: a vector of k zeros with a single 1 in the position of the correct class. Where logistic regression had one parameter vector, softmax has one per class, $\theta_1, \ldots, \theta_k$, which together form an $n \times k$ matrix. For an input x, each class gets a score

$\theta_c^\top x$, called a *logit*, which can be any real number. 1:16:24 To turn the scores into a probability distribution over classes, exponentiate them, which makes them all positive, and divide by their sum, which makes them add to one:

$$\hat{p}(y = c \mid x) = \frac{\exp(\theta_c^\top x)}{\sum_{j=1}^{k} \exp(\theta_j^\top x)}. \tag{4.6}$$

1:19:09 The label is read as a distribution too, $p(y)$, which puts probability 1 on the correct class. Learning makes $\hat{p}$ look like p by minimising the *cross entropy* between them,

$$\mathrm{CE}(p, \hat{p}) = -\sum_{c=1}^{k} p(c) \log \hat{p}(c) = -\log \hat{p}(y \mid x), \tag{4.7}$$

where the sum collapses because p is zero everywhere except at the true class y. (The minus sign is missing from the board the first time; it belongs there, since the loss has to be non-negative.) Figure 4.5 follows one example through the pipeline: an input whose true class is triangle but which scores highest for square receives $\hat{p}(\text{triangle}) = 0.19$ and a loss of 1.67. The last expression in Equation 4.7 is the negative log-likelihood of a multinomial model, so the lecture's route and the notes' arrive at the same objective. Its gradient for class c is $(\hat{p}(c \mid x) - \mathbb{1}\{y = c\})\, x$, the same "prediction minus truth, times input" shape as Equation 4.5; and with $k = 2$ the softmax probability of class 1 is exactly $1/(1 + e^{-(\theta_1 - \theta_2)^\top x})$, logistic regression with $\theta = \theta_1 - \theta_2$.

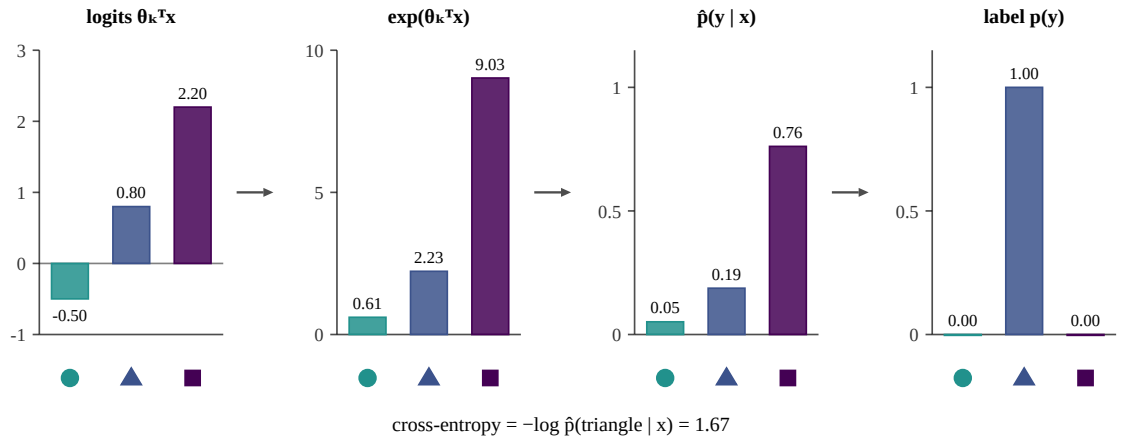


Figure 4.5.: Softmax regression for one example. The logits are exponentiated, then normalised into a distribution $\hat{p}$ over the three classes; the label is a one-hot distribution on triangle, and the cross-entropy loss is $-\log \hat{p}(\text{triangle} \mid x)$. *Computed by* `figures/04-perceptron-glm/make_diagrams.py`.

1:12:53 That two-class reduction exposes something the board picture hides. The lecture draws one line per class, $\theta_c^\top x = 0$, with each class's own region on the positive side, as though each θ_c carves out its class independently. But Equation 4.6 is unchanged if the same vector c is added to every θ_j: the factor $e^{c^\top x}$ cancels between numerator and denominator. Only the differences $\theta_i - \theta_j$ matter, so the decision boundary between two classes is $(\theta_i - \theta_j)^\top x = 0$, and the individual lines $\theta_c^\top x = 0$ can be moved anywhere without changing a single prediction (Figure 4.6). On the fitted model in the figure, the board's picture — exactly one positive score, and it belongs to the predicted class — holds on only 45 percent of the plotted square even before any shift. This redundancy is why the official notes give softmax only $k - 1$ parameter vectors and fix $\theta_k = 0$: the k-th vector adds nothing a shift could not remove.

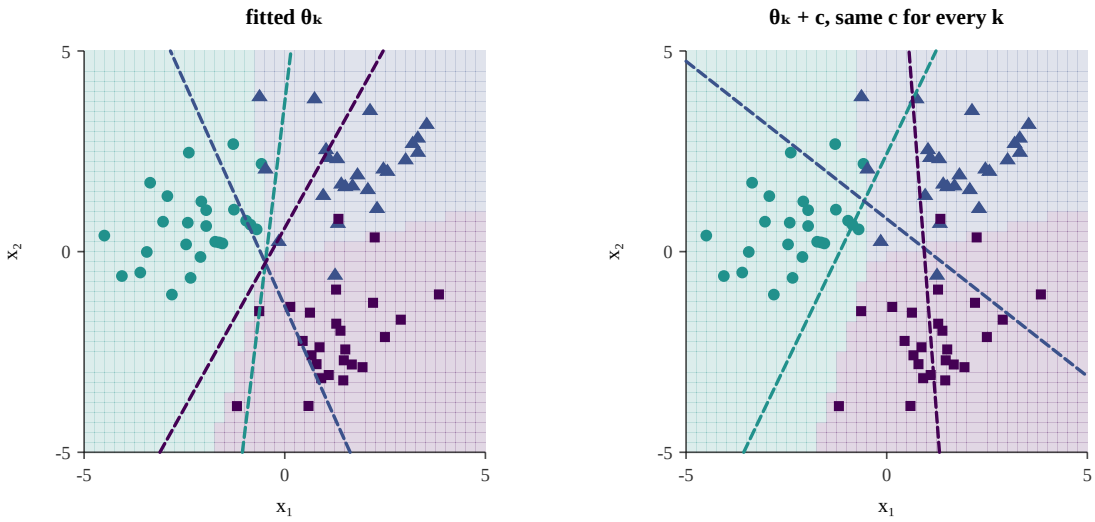


Figure 4.6.: The same softmax model with every θ_c shifted by one common vector. The shaded decision regions are identical; the dashed lines $\theta_c^\top x = 0$ move to entirely different places, because they carry no meaning on their own. *Fitted by gradient descent in* `figures/04-perceptron-glm/make_diagrams.py`.

> **In the official notes**
>
> cs229-notes1, section 9.3, derives softmax regression as the GLM for the multinomial distribution, with $T(y)$ a $(k-1)$-dimensional indicator vector, and arrives at the same hypothesis and log-likelihood.

5. GDA and Naive Bayes

Lecture 5

Lecture video: Lecture 5: GDA and Naive Bayes · 1:18:52

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Every algorithm so far has been *discriminative*: it models $p(y \mid x)$ directly, or maps x to a label, and learns by searching for a boundary between the classes. This lecture introduces the other family. A *generative* learning algorithm models what each class looks like, $p(x \mid y)$, together with how common each class is, $p(y)$, and classifies a new example by asking which class model it fits better. Gaussian discriminant analysis (GDA) does this with a Gaussian per class and turns out to imply exactly the logistic form of lecture 3 — but with stronger assumptions, which help when data are scarce and hurt when they are wrong. Naive Bayes does it for text, with one binary feature per word and an assumption Ng calls flatly untrue, and still makes a serviceable spam filter.

5.1. Two ways to build a classifier

1:09 Ng starts from the tumour data again, two features and two classes. Logistic regression initialises a line and lets gradient descent move it until it separates the malignant from the benign examples; it is searching for a boundary. A generative algorithm never looks for a boundary. It takes the malignant tumours alone and fits a model of where they live — roughly inside one ellipse — then does the same for the benign tumours. When a new patient arrives, it compares their features against the two models and picks the class whose model the patient resembles more.

3:56 In symbols, a discriminative algorithm learns $p(y \mid x)$, or a function from x to labels. A generative one learns $p(x \mid y)$ — what the features are like given the class — and the *class prior* $p(y)$, the odds that a tumour is malignant before the patient has even been examined. 5:50 Bayes' rule converts those two into the quantity a classifier needs:

$$p(y = 1 \mid x) = \frac{p(x \mid y = 1)\, p(y = 1)}{p(x)}, \qquad p(x) = p(x \mid y = 1)\, p(y = 1) + p(x \mid y = 0)\, p(y = 0). \tag{5.1}$$

35:49 If all that is wanted is the most likely label, the denominator can be skipped. $p(x)$ does not depend on y, so

$$\arg\max_y p(y \mid x) = \arg\max_y p(x \mid y)\, p(y),$$

and implementations often never compute $p(x)$ at all. Ng's aside on notation is worth keeping: $\min_z (z-5)^2$ is 0, the smallest value attained, while $\arg\min_z (z-5)^2$ is 5, the input that attains it.

5.2. The multivariate Gaussian

9:46 GDA needs a Gaussian over vectors. A random vector $z \in \mathbb{R}^n$ is distributed $\mathcal{N}(\mu, \Sigma)$, with mean vector $\mu \in \mathbb{R}^n$ and covariance matrix $\Sigma \in \mathbb{R}^{n \times n}$, symmetric and positive definite, if its density is

$$p(z; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp\!\left(-\tfrac{1}{2}(z-\mu)^\top \Sigma^{-1} (z-\mu)\right), \tag{5.2}$$

where $\Sigma = \mathbb{E}[(z-\mu)(z-\mu)^\top]$. Ng's view is that almost nobody memorises this formula until they have used it enough times, and that pictures teach it better. 12:00 Ng projected a sequence of three-dimensional plots; Figure 5.1 recomputes them as contour maps, every panel at the same four density heights so that panels can be compared.

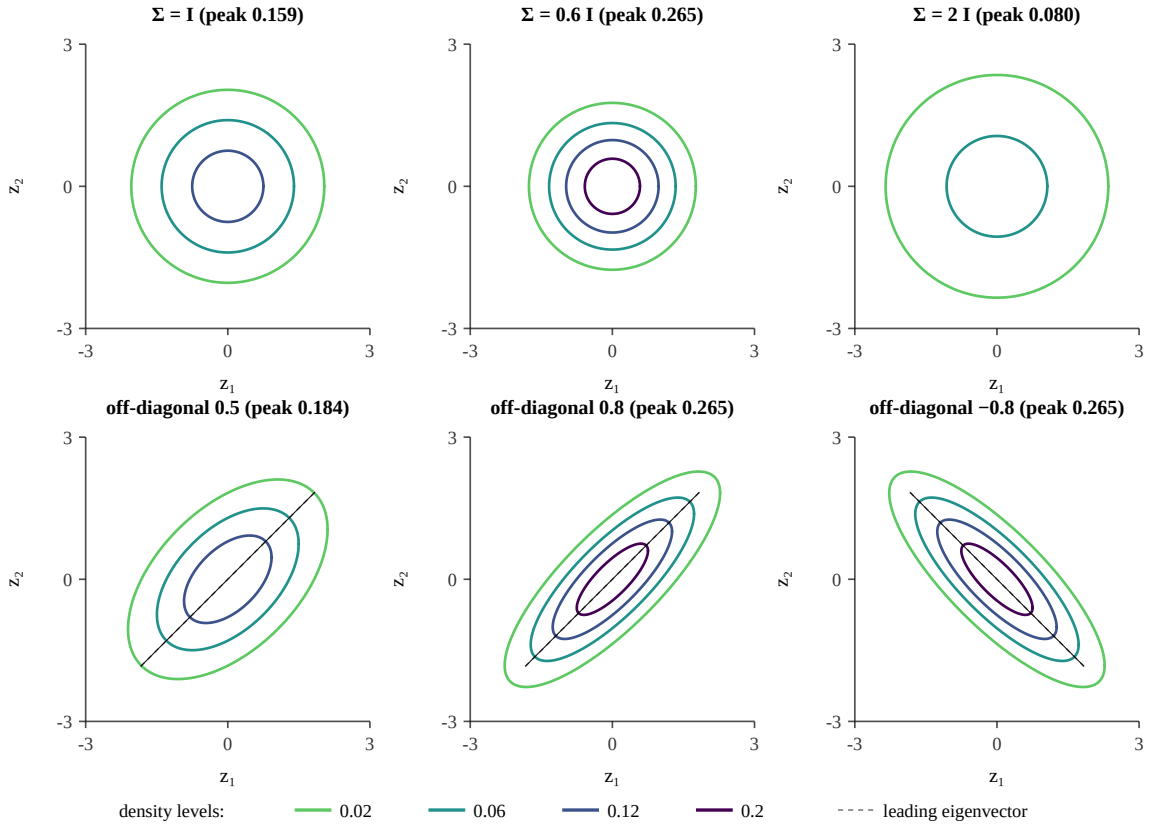


Figure 5.1.: Contours of two-dimensional Gaussians with mean zero, each panel drawn at the same four density heights. Shrinking Σ makes the bump taller and narrower — at $0.6I$ the highest level appears; enlarging it lowers the peak until only the lowest level remains. Positive off-diagonal entries tilt the contours along $z_1 = z_2$, negative ones the other way; the dashed line is the leading eigenvector of Σ. *Computed by* `figures/05-gda-naive-bayes/make_diagrams.py`.

With $\mu = 0$ and $\Sigma = I$ — the *standard* Gaussian — the contours are circles. Multiplying Σ by 0.6 shrinks the spread, and because the density must still integrate to one, it grows taller: in two dimensions the peak $1/(2\pi|\Sigma|^{1/2})$ scales by $1/0.6$, from 0.159 to 0.265. Doubling Σ halves it. Raising the off-diagonal entries makes z_1 and z_2 positively correlated, and the circles stretch into ellipses along the diagonal, narrower as the entry rises to 0.8; negative entries tilt them the other way. One coincidence in the figure is a useful check on the formula: the off-diagonal-0.8 Gaussian has the same peak as $0.6I$, because both have $|\Sigma| = 0.36$. Changing μ moves the bump without changing its shape.

16:26 A student asked whether the columns of Σ point in meaningful directions. Ng's answer is that single columns are not worth reading in isolation — the matrix is symmetric — but its eigenvectors are: they are the principal axes of the elliptical contours, the long axis belonging to the largest eigenvalue. For the off-diagonal-0.8 panel the eigenvalues are 1.8 and 0.2, with the long axis along $(1,1)/\sqrt{2}$. This is the idea principal component analysis builds on later in the course.

> **In the official notes**
>
> cs229-notes2, section 1.1, defines the multivariate normal and shows the same family of surface plots.

5.3. The GDA model and its fit

19:02 Gaussian discriminant analysis assumes that each class's features are Gaussian, with a mean of their own and a covariance they share, and that the label is a coin flip:

$$\begin{aligned} y &\sim \text{Bernoulli}(\phi), \\ x \mid y = 0 &\sim \mathcal{N}(\mu_0, \Sigma), \\ x \mid y = 1 &\sim \mathcal{N}(\mu_1, \Sigma). \end{aligned} \tag{5.3}$$

For generative models Ng drops the convention $x_0 = 1$, so $x \in \mathbb{R}^n$. The parameters are $\phi \in (0,1)$, $\mu_0, \mu_1 \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$. 20:24 A shared Σ is a choice, not a necessity — separate Σ_0 and Σ_1 are perfectly reasonable — and the reason for it becomes clear when the decision boundary is drawn.

23:05 The fit is where the two families part. Logistic regression chose θ to maximise the *conditional* likelihood $\prod_i p(y^{(i)} \mid x^{(i)}; \theta)$. GDA maximises the *joint* likelihood,

$$L(\phi, \mu_0, \mu_1, \Sigma) = \prod_{i=1}^{m} p\big(x^{(i)}, y^{(i)}\big) = \prod_{i=1}^{m} p\big(x^{(i)} \mid y^{(i)}\big)\, p\big(y^{(i)}\big)\,.$$

26:56 Taking logs, differentiating and setting the derivatives to zero gives closed forms — the derivation is a problem-set exercise, and Ng gives the answers along with the intuition for each:

$$\begin{aligned} \phi &= \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\}, & \mu_k &= \frac{\sum_i 1\{y^{(i)} = k\}\, x^{(i)}}{\sum_i 1\{y^{(i)} = k\}}, \\ \Sigma &= \frac{1}{m}\sum_{i=1}^{m} \big(x^{(i)} - \mu_{y^{(i)}}\big)\big(x^{(i)} - \mu_{y^{(i)}}\big)^{\top}. \end{aligned} \tag{5.4}$$

The *indicator* $1\{\cdot\}$ is 1 when its argument is true and 0 when it is false. The estimate of ϕ is the fraction of malignant tumours in the training set, as the maximum-likelihood bias of a coin is the fraction of heads. Each mean is the average of its class's feature vectors: the indicator in the numerator keeps only that class's examples, the one in the denominator counts them. And Σ is the average outer product of each example's deviation from *its own* class mean — one covariance fitted to both clouds at once. Ng is careful to say that this intuition is not the derivation; the derivation is calculus on $\log L$. The generator takes the shortcut of checking the result instead: on 80 synthetic points, 500 random perturbations of the closed-form estimates all lower the joint log-likelihood.

There is no iteration anywhere in Equation 5.4: fitting GDA is computing two means and a covariance. That will matter when Ng explains why GDA is still worth using.

> **In the official notes**
>
> The model and its maximum-likelihood estimates are cs229-notes2, section 1.2.

5.4. GDA's boundary is logistic

37:07 On the same two-feature data, Ng shows logistic regression's line migrating over about twenty iterations of gradient descent, then GDA fitting one Gaussian to each class and classifying by Bayes' rule. Both produce straight boundaries, and the two lines are close but not the same (Figure 5.2, left).

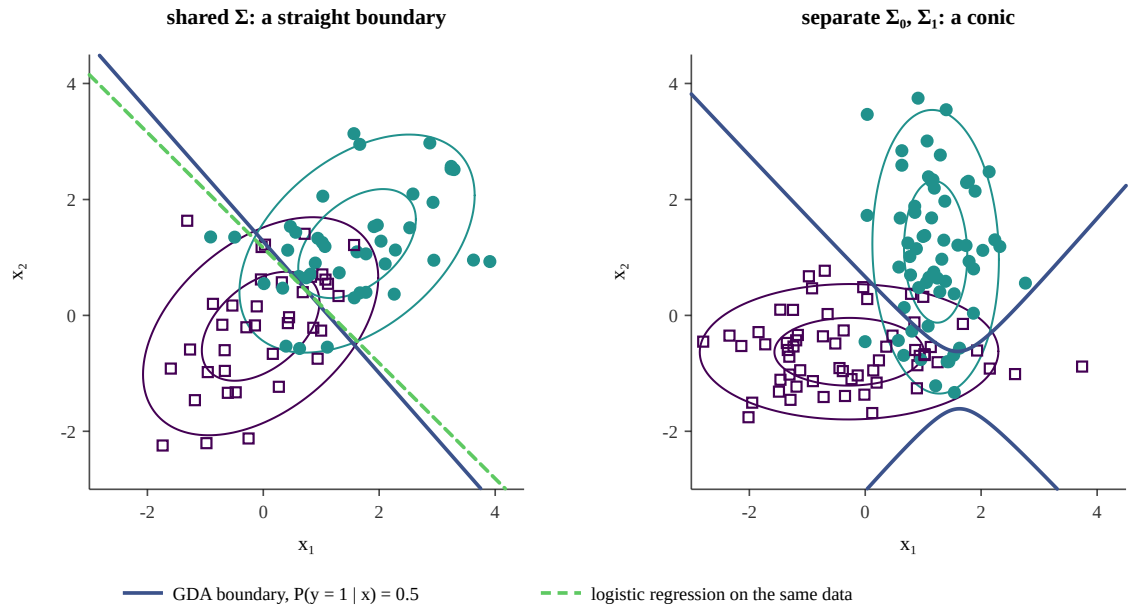


Figure 5.2.: Left: GDA with a shared covariance on synthetic data, with the fitted Gaussians' contours at one and two standard deviations, the GDA boundary, and logistic regression's boundary on the same 80 points. Right: classes with very different spreads, each given its own covariance; the boundary becomes a conic, here a hyperbola. *Computed by* `figures/05-gda-naive-bayes/make_diagrams.py`.

39:42 A student asked why the model uses two means but one covariance. With a shared Σ the boundary is linear, which is what many problems want. With Σ_0 and Σ_1 the model also works,

but the boundary is no longer a line, and Ng says the parameter count roughly doubles. That holds for the covariance part, and for the whole model only when n is large: counting ϕ and the means too, a shared covariance needs $1 + 2n + n(n+1)/2$ parameters and two covariances $1 + 2n + n(n+1)$, a ratio of 1.38 at $n = 2$, 1.72 at $n = 10$ and 1.96 at $n = 100$. 58:13 Asked later what the posterior looks like with separate covariances, Ng's recollection was that it is still a logistic function, but with quadratic terms inside it. That is right: the log-odds becomes a quadratic in x, and its zero set is a conic (Figure 5.2, right).

41:02 The connection to logistic regression is the lecture's most interesting property of GDA. Fix the parameters and, for a single feature, evaluate $p(y = 1 \mid x)$ by Bayes' rule at point after point along the axis. 43:31 Far to the left, almost any point must have come from the class-0 Gaussian, so the posterior is near 0; far to the right it is near 1; at the midpoint between the two means, Ng says, you cannot tell, so it is 0.5. Joining the dots gives an S-shaped curve, and 47:13 it is exactly a sigmoid (Figure 5.3). With a shared covariance the quadratic terms of the two Gaussians' exponents cancel in the log-odds, leaving

$$p(y = 1 \mid x) = \frac{1}{1 + e^{-(\theta^\top x + \theta_0)}}, \qquad \theta = \Sigma^{-1}(\mu_1 - \mu_0), \quad \theta_0 = -\tfrac{1}{2}\left(\mu_1^\top \Sigma^{-1} \mu_1 - \mu_0^\top \Sigma^{-1} \mu_0\right) + \log \frac{\phi}{1 - \phi}. \tag{5.5}$$

The generator confirms the identity at 200 random points against Bayes' rule computed directly.

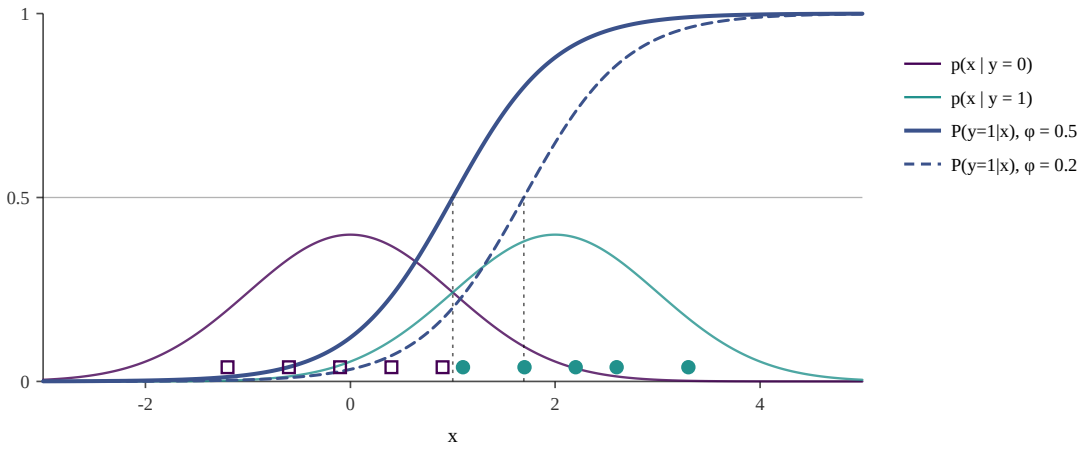


Figure 5.3.: The posterior along one feature. Two unit-variance class densities (thin), five training points from each class, and $p(y = 1 \mid x)$ computed by Bayes' rule (thick): a sigmoid crossing 0.5 midway between the means when $\phi = 0.5$, and at $x = 1.69$ when $\phi = 0.2$. *Computed by* `figures/05-gda-naive-bayes/make_diagrams.py`.

The midpoint claim holds only when the classes are equally likely. The last term of θ_0 is the prior log-odds, and it shifts the curve. For one feature the crossing point is

$$x_{1/2} = \frac{\mu_0 + \mu_1}{2} + \frac{\sigma^2}{\mu_1 - \mu_0} \log \frac{1 - \phi}{\phi},$$

so with means 0 and 2, unit variance and a prior of 0.2 on class 1, the boundary moves from $x = 1$ to $x = 1.69$: a rarer class needs stronger evidence. On a malignant/benign problem, where ϕ is usually far from one half, this term is anything but negligible.

47:53 So both algorithms end up with a sigmoid in x, but they do not end up with the same sigmoid. GDA's θ is a byproduct of fitting the joint likelihood; it is not the θ that maximises the conditional likelihood, which is what logistic regression computes. On the 80 points of Figure 5.2, logistic regression reaches a conditional log-likelihood of -29.73, and the θ GDA implies scores -29.97 on the same objective. Neither is wrong; they are answering different questions.

5.5. Which one to use

48:57 Ng frames the comparison as an implication that runs only one way. If Equation 5.3 holds, then $p(y = 1 \mid x)$ is logistic — that was the previous section. But assuming that $p(y = 1 \mid x)$ is logistic says nothing about $x \mid y$ being Gaussian. 53:08 Ng's counterexample: if $x \mid y = 0$ and $x \mid y = 1$ are Poisson with means λ_0 and λ_1, the posterior is again logistic, since the log-odds is $x \log(\lambda_1/\lambda_0) - (\lambda_1 - \lambda_0) + \log \frac{\phi}{1-\phi}$, linear in x. The same holds, Ng adds, whenever the two class-conditionals are the same exponential family differing only in the natural parameter, with $T(x) = x$. GDA therefore makes the *stronger* assumptions and logistic regression the *weaker*.

54:46 The principle that follows is one Ng says will recur throughout the course. Stronger modelling assumptions, when they are roughly right, let an algorithm learn from less data, because the assumptions are information it no longer has to extract from examples. When they are wrong, they cost accuracy that more data cannot recover. Weaker assumptions are more robust: if you do not know whether the data are Gaussian or Poisson or something else from the family, logistic regression is correct in all those cases.

Figure 5.4 puts numbers on this. In a world built to satisfy GDA's assumptions, GDA's average test error is lower than logistic regression's from 10 to about 100 training examples, by up to half a percentage point, and both converge to the Bayes error of 0.201 — though at 6 examples logistic regression, with a slight ridge penalty to keep it finite on separable samples, is ahead. With two features the advantage is real but small; Ng and Jordan (2002) analysed the same trade-off for naive Bayes against logistic regression and found it grows with the number of features: the generative model can approach its (higher) asymptotic error with a number of examples logarithmic in n, the discriminative one with roughly linear. In the Poisson world GDA is misspecified, and still it is slightly ahead at 6 and 10 examples — the assumptions are wrong but informative. The curves cross near 25 examples, and from there logistic regression pulls ahead, reaching 0.160 at 400 examples against GDA's 0.168. With 20,000 examples logistic regression sits at the Bayes error of 0.158 while GDA stalls at 0.162, an error no amount of data removes.

56:13 Ng's own practice follows from this. With large datasets, which is most problems now, Ng would usually reach for logistic regression, because data can make up for what the algorithm was not told. There are two sources of knowledge, in Ng's account: the assumptions you build in and what is learned from data. With plenty of the second you can afford fewer of the first. 56:56 When Ng does use GDA today it is more for computation than accuracy — two means and a covariance, no iterative optimisation.

58:45 Should one test whether the data are Gaussian before using GDA? In practice, Ng says, if you have enough data to run a statistical test and believe it, you probably have enough to use logistic regression. What people actually do is plot the data, rule out GDA if it is obviously non-Gaussian, and otherwise — with, say, fifty healthcare records — use judgment, perhaps by asking doctors whether the distribution is plausibly Gaussian. 59:47 This leads Ng to a broader view: the field has somewhat overhyped big data. More data almost never hurts, but many applications have a hundred examples rather than a million, and there the designer's skill — including the choice of what to assume — matters far more: on a million-image benchmark

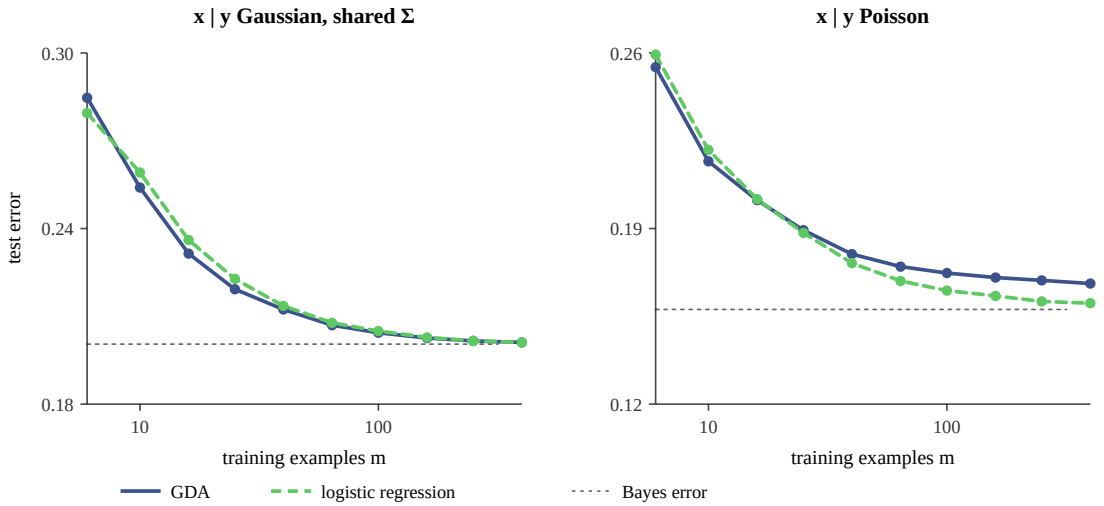


Figure 5.4.: Mean test error over 400 training sets at each size; the vertical axes do not start at zero. Left, data satisfying GDA's assumptions: GDA is slightly ahead from 10 to about 100 examples. Right, Poisson classes, where logistic regression is correctly specified and GDA is not: GDA's error levels off above the Bayes error. *Computed by* `figures/05-gda-naive-bayes/make_diagrams.py`.

many teams do well, while with a hundred examples the skilled teams do much better than the rest. 1:02:42 Asked whether the comparison carries over to more than two classes, Ng says it does, with softmax regression as the discriminative counterpart.

> **i** In the official notes
>
> cs229-notes2, section 1.3, makes the same comparison and states that GDA is asymptotically efficient when its assumptions hold.

5.6. Naive Bayes

1:04:08 GDA needs continuous features. Naive Bayes is a generative algorithm for discrete ones, and Ng builds it around spam filtering, though the same method applies to any text classification — the lecture mentions eBay sorting free-text listings into product categories.

1:05:08 The first question is how to turn an e-mail into a feature vector. Take a list of words — conceptually the dictionary, from "a" and "aardvark" to "zymurgy", in practice the 10,000 most frequent words in your own training set, so that "CS229" makes the list even though no dictionary has it — and set $x_j = 1$ if word j appears in the e-mail and 0 if not. So $x \in \{0,1\}^n$ with $n = 10{,}000$.

1:09:28 Modelling $p(x \mid y)$ directly is hopeless: a distribution over all $2^{10{,}000}$ binary vectors needs $2^{10{,}000} - 1$ parameters for each class, a number with 3,011 digits. 1:10:00 Naive Bayes assumes that the x_j are *conditionally independent* given y. The chain rule of probability, which is always true, writes the joint as $p(x_1 \mid y)\, p(x_2 \mid x_1, y)\, p(x_3 \mid x_1, x_2, y) \cdots$; the assumption drops every conditioning variable except y,

$$p(x_1, \dots, x_n \mid y) = \prod_{j=1}^{n} p(x_j \mid y). \tag{5.6}$$

In words: once you know whether an e-mail is spam, learning that it contains “buy” tells you nothing about whether it contains “aardvark”. Figure 5.5 shows this as the graphical model Ng sketches for those who have met them.

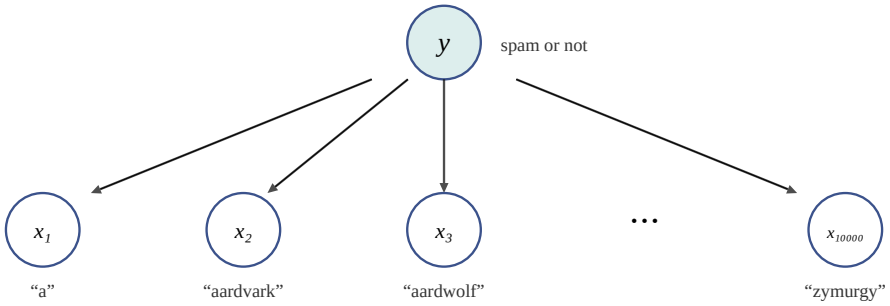


Figure 5.5.: The naive Bayes assumption as a graphical model: the label points to every word indicator, and no word indicator points to another.

1:11:53 Ng does not defend the assumption as true. It is not, mathematically — words in an e-mail are obviously correlated — but, as with data that are not quite Gaussian, it may be not so wrong that you cannot get away with it. Where it does bite is worth seeing. Suppose a word appears in 60 percent of spam and 10 percent of other mail, and a second feature happens to be an exact copy of it (in real text, a word and its plural, or “click” and “here”). The true posterior from seeing the word, with equal priors, is 0.857; the second feature adds no information, but naive Bayes multiplies its likelihood in again and reports 0.973. Correlated features are counted as independent evidence, which makes naive Bayes overconfident even when its decisions are reasonable.

1:13:46 The parameters are $\phi_{j|y=1} = p(x_j = 1 \mid y = 1)$, the chance that word j appears in a spam e-mail, $\phi_{j|y=0}$ for non-spam, and the prior $\phi_y = p(y = 1)$ — $2n + 1$ numbers, 20,001 for the 10,000-word vocabulary, instead of a number with three thousand digits. 1:15:00 Maximising the joint likelihood, exactly as for GDA, gives counts:

$$\phi_y = \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\}, \qquad \phi_{j|y=1} = \frac{\sum_i 1\{x_j^{(i)} = 1,\ y^{(i)} = 1\}}{\sum_i 1\{y^{(i)} = 1\}}, \tag{5.7}$$

and the same with $y = 0$. Of all the spam e-mails in the training set, what fraction contain “buy”? That fraction is $\phi_{\text{buy}|y=1}$. Prediction multiplies these numbers by Equation 5.6 and applies Bayes’ rule.

Naive Bayes has the same relationship to logistic regression that GDA has. Its log-odds is a sum over words of terms that depend on x_j only through $x_j \in \{0, 1\}$, so it is exactly linear: $\log \frac{p(y=1|x)}{p(y=0|x)} = b + w^\top x$ with $w_j = \log \frac{\phi_{j|1}(1-\phi_{j|0})}{\phi_{j|0}(1-\phi_{j|1})}$. The generator checks this on every input of a six-word vocabulary. Naive Bayes is a linear classifier whose weights are set by counting rather than by optimisation.

1:17:32 Ng’s verdict is that naive Bayes is “not too horrible”, that logistic regression does better almost all the time, and that naive Bayes is nonetheless very efficient: fitting is counting,

prediction is multiplying, and the counts can be updated as new labelled mail arrives without refitting anything. This is the regime Ng and Jordan studied: naive Bayes reaches its own, higher, asymptotic error with fewer examples. 1:18:14 Ng flags one flaw that makes the version above "nearly work": zeros. If a word never appears in any training e-mail, its estimate is 0 in both classes, an e-mail containing it gets $p(x \mid y) = 0$ under both, and the posterior is $0/0$. Even a word seen only in non-spam drives the spam probability of any e-mail containing it to exactly zero, however spammy the rest. The fix, Laplace smoothing, opens the next lecture.

> **In the official notes**
>
> cs229-notes2, section 2, develops naive Bayes for the same spam example; Laplace smoothing (2.1) and the multinomial event model for text (2.2) come with lecture 6.

6. Support Vector Machines

Lecture 6

Lecture video: Lecture 6: Support Vector Machines · 1:20:57

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Naive Bayes, as lecture 5 left it, breaks the first time a word turns up that the training set never contained: every class gets probability zero and the posterior is 0/0. The fix is to add one to every count, and that rule, Laplace smoothing, is what a Bayesian with a uniform prior would compute anyway. With smoothing in place the lecture builds a second naive Bayes for text, the multinomial event model, which counts how often each word occurs instead of only whether it does, and then steps back to argue that an algorithm this cheap is the right first thing to build. The second half starts the support vector machine. Before kernels, an SVM is a linear classifier chosen for one property: of all the lines that separate the training data, it takes the one that stays furthest from every example. Most of the work is in defining "furthest" so that it cannot be gamed.

6.1. Laplace smoothing

4:42 Ng's example of the zero problem is a word that is new to your inbox but not to the world. Suppose "NIPS", the machine learning conference, is word 6017 of a 10,000-word vocabulary, and nobody has yet e-mailed you about it. The maximum likelihood estimates from lecture 5 are fractions of e-mails containing the word, so both $\phi_{6017|y=1}$ and $\phi_{6017|y=0}$ come out as 0. 8:03 Months later your project team writes about submitting to NIPS. The product $\prod_j p(x_j \mid y)$ now contains a factor of zero for both classes, and the classifier's posterior is $0/(0+0)$. The division by zero is the visible symptom; the underlying error, Ng says, is statistical — estimating the probability of something as zero because you have not seen it yet.

9:58 He sets spam aside to make the point with a smaller example. One season, Stanford's football team played four out-of-state games — Wake Forest, Oregon State, Arizona and, Ng adds for the room, Caltech — and lost all four. You followed them to every one. What is the chance they win the next? Maximum likelihood counts wins over games played, $0/4 = 0$: absolute certainty of a loss, which is unkind and also unreasonable. 11:59 Laplace smoothing pretends you saw one more of each outcome than you did — one extra win and one extra loss — so the estimate becomes $(0+1)/(4+2) = 1/6$.

12:58 The rule is not arbitrary. Laplace derived it for the probability that the sun rises tomorrow: having seen it rise every day so far, you should be very confident but not certain. Ng compresses the justification into one sentence, which is worth unpacking. If the unknown win probability p is given a uniform prior on $[0, 1]$ and you observe s wins in n games, the posterior is proportional to $p^s(1-p)^{n-s}$, and its mean is

$$\mathbb{E}[p \mid \text{data}] = \frac{\int_0^1 p \cdot p^s (1-p)^{n-s}\, dp}{\int_0^1 p^s (1-p)^{n-s}\, dp} = \frac{s+1}{n+2}. \tag{6.1}$$

That is exactly the add-one estimate. It is also the posterior predictive probability that the *next* game is a win, which is the quantity a classifier needs, so "Bayesian optimal" in the lecture means this: the posterior mean, the best guess under squared error. The generator integrates the posterior numerically for 0 wins in 4 and recovers $1/6$ to six decimal places, then repeats the check for other counts.

14:22 For a variable with k possible values the rule adds one to each count and k to the denominator,

$$\hat{p}(x = j) = \frac{\sum_{i=1}^{m} 1\{x^{(i)} = j\} + 1}{m + k}, \tag{6.2}$$

which still sums to one over the k values and never returns exactly 0 or 1. 15:14 For the binary word features of naive Bayes, $k = 2$:

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1,\ y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} + 2},$$

and the same for $y = 0$. A word never seen in training now gets a small positive probability in both classes, the zero factor disappears, and the NIPS e-mail is classified on the strength of its other words. Ng's assessment of the resulting filter is measured — "not a great spam classifier but it's not terrible either" — and its virtue is that training is counting and prediction is multiplying.

6.2. Discretising a continuous feature

17:58 Naive Bayes extends past binary features with no change to its structure. Ng's example goes back to houses, now as a classification problem — will this house sell within 30 days? — with floor area as a feature. Cut the area into buckets (under 400 square feet, 400 to 800, 800 to 1,200, over 1,200), let $x_j \in \{1, 2, 3, 4\}$ record which bucket a house falls in, and model $p(x_j \mid y)$ as a multinomial over four values instead of a Bernoulli over two. The factorisation $p(x \mid y) = \prod_j p(x_j \mid y)$ is unchanged, the estimates are still counts, and Laplace smoothing adds $k = 4$ to each denominator. 19:29 His rule of thumb when discretising is ten buckets; it "often seems to work well enough", and the board had room for four.

6.3. The multinomial event model

20:17 For text specifically there is a better variant, and Ng motivates it with what the binary representation throws away. Take the spam e-mail "drugs buy drugs now", with "buy", "drugs" and "now" at positions 800, 1600 and 6200 of the vocabulary. 21:40 Under the representation so far, x is a 10,000-dimensional binary vector with three ones. The word "drugs" appears twice, and the vector records it once; the e-mail "drugs buy now" maps to exactly the same x (Figure 6.1). A word that a spammer repeats is arguably stronger evidence than one that appears once, and the binary model cannot see the difference.

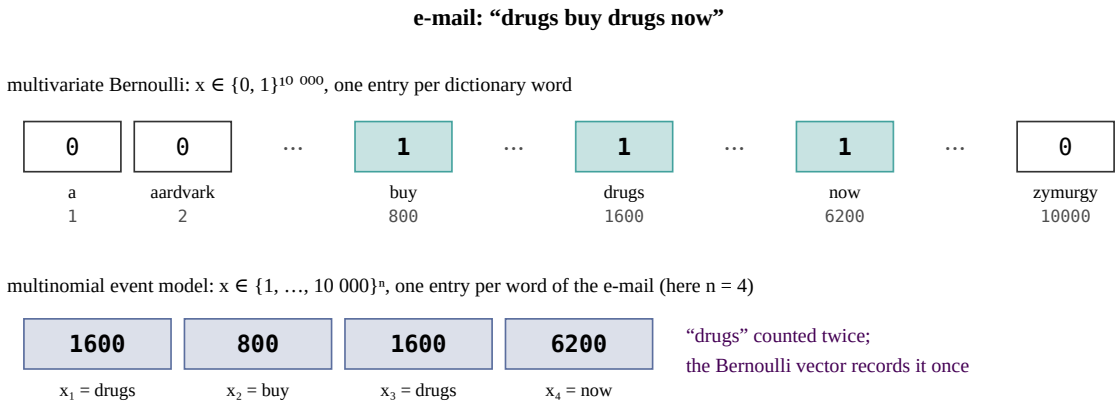


Figure 6.1.: The e-mail “drugs buy drugs now” under the two event models. Above, one binary entry per dictionary word; below, one dictionary index per word of the e-mail, so the repeated word is represented twice.

22:34 The alternative represents an e-mail of n words as a vector of length n, whose jth entry $x_j \in \{1, \dots, 10{,}000\}$ is the dictionary index of the jth word: here $x = (1600, 800, 1600, 6200)$. The length now varies from e-mail to e-mail, n_i for the ith training example, which also answers a property of text Ng points out — e-mails can be five words long or a thousand, and the binary representation forces both into one fixed-length vector.

23:57 The names are, in Ng’s words, “very confusing, very horrible names”, but they are the ones the field uses. The binary model is the *multivariate Bernoulli event model* — 10,000 coin tosses, one per word, each deciding whether that word appears. The new one is the *multinomial event model*. 24:49 Ng credits the naming to his friend Andrew McCallum, and the paper that compares the two, by McCallum and Nigam, says it set out to clear up exactly this confusion.

25:12 The generative story is that the sender first decides spam or not, with probability ϕ_y, and then draws each word of the e-mail independently from one distribution over the vocabulary that depends on the class. So

$$p(x, y) = p(y) \prod_{j=1}^{n} p(x_j \mid y),$$

which looks identical to lecture 5’s naive Bayes and means something different: the product now runs over the words *of the e-mail* rather than over the dictionary, and each factor is a multinomial over 10,000 outcomes rather than a coin toss. The parameters are ϕ_y and $\phi_{k|y=c} = p(x_j = k \mid y = c)$, the chance that a word in a class-c e-mail is word k. 28:05 Ng puts a question to the room — “the chance of word blank being blank” — and the answer, *any* position being word k, exposes the hidden assumption: the distribution does not depend on j. The first word of a spam e-mail is “drugs” with the same probability as the third, which is why j does not appear on the left-hand side.

29:41 The maximum likelihood estimates are again counts, now over word occurrences instead of e-mails, and with Laplace smoothing they are

$$\phi_{k|y=0} = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k,\ y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}\, n_i + 10{,}000}. \tag{6.3}$$

In English: pool every word of every non-spam e-mail, and ask what fraction of them are word k. If the non-spam e-mails contain 100,000 words in all and "drugs" is 200 of them, the unsmoothed estimate is 200/100,000. 31:59 Ng asks the room what to add to the denominator for smoothing, and the answer is the number of values x_j can take: the vocabulary size, 10,000, not two.

33:44 A student asks what to do with a word that is not in the dictionary at all. Either drop it, or map every out-of-vocabulary word to one special token, conventionally `UNK`, which then gets a probability like any other word. The second keeps the information that the e-mail contained rare words, which can itself be evidence.

The multinomial model is still a linear classifier, now in word *counts*. Taking the log of Bayes' rule, the log-odds of spam is

$$\log \frac{p(y=1 \mid x)}{p(y=0 \mid x)} = \log \frac{\phi_y}{1-\phi_y} + \sum_{k=1}^{10{,}000} c_k \log \frac{\phi_{k|y=1}}{\phi_{k|y=0}},$$

where c_k is the number of times word k occurs in the e-mail. The generator confirms this against Bayes' rule on 200 random e-mails. A repeated word now adds its weight once per occurrence, which is precisely the information the Bernoulli model discarded. McCallum and Nigam's comparison on five corpora found the Bernoulli model competitive with small vocabularies and the multinomial model usually better with large ones, averaging a 27 percent reduction in error — "much better", as the lecture puts it, for the typical text problem.

> **In the official notes**
>
> cs229-notes2, section 2.1, gives Laplace smoothing for the binary naive Bayes model; section 2.2 writes out the multinomial event model's likelihood and both smoothed estimates.

6.4. Build something quick and dirty first

35:26 Naive Bayes is rarely the most accurate choice — logistic regression usually beats it — so the question is what it is for. Ng's answer is a piece of working advice that appears in no PDF. 36:55 If your goal is to make a learning algorithm work for an application, rather than to invent a new algorithm, start by implementing something simple and fast, train it, look at how it fails, and let the failures decide what to build next. 38:11 His analogy is software: nobody writes 10,000 lines of code and compiles them for the first time at the end; you write a module, test it, and build up. A complicated learning system designed in full before any of it runs is the same mistake.

39:07 The spam filter shows why. Ng started a conference on e-mail and anti-spam, and his example list of promising ideas is long. Spammers misspell deliberately — "m0rtgage", or the word broken up with slashes — so a filter keyed to words sees an unknown token where a person sees "mortgage"; one of his students published on mapping such spellings back to words. Spammers spoof headers to hide where mail came from. You could fetch the URLs in an e-mail and analyse the pages behind them. 40:59 Each of these is three to six months of work, and at the start you cannot know which one matters. So build the crude filter, then read the e-mails it gets wrong. If many of them are deliberate misspellings, you now have evidence that the misspelling problem is worth months; if few are, "don't bother" — and you have saved the months. This is error analysis, which the course returns to later.

42:06 That is the niche for naive Bayes and GDA. Neither will give the best accuracy; logistic regression, SVMs and neural networks almost always do better. But both are non-iterative — counting, or computing means and covariances — and short to write, so they are a fast way to get the first system running. Ng's advice for course projects follows: do not spend weeks designing; get a data set and apply something simple, logistic regression or naive Bayes rather than a neural network, and go from there.

44:26 A student points to a weakness of the word features themselves: "mother" and "father", or "one" and "two", are entirely unrelated indices to naive Bayes, which has no way to know that similar words should behave similarly. Ng agrees and names the fix from neural networks, *word embeddings*, which represent each word by a vector learned so that related words land close together. Because the embedding comes with that knowledge built in, a classifier built on it needs fewer labelled examples. The course leaves the topic to CS230 and CS224N.

6.5. Why support vector machines

46:22 The SVM enters with a picture: two classes in the plane whose boundary is clearly curved. Logistic regression and GDA, as they stand, draw straight lines. 47:06 You can get a curve out of logistic regression by inventing features — map (x_1, x_2) to $\phi(x) = (x_1, x_2, x_1^2, x_2^2, x_1x_2, x_1^3, \dots)$ and fit the same linear model to $\phi(x)$ — and with the quadratic terms the boundary can become an ellipse (Figure 6.2). The generator fits both models to the same data: on the raw features logistic regression misclassifies a third of the training set, with the quadratic terms 12 percent.

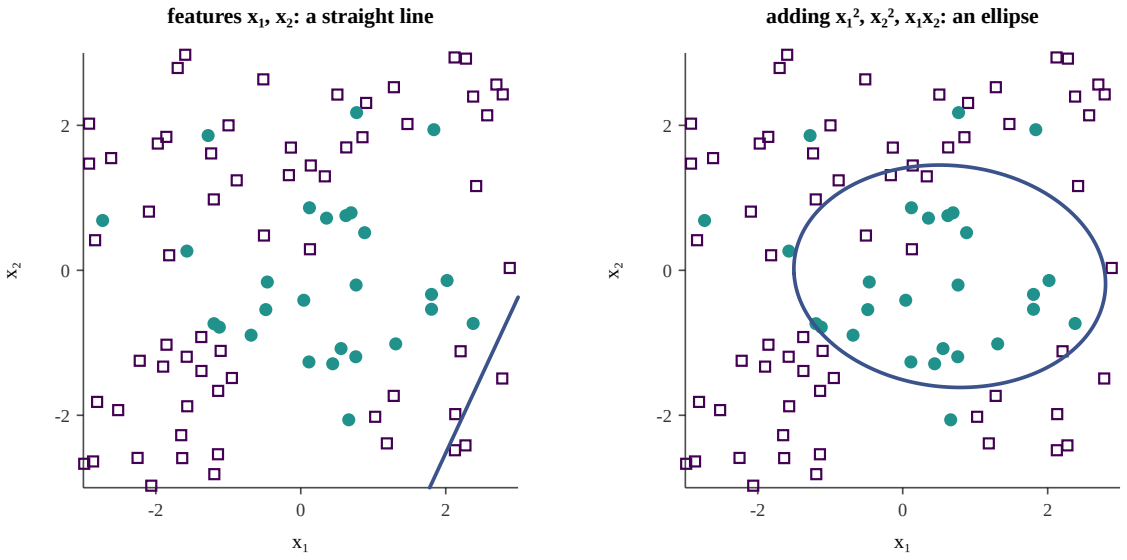


Figure 6.2.: Logistic regression on the same data with and without hand-picked features. With x_1, x_2 alone the boundary must be a straight line and it gives up; with the squares and the product added, the boundary is an ellipse around the inner class.

48:05 The trouble is choosing the features. An ellipse needs quadratic terms; Ng admits he would not know which features produce the irregular boundary he drew, and nobody wants to guess between $\sqrt{x_1}$ and $(x_1x_2)^{2/3}$. The SVM's promise is to map x into a much higher-dimensional feature space — 51:15 hundreds of thousands of dimensions, or infinitely many — and fit a linear classifier there, which is non-linear back in the original space, without anyone choosing the features by hand. That is the *kernel* idea of lecture 7.

48:48 Ng's other reason is practical. The SVM is "turnkey": mature software solves its optimisation problem reliably, without a learning rate to tune or a convergence to babysit. He is explicit that for many problems neural networks are now more effective, and equally explicit that practice uses a much wider set of tools than the press coverage of deep learning suggests.

49:57 The road map has three stages: the *optimal margin classifier* for data that a straight line can separate, which this lecture starts; kernels, which run it in huge feature spaces; and a modification for data that cannot be separated. The first stage needs two definitions of margin.

6.6. The functional margin

54:50 Ng motivates the margin with logistic regression. The classifier predicts 1 when $\theta^\top x \geq 0$, but a correct prediction can be confident or marginal. If $y^{(i)} = 1$ we would like $\theta^\top x^{(i)} \gg 0$, so that $g(\theta^\top x^{(i)})$ is close to 1; if $y^{(i)} = 0$ we would like $\theta^\top x^{(i)} \ll 0$. A classifier for which both hold on every example is correct and confident.

1:00:02 The SVM changes notation to make the algebra cleaner. Labels are $y \in \{-1, +1\}$, the hypothesis outputs a hard ± 1 instead of a probability, $g(z) = 1$ if $z \geq 0$ and -1 otherwise, and the intercept is pulled out of the parameter vector: $h_{w,b}(x) = g(w^\top x + b)$, with b playing the role of θ_0 and w of $\theta_1, \dots, \theta_n$, and no $x_0 = 1$ convention.

1:03:36 The *functional margin* of the hyperplane (w, b) with respect to the example $(x^{(i)}, y^{(i)})$ is

$$\hat{\gamma}^{(i)} = y^{(i)} \left(w^\top x^{(i)} + b \right). \tag{6.4}$$

The label folds the two hopes into one: if $y^{(i)} = 1$ we want $w^\top x^{(i)} + b$ large and positive, if $y^{(i)} = -1$ large and negative, and either way we want $\hat{\gamma}^{(i)}$ large. 1:06:30 Its sign alone already says something: $\hat{\gamma}^{(i)} > 0$ exactly when the example is classified correctly. 1:07:46 For the whole training set, the SVM takes the worst case, $\hat{\gamma} = \min_i \hat{\gamma}^{(i)}$ — how well the classifier does on the example it does worst on. That is a harsh notion, and it makes sense for now only because this lecture assumes the data are linearly separable.

1:09:01 The functional margin has a flaw Ng calls cheating. Multiply w and b by 2, or 10, and every $\hat{\gamma}^{(i)}$ multiplies by the same factor, yet the boundary $w^\top x + b = 0$ is the same line and every prediction is unchanged (the generator checks this for factors of 2, 10 and 1/17, Ng's choices). So "make the functional margin large" is not a goal at all: any separating classifier achieves an arbitrarily large one. 1:10:00 The fix is to normalise — require $\|w\| = 1$, or equivalently divide w and b by $\|w\|$ — and the freedom to rescale (w, b) without changing the classifier turns out to be useful later rather than only a nuisance.

6.7. The geometric margin

1:11:29 The geometric margin is the normalised version, and it has a picture. The line $w^\top x + b = 0$ splits the plane into the half where the classifier predicts $+1$ and the half where it predicts -1, and the vector w is perpendicular to the line, pointing into the positive half. 1:12:33 For a correctly classified positive example A, the *geometric margin* $\gamma^{(i)}$ is its Euclidean distance to the line: the length of the segment from A to the nearest point B on the boundary (Figure 6.3).

1:13:49 Ng states the formula and leaves the derivation to the notes; it is short enough to give. B is A moved back along the unit normal by the distance $\gamma^{(i)}$, so $B = x^{(i)} - \gamma^{(i)} w / \|w\|$, and B

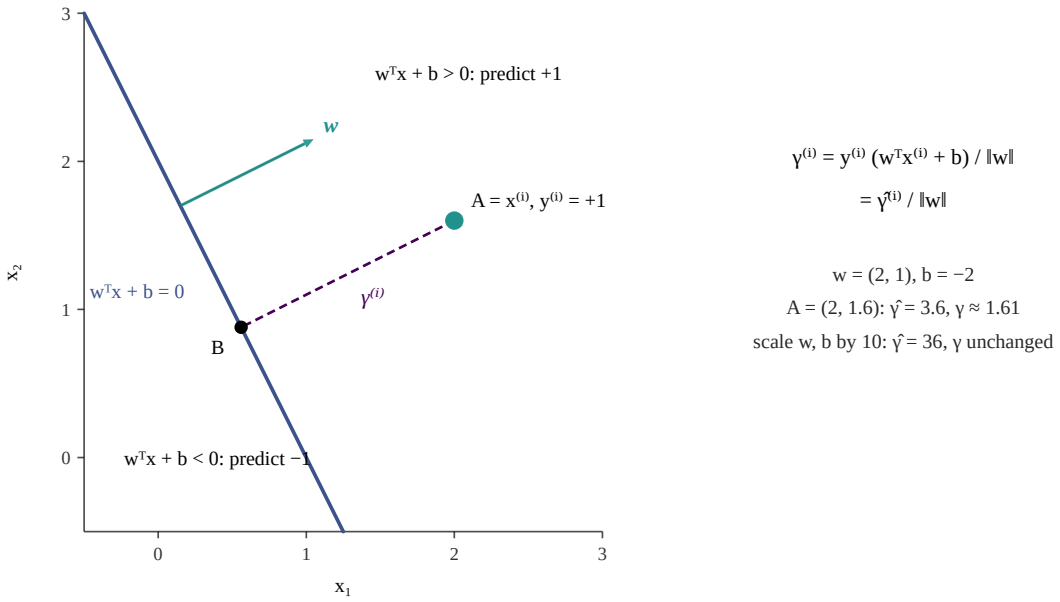


Figure 6.3.: The geometric margin of one example. The boundary is $w^\top x + b = 0$ with $w = (2, 1)$ and $b = -2$; w is perpendicular to it; the positive example A sits at distance $\gamma^{(i)}$ from its foot B on the line.

lies on the line, so $w^\top B + b = 0$. Solving gives $\gamma^{(i)} = (w^\top x^{(i)} + b)/\|w\|$ for a positive example, and including the label makes it hold for negative ones too:

$$\gamma^{(i)} = y^{(i)} \left(\frac{w^\top x^{(i)} + b}{\|w\|} \right) = \frac{\hat{\gamma}^{(i)}}{\|w\|}. \tag{6.5}$$

For the point $A = (2, 1.6)$ in Figure 6.3, the functional margin is 3.6 and the geometric margin $3.6/\sqrt{5} \approx 1.61$; multiply w and b by ten and the functional margin becomes 36 while the geometric margin stays at 1.61. The geometric margin is the functional margin with the cheat removed. The generator checks Equation 6.5 against an explicit projection onto the hyperplane for 500 random hyperplanes and points in three dimensions. 1:15:23 As before, the training set's geometric margin is the worst case, $\gamma = \min_i \gamma^{(i)}$.

6.8. The optimal margin classifier

58:15 Ng drew the motivating picture well before the definitions. Two lines both separate a training set perfectly; one passes close to a few of the examples, the other keeps a wide berth from both classes, and the second "looks much better". The geometric margin is what "looks better" means: the first line's worst example is barely on the right side, the second line's is far from it (Figure 6.4).

1:16:28 The *optimal margin classifier* chooses w and b to maximise the geometric margin of the training set. 1:17:59 Ng poses it as a problem in three unknowns, γ, w and b:

$$\max_{\gamma, w, b} \ \gamma \quad \text{subject to} \quad \frac{y^{(i)}(w^\top x^{(i)} + b)}{\|w\|} \geq \gamma, \quad i = 1, \ldots, m. \tag{6.6}$$

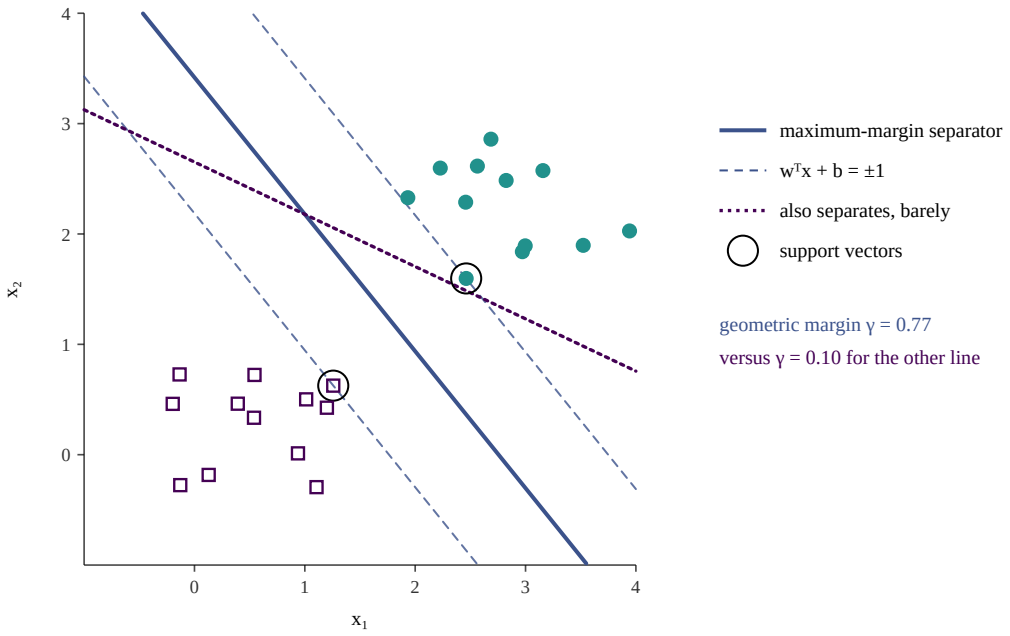


Figure 6.4.: Two lines that both separate the training set. The dotted one passes within 0.10 of the closest example; the solid one, the maximum-margin separator, keeps a geometric margin of 0.77 from both classes. The dashed lines are where $w^\top x + b = \pm 1$, and the circled examples on them are the support vectors.

Every example must have geometric margin at least γ, and γ is pushed as high as the constraints allow, so at the optimum γ equals the worst example's margin and that worst margin is as large as it can be.

1:19:03 In this form the problem is not convex, so there is no guarantee that a local method finds the global optimum. 1:19:13 Ng states, and leaves the steps to lecture 7 and the notes, that a few rewrites turn it into an equivalent convex problem: minimise $\frac{1}{2}\|w\|^2$ subject to every functional margin being at least 1. His one-line intuition is that the margin is the functional margin divided by $\|w\|$, so a smaller w means a smaller divisor and a larger geometric margin. Off-the-shelf solvers handle the convex form, and for separable data its solution is the optimal margin classifier — "really a baby SVM", which kernels will grow up.

The solution has a property the picture already shows. In Figure 6.4 only two examples touch the dashed lines; they are the *support vectors*, and the separator depends on them alone. The generator finds this separator in two independent ways. It searches over directions for the one that leaves the widest gap between the classes, and then checks the result against the optimality conditions of the convex problem: every functional margin is at least 1, the two closest examples have functional margin exactly 1, and $w = \sum_i \alpha_i y^{(i)} x^{(i)}$ over those two examples alone, with both $\alpha_i > 0$. That last fact — the solution is built from a few training examples — is where lecture 7 begins.

> **In the official notes**
>
> cs229-notes3, sections 1–4, defines the functional and geometric margins with the same notation and gives the full chain of rewrites from Equation 6.6 to the convex quadratic program.

7. Kernels

Lecture 7

Lecture video: Lecture 7: Kernels · 1:20:25

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

A support vector machine is the optimal margin classifier of lecture 6 run in a feature space of enormous, possibly infinite, dimension. That sounds impossible to compute, and the lecture's argument is a chain of three observations that make it cheap. The optimal w is always a weighted sum of the training examples. Once w is written that way, the training problem and the prediction rule touch the data only through inner products $\langle x^{(i)}, x^{(j)} \rangle$. And for some very large feature maps ϕ, the inner product $\phi(x)^\top \phi(z)$ can be computed directly from x and z in time proportional to their own dimension — that shortcut is a *kernel*. Ng calls the SVM one of his favourite algorithms because it is "very turnkey": the optimisation is convex and good packages solve it. The lecture ends by dropping the assumption that the data are separable and with kernels designed for inputs that are not vectors at all.

7.1. From the largest margin to the smallest w

1:13 Lecture 6 posed the optimal margin classifier as: choose γ, w and b to maximise γ, subject to every example having geometric margin at least γ. 5:31 Ng spends a minute on why that maximises the *worst* margin. If three examples had geometric margins 17, 2 and 5, the set's margin is 2, and the only way to raise γ under the constraints is to raise the margin of whichever example is lowest. Pushing γ up as far as it goes is the same as maximising $\min_i \gamma^{(i)}$.

6:33 The step to a tractable problem uses the freedom that made the functional margin useless in lecture 6. Take $w = (2, 1)$ and $b = -2$: the boundary $2x_1 + x_2 = 2$ crosses the axes at $(1, 0)$ and $(0, 2)$, and so does the boundary for $10w$ and $10b$, or for $w/10$ and $b/10$. Because rescaling (w, b) changes nothing about the classifier, we may pick the scale, and 8:49 the convenient choice is the one that makes the worst functional margin exactly 1: $\min_i y^{(i)}(w^\top x^{(i)} + b) = 1$. Then the geometric margin is $\gamma = \hat{\gamma}/\|w\| = 1/\|w\|$, maximising it is minimising $\|w\|$, and minimising the smooth $\frac{1}{2}\|w\|^2$ has the same solution:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y^{(i)}\left(w^\top x^{(i)} + b\right) \geq 1, \quad i = 1, \dots, m. \tag{7.1}$$

This is a convex quadratic objective under linear constraints — a quadratic program, with one global optimum and reliable solvers. On lecture 6's example set the solution has $\|w\| = 1.293$, so $\gamma = 1/\|w\| = 0.773$, the same margin lecture 6's generator found by searching over directions; the two examples that touch the margin have functional margin exactly 1 and every other example more.

7.2. The representer theorem

10:34 So far x has been two- or a hundred-dimensional. The goal is feature vectors with a hundred trillion dimensions, or infinitely many, where w cannot even be stored. 11:07 The way through is to assume that w is a linear combination of the training examples,

$$w = \sum_{i=1}^{m} \alpha_i \, y^{(i)} x^{(i)}, \tag{7.2}$$

with the labels $y^{(i)} = \pm 1$ written in by convention because they make later algebra tidier. 11:57 Ng presents this as an assumption but stresses that it is not one: the *representer theorem* proves that the optimal w always has this form, so nothing is lost. He skips the proof as long and involved and gives two intuitions.

13:11 The first is from gradient descent. Run stochastic gradient descent for logistic regression from $\theta = 0$. Each update adds a multiple of one training example, $\theta := \theta + \alpha \, (y^{(i)} - h_\theta(x^{(i)})) \, x^{(i)}$ (this α is the learning rate, unrelated to the α_i above). By induction, θ is a linear combination of training examples after every step, however many steps you take; 15:22 batch gradient descent is the same, and so is gradient descent on the SVM's own objective. The generator runs 300 SGD steps in five dimensions on three examples and finds θ within 10^{-15} of the span of those three vectors.

16:09 The second is geometric. 16:50 The vector w is perpendicular to the decision boundary: it fixes the boundary's orientation, and b slides the boundary along it. 18:39 Now suppose the inputs have three features but every training example has $x_3 = 0$, so the data lie in the x_1x_2-plane. Then, Ng says, w_3 should be 0 — w should lie in the space the examples span.

That second intuition is worth sharpening, because as stated it is not true of every separator. With all $x_3 = 0$, the term w_3x_3 is zero on every training example, so *any* value of w_3 classifies the training set identically: the generator takes the SVM's w and sets $w_3 = 5$, and every margin constraint still holds. What forces $w_3 = 0$ is the objective. A non-zero w_3 adds w_3^2 to $\|w\|^2$ and buys nothing in the constraints, so the *minimum-norm* solution drops it. That is the real content of the representer theorem: among all the w that fit the data equally well, minimising $\|w\|^2$ (or, in general, any loss plus a squared-norm regulariser) picks one in the span of the data.

7.3. Everything is an inner product

20:34 Substitute Equation 7.2 into Equation 7.1. Since $\|w\|^2 = w^\top w$, the objective becomes

$$\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j \, y^{(i)} y^{(j)} \left\langle x^{(i)}, x^{(j)} \right\rangle,$$

and the constraints become $y^{(i)}\big(\sum_j \alpha_j y^{(j)} \langle x^{(j)}, x^{(i)} \rangle + b\big) \geq 1$, where $\langle x, z \rangle = x^\top z$ is the inner product. 24:13 The point is where the inputs now appear: nowhere except inside inner products. Whatever the dimension of x, the problem is stated in terms of the $m \times m$ table of inner products between training examples.

25:14 The form actually solved in practice goes one step further. Ng skips the derivation, which uses Lagrange duality, and writes down the result, the *dual optimisation problem*:

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \left\langle x^{(i)}, x^{(j)} \right\rangle \quad \text{subject to} \quad \alpha_i \geq 0, \quad \sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{7.3}$$

b has dropped out, to be recovered afterwards from any example with $\alpha_i > 0$. Two facts about Equation 7.3 go beyond the lecture and are worth having. First, it is not merely a simplification of the substituted problem: when there are more examples than dimensions, many different α give the same w, and the dual picks out particular ones — non-negative, and zero for every example that is not on the margin. Those with $\alpha_i > 0$ are the support vectors. Second, its optimal value equals the primal's: on lecture 6's data the generator solves Equation 7.3 by sequential minimal optimisation, the algorithm in the official notes, and finds $W(\alpha^*) = \frac{1}{2}\|w^*\|^2 = 0.8364$, with only 2 of the 24 α_i non-zero, and the resulting w and b agree with lecture 6's direction search to five decimal places.

27:05 Prediction is an inner product too:

$$h(x) = g(w^\top x + b) = g\left(\sum_{i=1}^{m} \alpha_i y^{(i)} \left\langle x^{(i)}, x \right\rangle + b \right). \tag{7.4}$$

28:20 Once the α_i are stored, both training and prediction need nothing but inner products of inputs — and since most α_i are zero, prediction sums only over the support vectors.

> **In the official notes**
>
> cs229-notes3, section 5, derives Lagrange duality and the KKT conditions, and section 6 applies them to obtain Equation 7.3 and the formula for b. The representer theorem, with a proof for general losses, is in the supplementary notes on the representer theorem, section 2.

7.4. The kernel trick

28:42 Ng gives the recipe in four steps. First, write the whole algorithm in terms of inner products $\langle x, z \rangle$. Second, choose a feature map ϕ from the original inputs to a higher-dimensional space — 30:12 a house's floor area x could become (x, x^2, x^3, x^4), or area and bedrooms (x_1, x_2) could become all their products up to some degree. Third, find a way to compute

$$K(x, z) = \phi(x)^\top \phi(z), \tag{7.5}$$

the *kernel function*, cheaply. Fourth, replace every $\langle x, z \rangle$ in the algorithm by $K(x, z)$. 32:19 The algorithm is then running on $\phi(x)$, in the high-dimensional space, without ever forming $\phi(x)$. Forming it is what would be expensive: Ng has seen problems with a billion features, and even 100,000 makes every vector operation costly.

33:55 A student asks about the no-free-lunch theorem, and Ng's answer frames why this works at all. The theorem — Wolpert's form of it for supervised learning shows that, averaged over all possible target functions, no learning algorithm has lower off-training-set error than any other — says that in the worst case learning does not work. Ng finds it less useful than fascinating, because the problems we meet are not drawn adversarially: "the universe is not that hostile

toward us", and an inductive bias such as "the boundary is smooth in some feature space" tends to pay off.

7.5. Polynomial kernels

34:42 The first example shows the saving. Let $x \in \mathbb{R}^n$ and let $\phi(x)$ list every product $x_i x_j$ — for $n = 3$, the nine terms $x_1x_1, x_1x_2, x_1x_3, x_2x_1, \dots, x_3x_3$, duplicates included. 35:38 So $\phi(x) \in \mathbb{R}^{n^2}$: with $n = 10{,}000$ that is 10^8 features, and computing $\phi(x)^\top \phi(z)$ explicitly takes order n^2 time. 37:18 But

$$(x^\top z)^2 = \Big(\sum_{i=1}^{n} x_i z_i\Big)\Big(\sum_{j=1}^{n} x_j z_j\Big) = \sum_{i=1}^{n}\sum_{j=1}^{n} (x_i x_j)(z_i z_j) = \phi(x)^\top \phi(z), \tag{7.6}$$

and the left-hand side takes order n time: one inner product of n-vectors, then one squaring. The middle expression is the explicit computation — march down both lists of products, multiply, add — and the identity says that the cheap computation gives the same number.

41:11 Adding a constant, $K(x, z) = (x^\top z + c)^2$, corresponds to appending the first-degree terms and a constant to the features: $\phi(x)$ now also contains $\sqrt{2c}\,x_1, \dots, \sqrt{2c}\,x_n$ and c. The constant c sets the relative weight of the linear terms against the quadratic ones. 42:29 More generally, $K(x, z) = (x^\top z + c)^d$ is still order n to compute and corresponds to a $\phi(x)$ containing every monomial of degree up to d — terms like $x_1x_2x_5x_{17}x_{29}$ or $x_1x_2^2x_3x_{18}$ when $d = 5$ — each with a weight that involves c and a multinomial coefficient. The generator builds these explicit feature vectors for $d = 2$ and 3 and confirms that their inner product equals the kernel on 200 random pairs.

42:56 Ng gives the number of distinct monomials as $\binom{n+d}{d}$, which the generator confirms by enumeration, and then glosses it as "roughly $(n + d)^d$, very roughly". The binomial coefficient is the right count; the gloss is high by about a factor of $d!$. For $n = 10{,}000$ and $d = 5$, $\binom{n+d}{d} \approx 8.3 \times 10^{17}$ while $(n + d)^d \approx 1.0 \times 10^{20}$, a ratio of $120 = 5!$. The point survives — the feature space grows like $n^d/d!$, astronomically, while the kernel costs order n for every d.

44:23 This is what Ng means by the SVM: the optimal margin classifier plus the kernel trick. Choose a kernel, solve Equation 7.3 with $K(x^{(i)}, x^{(j)})$ in place of the inner products, and predict with Equation 7.4 likewise; the computation scales with the dimension of x, not of $\phi(x)$. 45:51 A student asks whether a kernel belongs to one feature map. Essentially yes: up to trivial changes such as permuting the features, which leave every inner product unchanged, a different feature map needs a different kernel.

7.6. Seeing the lift

47:09 To show what this does geometrically, Ng plays a short visualisation by Udi Aharoni. Red and blue points in the plane cannot be separated by a line; a feature map lifts them into three dimensions, where a plane separates them; and the plane, viewed back in the original plane, cuts out an ellipse. Figure 7.1 does the same thing one dimension lower, so it fits on a page. The positives sit between the negatives on a line, so no single threshold separates them. Map each x to $\phi(x) = (x, x^2)$ and the points land on a parabola, where the SVM finds a straight line; pulled back to the original line, that straight line becomes the two thresholds $x = -1.30$ and $x = 1.40$.

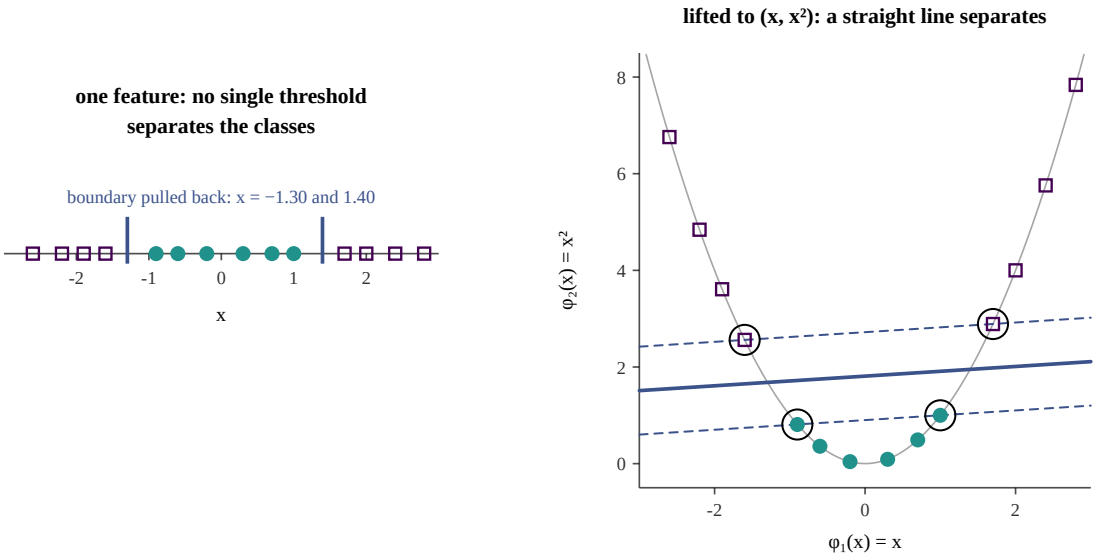


Figure 7.1.: A one-dimensional data set lifted by $\phi(x) = (x, x^2)$. On the left no threshold separates the classes; on the right the SVM separates them with a straight line, and the four circled support vectors fix it. Pulled back to the line, the linear boundary is the pair of thresholds marked on the left.

49:10 The boundary is linear in a high-dimensional space and very non-linear in the original one. 49:27 A student asks whether the data must be linearly separable in that high-dimensional space; so far the derivation has pretended so, and the soft margin below removes the assumption.

7.7. Which functions are kernels

50:02 Ng's working intuition is that a kernel measures similarity. An inner product is large when two vectors point the same way and small when they point in unrelated directions, so $K(x, z)$ should be large when x and z are alike and small when they are not. 51:19 By that standard, here is a function pulled "out of the air":

$$K(x, z) = \exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right), \tag{7.7}$$

close to 1 when x and z are near each other and close to 0 when they are far apart. 52:13 Can it be used? Only if some ϕ makes $K(x, z) = \phi(x)^\top \phi(z)$; the whole derivation assumed so, and a K without such a ϕ breaks it — the optimisation problem may no longer correspond to any classifier at all.

53:01 One necessary condition is immediate: $K(x, x) = \phi(x)^\top \phi(x) \geq 0$. 53:27 A stronger one comes from taking any m points $x^{(1)}, \ldots, x^{(m)}$ and forming the *kernel matrix* (or Gram matrix) $K_{ij} = K(x^{(i)}, x^{(j)})$. For any vector z,

$$z^\top K z = \sum_i \sum_j z_i \, \phi(x^{(i)})^\top \phi(x^{(j)}) \, z_j = \sum_k \Big(\sum_i z_i \, \phi_k(x^{(i)}) \Big)^2 \geq 0,$$

so a valid kernel always produces a positive semidefinite kernel matrix (and a symmetric one, since inner products are). 57:21 *Mercer's theorem* says the converse holds too: K is a valid kernel

if and only if, for every finite set of points, the kernel matrix is symmetric positive semidefinite. Ng proves only the direction above and states the other.

The similarity intuition is a good guide but not a test, and the gap is easy to fall into. "Two points are similar if they are within 1.5 of each other" sounds like a reasonable similarity, $K(x,z) = 1\{|x-z| < 1.5\}$. At the three points $0, 1, 2$ its kernel matrix has eigenvalues $1+\sqrt{2}$, 1 and $1-\sqrt{2} < 0$; the vector $z = (1, -\sqrt{2}, 1)$ gives $z^\top K z \approx -1.66$. No feature map produces that matrix, so it is not a kernel, and the SVM built on it can fail to be a convex problem. The generator checks this alongside the Gaussian kernel, whose kernel matrix on 25 random points has every eigenvalue non-negative.

59:13 Equation 7.7 is valid, and it is called the *Gaussian kernel*. 59:36 The most widely used kernel is in fact the *linear kernel*, $K(x,z) = x^\top z$ with $\phi(x) = x$ — the SVM with no feature map at all — and the Gaussian is next. 1:00:11 Its feature space is infinite-dimensional, and Ng describes it as containing every monomial, of every degree, with the high-degree ones given smaller weight. That can be made exact in one dimension:

$$e^{-\frac{(x-z)^2}{2\sigma^2}} = e^{-\frac{x^2}{2\sigma^2}} e^{-\frac{z^2}{2\sigma^2}} \sum_{k=0}^{\infty} \frac{1}{k!} \left(\frac{xz}{\sigma^2}\right)^k = \sum_{k=0}^{\infty} \phi_k(x)\phi_k(z), \qquad \phi_k(x) = e^{-\frac{x^2}{2\sigma^2}} \frac{x^k}{\sigma^k \sqrt{k!}}.$$

The weight on the degree-k feature falls like $1/\sqrt{k!}$, which is the "smaller weighting" in concrete form. The generator truncates the sum: with $\sigma = 1$, on sixteen pairs of inputs drawn from $[-1.5, 1.5]$, three features reproduce the kernel to within 0.26, twelve to within 3×10^{-6}, and twenty to rounding error.

Figure 7.2 shows the Gaussian kernel at work on data no polynomial of low degree separates cleanly: a disc of one class inside a ring of the other. The decision function $f(x) = \sum_i \alpha_i y^{(i)} K(x^{(i)}, x) + b$ is a sum of bumps centred on the support vectors, 22 of the 110 examples, and its zero level set is the closed curve between the classes.

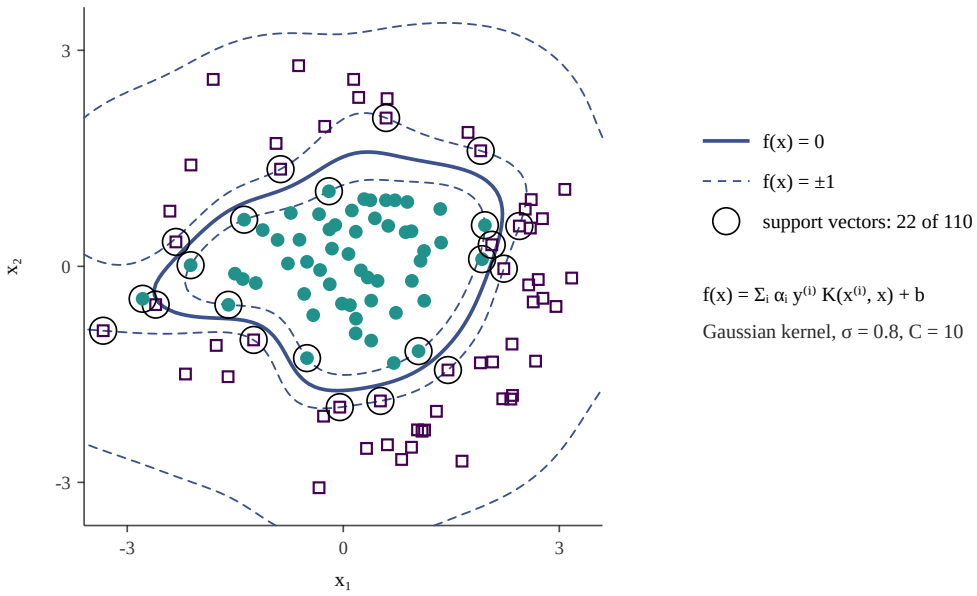


Figure 7.2.: A Gaussian-kernel SVM ($\sigma = 0.8$, $C = 10$) on a disc of positives inside a ring of negatives. The solid curve is $f(x) = 0$, the dashed curves $f(x) = \pm 1$, and the circled points are the support vectors. Far from every training example $f(x)$ falls back to b, which is why a dashed curve also appears near the corners.

7.8. Kernelising other algorithms

1:01:20 The kernel trick is not specific to the SVM. Any algorithm that can be written in terms of inner products of inputs can be kernelised, and Ng lists linear regression, logistic regression, every generalised linear model, the perceptron (which the problem sets kernelise) and, later in the course, principal components analysis. Linear regression in an infinite-dimensional feature space is a perfectly well-defined computation. 1:03:07 In practice, though, the trick is applied overwhelmingly to SVMs and much less often elsewhere.

1:01:26 Ng credits the popularisation to Vladimir Vapnik and Corinna Cortes. The history is slightly split between two papers. The kernelised maximum-margin classifier was introduced by Boser, Guyon and Vapnik in 1992; Cortes and Vapnik's "Support-vector networks" of 1995 added the soft margin of the next section.

7.9. The soft margin

1:04:07 Mapping to a high-dimensional space makes data much more separable, but a separating boundary is not always wanted. If the data are slightly noisy, a boundary that classifies every training example correctly may be contorting itself around a few mislabelled ones. 1:05:17 The *ℓ_1-norm soft margin SVM* relaxes the constraint that every functional margin be at least 1, allowing a slack $\xi_i \geq 0$ per example and charging for it:

$$\min_{w,b,\xi} \ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \quad \text{subject to} \quad y^{(i)}\left(w^\top x^{(i)} + b\right) \geq 1 - \xi_i, \quad \xi_i \geq 0. \tag{7.8}$$

1:06:49 Recall that a positive functional margin means a correct prediction; the hard-margin SVM demands more, a functional margin of at least 1. A slack of $\xi_i = 0.5$ lets example i get away with functional margin 0.5; a slack above 1 lets it be misclassified. At the optimum each slack is as small as its constraint allows, $\xi_i = \max(0,\, 1 - y^{(i)}(w^\top x^{(i)} + b))$, so Equation 7.8 is the same as minimising $\frac{1}{2}\|w\|^2$ plus C times the *hinge loss* summed over the training set — a form worth recognising, since it puts the SVM beside logistic regression as a regularised loss. 1:07:55 The examples on the margin have functional margin exactly 1, Ng notes, because otherwise the line could be moved to widen the margin; examples further away have larger ones.

1:08:59 His second reason for the soft margin is robustness. The hard-margin classifier maximises the *worst-case* margin, so a single outlier — technically leaving the data separable — can swing the boundary dramatically. Figure 7.3 measures this on lecture 6's data with one positive example added near the negatives. The hard-margin normal turns by 43° to squeeze past the outlier and the margin shrinks from 0.77 to 0.30. With $C = 0.1$, adding the same outlier turns the soft-margin boundary by under 4°; the outlier simply sits on the wrong side, with slack $\xi = 1.21$.

How much protection the soft margin gives depends on C, which is the part to take away from the figure. With $C = 1$ on the same data the outlier still turns the boundary by 27°: the slack is so expensive that the solution prefers to move. C trades the width of the margin against the training errors it will tolerate, and 1:12:58 choosing it is a bias–variance question that lecture 8 takes up.

1:11:34 Repeating the dual derivation for Equation 7.8 changes exactly one thing: the constraint $\alpha_i \geq 0$ becomes the box $0 \leq \alpha_i \leq C$. The generator checks the consequences on its solution, which are the optimality conditions in the notes: examples with $0 < \alpha_i < C$ sit exactly on the

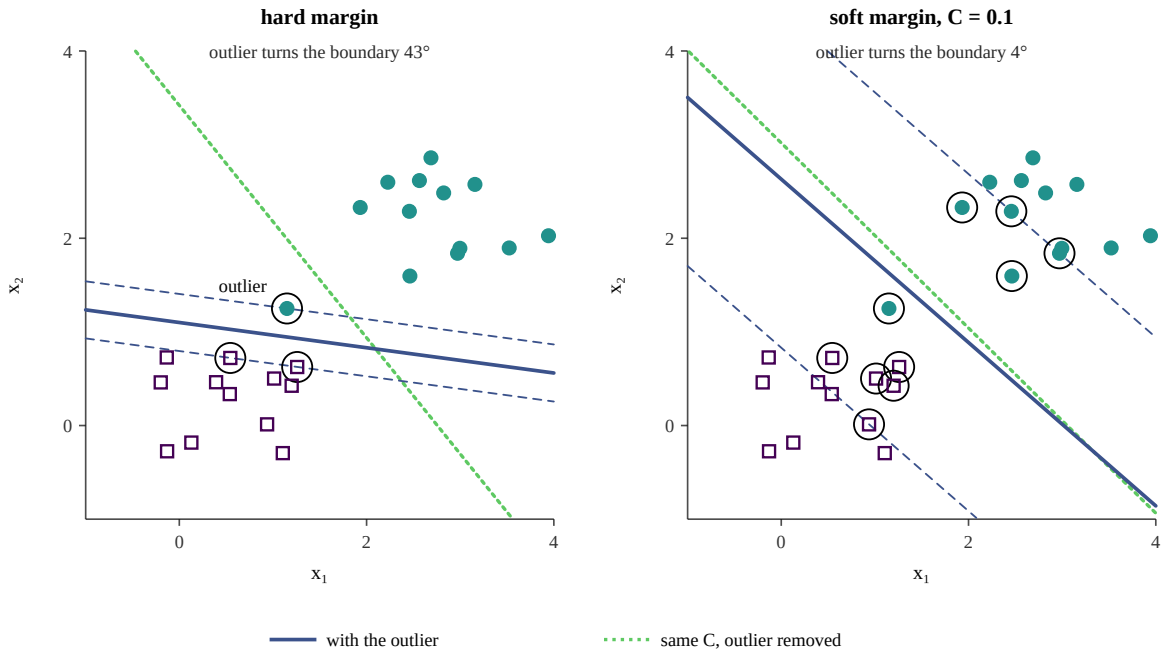


Figure 7.3.: The same data with one positive outlier near the negatives. Left, the hard-margin SVM turns its boundary by 43° to separate the outlier. Right, a soft margin with $C = 0.1$ barely moves: the dotted line is the solution with the same C and the outlier removed.

margin, every example with positive slack has $\alpha_i = C$, and examples beyond the margin have $\alpha_i = 0$. 1:12:19 Ng's practical remark is that in the early days of SVMs solving this problem was itself a research worry, much as matrix inversion once was; today's packages solve it and you need not think about it. The generator's solver is a version of sequential minimal optimisation, the coordinate method of the official notes, which updates two α_i at a time so that $\sum_i \alpha_i y^{(i)} = 0$ stays satisfied.

> **In the official notes**
>
> cs229-notes3, section 8, derives the dual of the soft-margin problem and its KKT conditions, and section 9 presents the SMO algorithm, including the closed-form update for a pair of α_i under the box constraint.

7.10. Kernels for inputs that are not vectors

1:13:39 In practice the polynomial kernel $(x^\top z)^d$ works well, and Ng now calls the Gaussian kernel the one most used; earlier he gave that place to the linear kernel, and between them the two are the usual first choices. 1:14:05 One of the early proof points for SVMs was handwritten digit recognition: unroll an image into a vector of pixel intensities and train an SVM on it. On the classic MNIST benchmark this was, in Ng's words, a very surprising result, and polynomial-kernel SVMs appear beside the convolutional networks of the time in the comparison of classifiers by LeCun et al. 1:15:07 Ng's summary is that SVMs were for a time the best algorithm on it, with convolutional networks pulling ahead in more recent years.

1:15:25 Much of the inventive work on SVMs, Ng says, has gone into designing kernels, and his

example is a protein classifier. A protein is a variable-length sequence of amino acids, of which there are 20 (Ng uses the letters A to Z on the board for convenience). 1:17:01 One feature vector for a sequence counts every possible run of four amino acids — how many times AAAA occurs, then AAAB, and so on to the end of the list — so that $\phi(x)$ has $20^4 = 160{,}000$ entries (with Ng's 26 letters it would be 456,976). 1:18:20 Writing it out is expensive, but $K(x, z) = \phi(x)^\top \phi(z)$ can be computed directly from the two sequences. Ng describes a dynamic programme reminiscent of Knuth–Morris–Pratt string matching; the kernel as published, the spectrum kernel of Leslie, Eskin and Noble, builds a suffix tree and computes it in time linear in the two sequences' lengths. Only the four-letter runs that actually occur contribute, so a dictionary of counts per sequence is already enough, and the generator confirms that this gives the same number as the explicit 160,000-dimensional inner product.

1:19:35 If the inputs are histograms — the demographic make-up of two countries, say — one kernel sums the bin-by-bin minimum of the two histograms, $K(h, g) = \sum_k \min(h_k, g_k)$, a direct measure of their overlap. It is a valid kernel on non-negative histograms, and the proof is one line: $\min(a, b) = \int_0^\infty \mathbb{1}\{t < a\}\,\mathbb{1}\{t < b\}\,dt$, an inner product of two indicator functions. The generator's kernel matrix on twenty random histograms has, as it must, no negative eigenvalue. The general lesson is the recipe of the kernel trick read backwards: for an unusual kind of input, ask what similarity you would want between two examples, then make sure it is an inner product in some space, and the SVM machinery applies unchanged.

Part II.

Generalization

8. Data Splits, Models and Cross-Validation

Lecture 8

Lecture video: Lecture 8: Data Splits, Models and Cross-Validation · 1:23:26

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Every learning algorithm so far has come with a knob — the degree of a polynomial, the bandwidth τ of locally weighted regression, the constant C of the soft-margin SVM — and this lecture is about how to set it. The answer has two parts. The first is a diagnosis: a model that is too simple *underfits*, which Ng calls high bias, and a model that is too flexible *overfits*, which he calls high variance. The second is a procedure: never judge the knob by training error, because training error always prefers the most flexible model; judge it on data the model was not fitted to, held out once (a dev set) or in rotation (cross-validation). Between the two sits regularisation, the one-line change Ng says he uses more often than almost any other technique, which lets a large model behave like a small one.

Ng opens by saying that bias and variance is "easy to learn, hard to master": former PhD students of his report that their understanding of it kept deepening for years after the course. The formal learning theory that used to accompany it — error decomposition, uniform convergence, VC dimension — has moved to the Friday discussion section, which has its own chapter. This lecture keeps the working concepts.

8.1. Bias and variance

4:53 Ng draws the same six houses three times, size against price, with prices that rise and then level off. A straight line, $h_\theta(x) = \theta_0 + \theta_1 x$, catches the trend but not the bend. A quadratic follows the data well. A fifth-order polynomial has six parameters for six points, so it passes through every one of them exactly — and is plainly a worse model of house prices than the quadratic, wiggling between the points and swinging off wildly beyond the outermost ones.

7:04 The line *underfits*, and Ng says it has **high bias**. The word has nothing to do with bias in the social sense; it means that the algorithm arrived with a strong preconception — that price is linear in size — and that the preconception is false, so no amount of data will talk it out of the wrong shape. 8:04 The fifth-order polynomial *overfits*, and has **high variance**. The name comes from a thought experiment Ng returns to several times: suppose a friend collects their own six houses, from the same market, with slightly different noise. The quadratic fitted to the friend's data will look much like yours. The fifth-order polynomial will be some completely different curve, because it is fitting the noise, and the noise is what changed.

That thought experiment is easy to run, and Figure 8.2 runs it twelve times. The straight lines agree with one another and are all wrong in the same way; that shared error is the bias. The quintics agree only where the data pin them down and scatter everywhere else; that spread is the variance. Averaged over 4,000 friends, the linear fit at the middle of the range misses the

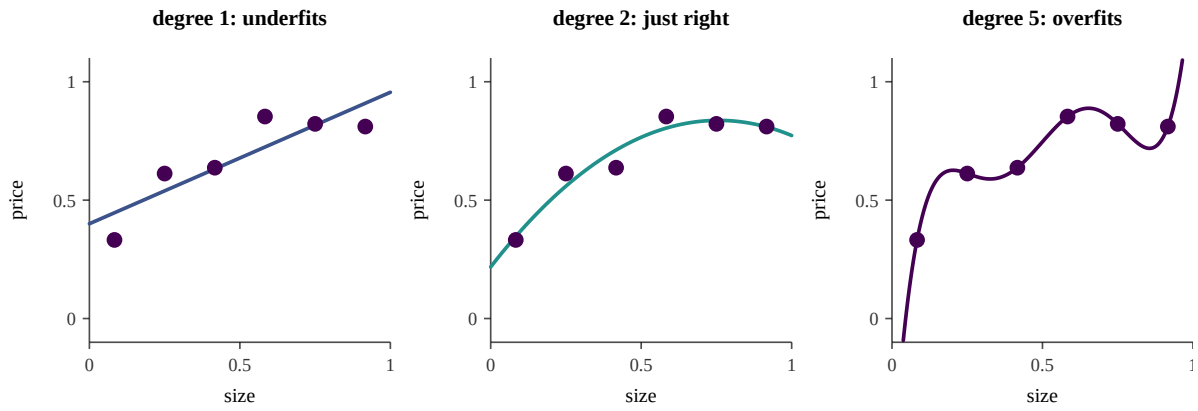


Figure 8.1.: The same six houses fitted three ways. The line misses the curvature; the quadratic is "just right"; the fifth-degree polynomial interpolates every point and makes absurd predictions between and beyond them. The data are synthetic: a concave quadratic plus Gaussian noise.

true price by -0.081 (it sits below the bend) with a standard deviation across datasets of only 0.025; the quadratic is unbiased to three decimals with standard deviation 0.037. Near the small end of the range, at $x = 0.02$, the quintic is unbiased too, but its standard deviation is 0.35 — five times the quadratic's 0.070 and larger than most house prices on the plot.

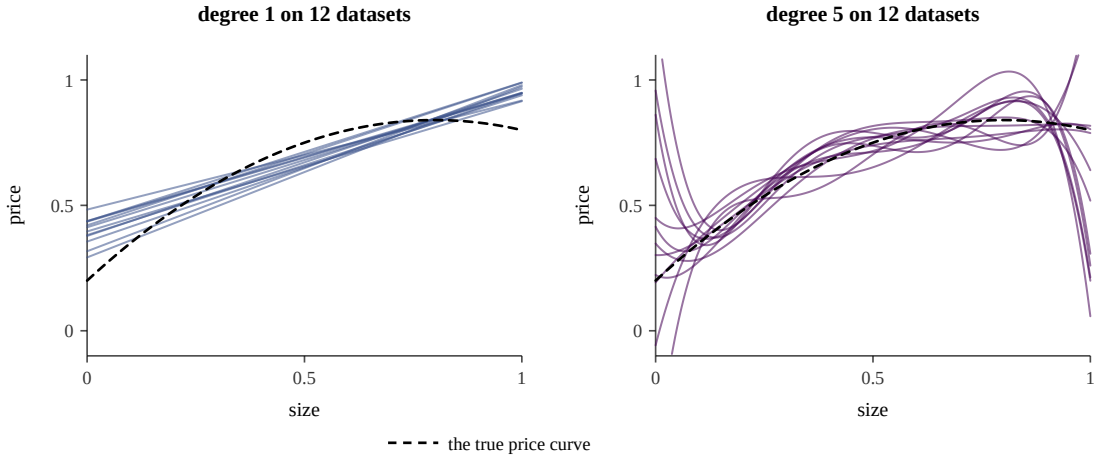


Figure 8.2.: Twelve friends' datasets, each of six houses drawn from the same curve (dashed). Degree-1 fits (left) cluster tightly around the wrong answer: high bias, low variance. Degree-5 fits (right) are right on average but disagree wildly, worst beyond the outermost houses, where several leave the plotted range: low bias, high variance.

10:06 Classification has the same three regimes. Logistic regression on the raw features x_1, x_2 draws a straight boundary through data that clearly want a curve; with a handful of hand-chosen features such as x_1^2, x_2^2, $x_1 x_2$ it draws a sensible curve; with a great many features, or an SVM with a rich kernel, it can draw a contorted boundary that classifies every training example correctly and generalises poorly.

9:30 The reason this matters is practical. Ng's workflow is to train something quick and dirty first, because a learning algorithm "almost never works the first time", and then to ask whether it is suffering from bias or from variance, since the remedies for the two are different and often

opposite. The diagnostics come in lecture 12. Here the lecture turns to the most common remedy for variance.

27:40 A student asks whether "underfit" and "high bias", "overfit" and "high variance", are just synonyms. To a first approximation, Ng says, yes. The exception he flags is a model that has both problems at once: very complex, and still not fitting the data well — a contorted boundary that misses the obvious structure. It is rarer, but it happens.

8.2. Regularisation

13:56 Take the least-squares objective and add one term:

$$\min_{\theta} \; \frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^{\top} x^{(i)} \right)^2 \; + \; \frac{\lambda}{2} \, \|\theta\|^2 . \tag{8.1}$$

The second term is the **regularisation** term, and $\lambda \geq 0$ sets its strength; whether it is written λ or $\lambda/2$ is a matter of which derivations one wants to come out tidy. It gives the algorithm an incentive to keep the parameters small. That is all there is to implement, and Ng warns against underestimating it on that account: it is "deceptively simple" and one of the techniques he reaches for most.

15:17 Put the fifth-order polynomial back on the six houses. With $\lambda = 0$ we have the interpolating curve of Figure 8.1, whose wiggles need large coefficients of opposite sign. A moderate λ makes those coefficients expensive, and the fit relaxes into something like the quadratic — in Figure 8.3, $\lambda = 0.1$, which is what leave-one-out cross-validation (Section 8.5) picks for these six points. 16:47 At the other extreme, $\lambda = 10^{100}$, every parameter is crushed to zero and the hypothesis becomes $h(x) = 0$. Sweeping λ from huge to zero therefore interpolates smoothly between a function that is far too simple and one that is far too complex, and somewhere in between is the one we want.

One detail of the board version deserves a correction in the margin. $h = 0$ is the limit only if the intercept θ_0 is penalised along with everything else, as $\|\theta\|^2$ written literally does. In practice θ_0 is usually left out of the penalty — there is no reason to prefer prices near zero — and then the limit is the constant $h(x) = \bar{y}$, the mean price. The generator checks both limits, and Figure 8.3 draws both. The distinction will reappear in a moment with a probabilistic meaning.

18:15 The same move regularises any model fitted by maximum likelihood. For logistic regression, or any generalised linear model, we were maximising the log-likelihood $\ell(\theta)$, so the penalty is subtracted:

$$\max_{\theta} \; \sum_{i=1}^{m} \log p(y^{(i)} \mid x^{(i)}; \theta) \; - \; \lambda \|\theta\|^2 . \tag{8.2}$$

19:23 Ng then answers a question the kernels lecture left hanging. An SVM with a Gaussian kernel works in an infinite-dimensional feature space; why does it not overfit like crazy? Because its objective is to minimise $\|w\|^2$, which is a regularisation term in all but name. 26:10 Asked later whether the protection comes from the small number of support vectors or from the small norm, he says the formal argument rests on the norm: the class of functions that separate the data with a large margin has low complexity, in the precise sense of a low VC dimension that the discussion section defines. An algorithm could have many support vectors and still enjoy that protection.

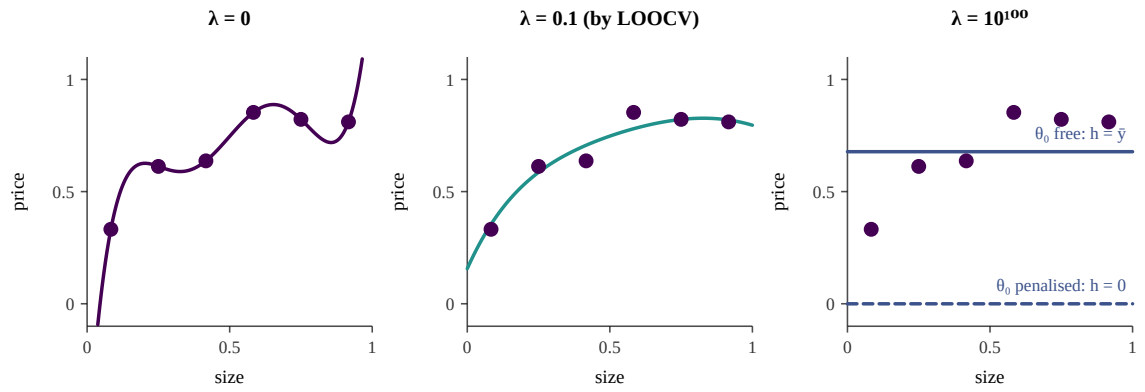


Figure 8.3.: A fifth-degree polynomial on the six houses with three penalties. $\lambda = 0$ interpolates. $\lambda = 0.1$, chosen by leave-one-out cross-validation, behaves like the quadratic. $\lambda = 10^{1}$ flattens everything: to the mean price if the intercept is left out of the penalty (solid), to zero if it is penalised too (dashed).

21:08 His favourite illustration is text classification. Take 100 labelled emails and a 10,000-word vocabulary, one binary feature per word. Unregularised logistic regression has 10,000 parameters and 100 examples, and it will badly overfit; on data like this the training set is almost always linearly separable, and on separable data the likelihood has no maximum: scaling up any separating θ always increases it, so $\|\theta\|$ grows without bound. (On a toy with ten features and four examples, gradient ascent's $\|\theta\|$ is 3.9 after 100 steps, 5.2 after 1,000 and 6.7 after 10,000, and would keep going.) For least squares the same situation shows up as a singular $X^\top X$: with more features than examples it has rank at most m, and the normal equations have no unique solution. Adding λI makes the matrix invertible for any $\lambda > 0$, which is the algebraic face of the same fix. Ng's experience is that regularised logistic regression on such data "will usually outperform" naive Bayes.

22:50 He attaches a rule of thumb. Without regularisation, it is nice to have at least as many examples as parameters, and he is personally only comfortable with roughly ten times as many. With regularisation, 10,000 parameters on 100 examples can make a perfectly decent classifier.

23:36 Why one λ for every parameter rather than a separate λ_j for each, as a student asks? Because then there are 10,000 more numbers to choose, and choosing them is as hard as choosing the parameters in the first place; the selection procedures later in this lecture work for one or a few hyperparameters, not for thousands. 24:41 The student's follow-up is the reason a single λ is reasonable at all: the penalty treats every θ_j alike, so the features must be on comparable scales. Binary word indicators already are. Square feet (hundreds to thousands) and bedrooms (one to five) are not, and the standard preprocessing is to subtract each feature's mean and divide by its standard deviation — which, as a bonus, also speeds up gradient descent.

8.3. Regularisation as a prior on

29:07 Least squares turned out, in lecture 3, to be maximum likelihood under Gaussian noise. Regularised least squares has a matching story, and it requires treating θ itself as random. Given the training set S, Bayes' rule gives the posterior

$$p(\theta \mid S) = \frac{p(S \mid \theta)\, p(\theta)}{p(S)},$$

and since $p(S)$ does not depend on θ, the most probable θ after seeing the data is

$$\theta_{\text{MAP}} = \arg\max_{\theta} \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}, \theta)\; p(\theta). \tag{8.3}$$

This is the **maximum a posteriori** estimate. It is maximum likelihood with one extra factor, the prior. (The semicolon of $p(y \mid x; \theta)$ has become a comma: once θ is a random variable we may condition on it.)

31:12 Now choose the prior $\theta \sim \mathcal{N}(0, \tau^2 I)$. Its log is $-\|\theta\|^2/(2\tau^2)$ plus a constant, so taking logs of Equation 8.3 adds exactly a squared-norm penalty to the log-likelihood. For logistic regression that is Equation 8.2 with $\lambda = 1/(2\tau^2)$. For linear regression with noise variance σ^2, the log-likelihood is $-\frac{1}{2\sigma^2}\sum_i (y^{(i)} - \theta^\top x^{(i)})^2$; multiplying through by $-\sigma^2$ turns the maximisation into Equation 8.1 with

$$\lambda = \frac{\sigma^2}{\tau^2}.$$

The ratio reads sensibly: a tight prior (small τ) or noisy data (large σ) both mean "trust the data less", and both raise λ. The generator confirms both identities by two routes: gradient ascent on the log posterior lands on the ridge solution with $\lambda = \sigma^2/\tau^2$ (for $\sigma = 0.06$, $\tau = 0.5$, that is $\lambda = 0.0144$), and on the penalised logistic solution with $\lambda = 1/(2\tau^2)$, each to within 10^{-15}.

This also explains the intercept detail from Section 8.2. A prior $\mathcal{N}(0, \tau^2 I)$ over the whole of θ says that, before seeing any houses, we believe the price at the average size is probably near zero — which is why the $\lambda \to \infty$ limit of the fully penalised fit is $h = 0$. Leaving θ_0 out of the penalty is the same as giving it a flat prior, an admission that we have no idea of the overall price level, and the limit becomes $\bar{y}$.

32:18 Ng frames this in terms of the two schools of statistics. To a *frequentist*, there is a true θ out in the world that generated the housing prices; it is unknown but not random, and the job is to devise a procedure, such as maximum likelihood, that estimates it well. To a *Bayesian*, θ is unknown and therefore uncertain, and the uncertainty is expressed as a distribution $p(\theta)$ before any data arrive. A Gaussian with mean zero is a reasonable default: we do not know whether each parameter is positive or negative, and we would bet against any of them being enormous. Problem set 2 asks for the equivalence above and for the penalty a Laplace prior produces instead.

36:19 A student asks whether "MLE versus MAP" is the same thing as "unregularised versus regularised". Mechanically yes, Ng says; but frequentists regularise too. They simply present the penalty as part of an estimator they designed, rather than deriving it from a prior. He considers the debate largely philosophical: "I just like my stuff to work."

> **In the official notes**
>
> cs229-notes5, section 3, goes one step past MAP to fully Bayesian prediction, which averages $p(y \mid x, \theta)$ over the whole posterior rather than plugging in its mode, and explains why that integral is usually intractable.

8.4. Choosing a model: train, dev and test sets

37:57 Plot error against model complexity. Training error falls monotonically: a fifth-order polynomial can always fit the training set at least as well as a fourth, because it contains the fourth as a special case. Error on new data falls at first, as the model becomes flexible enough to capture the trend, and then rises, as the extra flexibility goes into fitting noise. 39:42 The same U appears when the knob is λ instead of the degree, with large λ at the simple end. Figure 8.4 computes both curves for ten houses, averaged over 400 datasets: test error bottoms out at degree 2, at 0.0047 against a noise floor of $\sigma^2 = 0.0036$, and climbs to 0.042 at degree 8, while training error keeps falling.

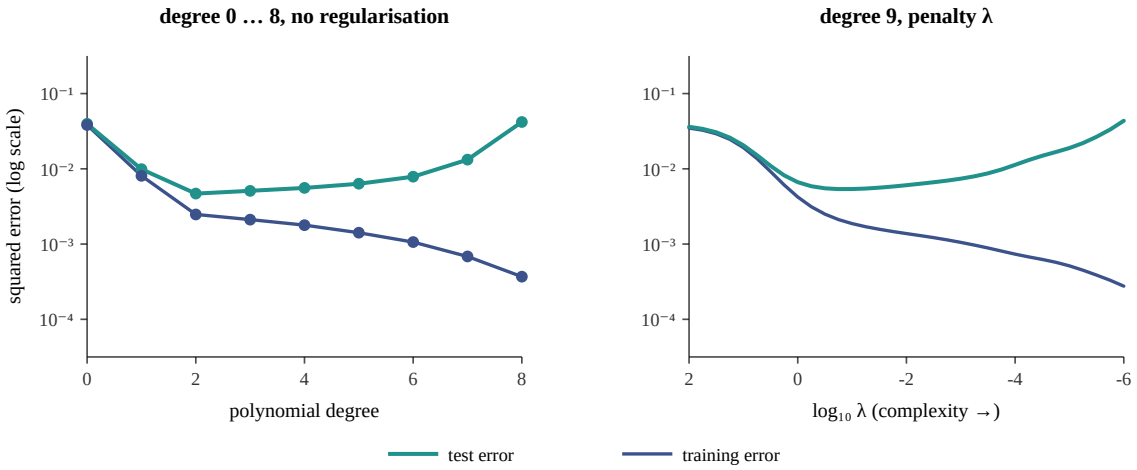


Figure 8.4.: Training error (indigo) and expected test error (teal) for polynomial fits to ten houses, averaged over 400 datasets. Left: against degree, without regularisation; test error is lowest at degree 2. Right: a ninth-degree polynomial as is relaxed from 10^2 to 10 ; test error is lowest near = 0.2. Both axes of error are logarithmic.

41:32 The task is to find the bottom of the U, and it is the same task whether the knob is the degree, λ, the bandwidth τ of locally weighted regression or the C of the soft-margin SVM. 42:47 The procedure is to split the data. Fit each candidate model $M_1, M_2, \ldots$ on one part, S_{train}, to get hypotheses $h_1, h_2, \ldots$; measure each one's error on the other part, S_{dev}; and pick the model with the lowest dev error. "Dev" is short for *development*; the same set is also called the hold-out set or the cross-validation set, and Ng notes that the shorter name has won out mostly because it saves syllables. The procedure is **hold-out cross-validation**.

45:17 The tempting alternative, choosing the model with the lowest *training* error, does not work, and the left panel of Figure 8.4 shows why without further argument: the most complex model always wins on training error. Selecting on the training set would pick the fifth-order polynomial every time.

46:46 Choosing on the dev set has a cost of its own. The chosen model has now been tuned to the dev set — its parameters were fitted on S_{train} and its complexity on S_{dev} — so its dev error is no longer an unbiased estimate of how it will do on new data. Anyone who wants to report a number, in a paper especially, needs a third set, S_{test}, that played no part in any decision.

48:38 A student asks whether dev and test error usually differ by much, and Ng answers with numbers. Suppose polynomials of increasing degree score dev errors of 10, 5.1, 5.0, 4.9, 7 and 10. The winner reports 4.9; but perhaps the middle three are all really 5.0, and the one at 4.9 simply got lucky with the noise. Choosing the minimum of noisy scores systematically selects a lucky

one. How lucky is computable: the expected minimum of k independent standard normals is about -1.27 for $k = 6$ (the generator gets -1.2672 by integration and -1.2665 by simulation), so six models of identical true error, scored with noise of standard deviation 0.1, report on average 4.87 for the winner. The bias grows with the number of models compared.

51:31 The same effect, Ng says, operates on a whole research field when everyone tunes against one public test set for years; he cites a then-recent study of CIFAR as arguing that some of the field's apparent progress on it was overfitting to the benchmark. The study's own conclusion is more specific (Recht et al., 2018). Its authors built a fresh CIFAR-10 test set and found that accuracy dropped by 4 to 10 percentage points across a broad range of models — but that newer models dropped less and kept their ranking, which they read as evidence *against* adaptive overfitting to the old test set; they attribute the drop to the models' sensitivity to small shifts in the data distribution instead. The worry Ng describes is legitimate, but this paper is evidence for a different one.

52:29 When Ng is shipping a product rather than making a claim, he sometimes does without a test set altogether, knowing that the dev number he sees is optimistic. That is acceptable only because nobody outside the team is relying on the number. 1:00:22 The opposite case is also allowed: a team may evaluate on the test set every week and chart the result. What is not allowed is *acting* on it. Rolling back to last week's model because this week's scored lower on the test set is a decision made on the test set, and from that moment it is one more dev set.

53:33 How much data goes into each set? The historical rule of thumb is 70 % train and 30 % test, or 60/20/20 with a dev set, and Ng considers it fine for datasets of a hundred to a few thousand examples. 54:46 For very large datasets the fractions shrink: with 10 million examples, a 20 % dev set is 2 million examples, and the question is whether that many are needed. His criterion is that the dev and test sets should be large enough to detect the size of difference you care about, and no larger; the rest belongs in training. Splits of 90/5/5, or 98/1/1, are common at that scale.

55:40 His example is online advertising, where a team might ship ten projects that each raise the click-through rate by 0.01 %. To tell 90 % from 90.01 % accuracy, he says, a hundred examples cannot do it, and you need "at least 1,000 examples and maybe 10,000 or more". That last figure is the size at which a *single* example is worth 0.01 % — the resolution of the measurement — but resolution is not detectability. The standard error of an accuracy near 90 % measured on m examples is $\sqrt{0.9 \times 0.1/m}$:

Table 8.1.: How the resolution of an accuracy and its sampling noise scale with the size of the dev set.

Dev-set size m	One example is worth	Standard error of a 90 % accuracy
100	1 %	3.0 %
1,000	0.1 %	0.95 %
10,000	0.01 %	0.30 %
1,000,000	0.0001 %	0.03 %

At 10,000 examples the noise is thirty times the difference being sought. Two independently measured accuracies 0.01 % apart are only two standard errors apart at around $4 \times 0.09/10^{-8} \approx 36$ million examples. Comparing both classifiers on the *same* examples helps a great deal, because the noise then comes only from the examples on which they disagree; but even then a 0.01 % difference needs tens of thousands of examples at the very least. Ng's broader point stands and

is, if anything, understated: small improvements need very large evaluation sets. The same table shows that his other example, telling apart algorithms 1 % apart with 1,000 examples, is about one standard error — enough for a hint, not for a verdict.

8.5. k-fold and leave-one-out cross-validation

1:02:10 Hold-out validation is wasteful when data are scarce. Ng's group has worked on medical datasets in which every example is a patient, collected, as he puts it, through "a lot of literal pain". With 100 examples, a 70/30 split trains every candidate on only 70 of them and then judges it on only 30, which is a noisy judge.

1:04:09 **k-fold cross-validation** recycles the data. Split S into k disjoint pieces $S_1, \dots, S_k$ of m/k examples each. For each candidate model, and for each $j = 1, \dots, k$, train on all the pieces except S_j and measure the error on S_j; the model's score is the average of those k errors. Pick the model with the best score. Ng draws $k = 5$ to keep the board legible, but $k = 10$ is by far the most common choice. 1:13:51 The averaging is of the k held-out errors, which are just numbers — a squared error of 5.0 on one fold, 5.7 on the next, 2.8 on the next — not of the models.

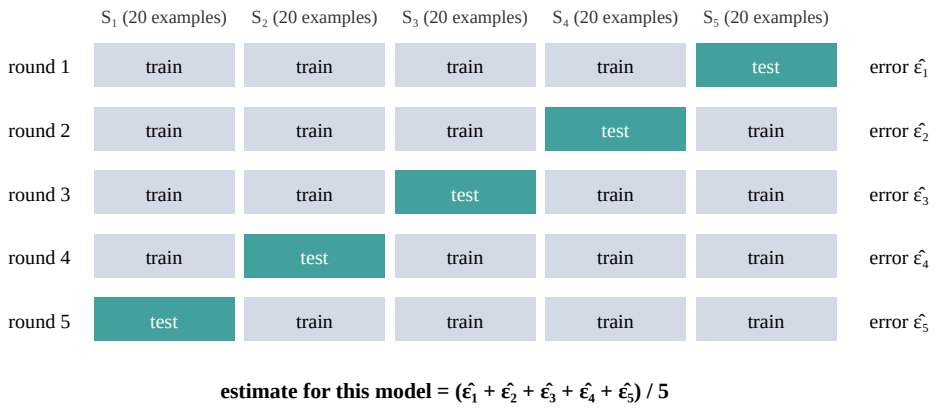


Figure 8.5.: Five-fold cross-validation of one model on 100 examples. Each fold of 20 is held out exactly once; the model is trained on the other 80 each time, and the five held-out errors are averaged into one score. The procedure is repeated for every candidate model.

1:07:41 At the end there are k trained copies of the winning model, each fitted on $(k-1)/k$ of the data. One could keep all of them and average their predictions, but the more common final step is to refit the chosen model once on all of S. With $k = 10$ each round holds out 10 % of the data rather than 30 %, so the models being compared are trained on nearly as much data as the final one; the price is fitting every candidate ten times.

1:09:31 Taking $k = m$ gives **leave-one-out cross-validation**: train on all but one example, test on that one, and repeat for every example. It is the most data-efficient version and the most expensive, m fits per model. Ng uses it only when m is roughly 100 or less, and would switch to k-fold somewhere between 20 and 50 examples. For linear models with a squared-error loss the expense is an illusion, because the leave-one-out residuals can be read off a single fit: if $H = X(X^\top X + \lambda D)^{-1} X^\top$ is the matrix that maps the targets to the fitted values (D is the identity with a zero for the intercept), then the error on example i when it is held out is $(h(x^{(i)}) - y^{(i)})/(1 - H_{ii})$. The generator computes leave-one-out error both ways, by m refits

and by this formula, and they agree to nine significant figures; it is how the $\lambda = 0.1$ of Figure 8.3 was chosen.

1:11:38 A student asks whether the spread of the ten fold errors gives a variance for the cross-validation estimate. It does not, quite, because the ten errors are correlated: any two of the ten training sets share eight-ninths of their examples. Ng recalls research by Michael Kearns on how correlated they are and says the theory does not settle it. The cleaner result came later: Bengio and Grandvalet (2004) proved that *no* unbiased estimator of the variance of k-fold cross-validation exists. The naive estimate errs low. On forty houses with a quadratic and $k = 10$, the actual variance of the cross-validation estimate across 1,500 datasets is 1.21 times the average of what the folds' own spread suggests.

1:12:46 Cross-validation is rare in deep learning, Ng adds, because training a network ten times is expensive, and with only twenty examples a network needs other help anyway, such as transfer learning. 1:15:32 And how should examples be assigned to the sets? Shuffle at random, provided all the data come from one distribution. When training and deployment data differ, as when speech recorded on phones is used to build a smart speaker, splitting becomes more delicate; Ng defers that case to his book *Machine Learning Yearning* and to CS230.

8.6. Feature selection

1:17:00 Feature selection is model selection where the models are subsets of features. It makes sense when there are many features and one suspects only a few matter: of 10,000 word indicators, stop words such as "the" and "of" say nothing about whether an email is spam; of the hundreds of sensor readings on a truck, perhaps a handful predict a breakdown. It does not make sense for images, where every pixel carries a little information. Ng stresses that deciding which regime one is in takes judgement.

With n features there are 2^n subsets, far too many to try, so the search is greedy. 1:19:09 **Forward search** starts from the empty set $\mathcal{F} = \emptyset$ — the model $h(x) = \theta_0$, which predicts the average y — and repeats one step: for every feature not yet in $\mathcal{F}$, fit the model on $\mathcal{F}$ plus that feature and evaluate it on the dev set (or by cross-validation); then add the feature whose addition helped most. 1:20:18 With five features, the first round fits five one-feature models; if x_2 wins, the second round fits four models of the form $\{x_2, x_j\}$; and so on, stopping when adding a feature no longer helps, or running to the end and returning the best subset seen. *Backward search* runs the other way, starting from all features and removing one at a time.

The generator runs forward search on five features of which only x_2 and x_4 matter. It adds x_2, then x_4, and after that dev error stops moving (0.320, 0.319, 0.322, 0.328 as the three irrelevant features are added). Run to the end, forward search makes $n(n+1)/2 + 1$ fits — 16 for five features, the count the generator verifies — which is quadratic rather than exponential in n. That is the sense in which it is affordable, and also the sense in which Ng calls it expensive: for 10,000 features it is fifty million fits.

> **In the official notes**
>
> cs229-notes5, section 2, also covers *filter* feature selection, which scores each feature on its own — for instance by its mutual information with y — and keeps the top k, choosing k by cross-validation. It is much cheaper than forward search and is the usual choice for naive Bayes on text.

Learning Theory

Lecture video: Discussion Section: Learning Theory · 1:26:03

Based on the Friday discussion section of CS229, Stanford University, Autumn 2018, taught by a member of the course staff.

Lecture 8 told you to choose models on held-out data because training error is optimistic. This section asks how optimistic, and answers with a theorem: if an algorithm picks the hypothesis with the lowest training error from a class of k hypotheses, then with probability at least $1-\delta$ its true error is within

$$2\sqrt{\frac{1}{2m}\log\frac{2k}{\delta}}$$

of the best the class can do. Two facts are hiding in that expression. The number of examples needed grows only with the *logarithm* of the size of the class, and, once the class is infinite, the logarithm is replaced by a measure of the class's expressiveness called the VC dimension. Getting there takes two elementary tools — the union bound and Hoeffding's inequality — and a change of viewpoint about what a learning algorithm is.

Ng explained in lecture 8 that this material used to be part of the main lectures and moved to a Friday section because it matters less to making algorithms work than it once did. The section promises to state results rather than prove them, and the proofs are in cs229-notes4. What this chapter adds is the section's own pictures, its answers to questions, and a computation behind every claim: the bounds are checked against exact probabilities and simulations, which also shows how loose they are.

The learning algorithm as an estimator

1:15 Everything rests on two assumptions. There is a fixed data distribution $\mathcal{D}$ over pairs (x, y), from which both the training set and every future test example are drawn; and the examples are drawn independently. Without the first, training data say nothing about test data, and the theory has nothing to work with. Lecture 8's aside about training on phone recordings to deploy on a smart speaker is exactly a case where it fails.

3:15 A learning algorithm is then a box. In goes a training set $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ of m samples from $\mathcal{D}$; out comes a hypothesis $\hat{h}$, or equivalently, for a parametric class such as logistic regression in n dimensions, a parameter vector $\hat{\theta}$. 4:43 The point the section makes first is that S is random, the box is a deterministic function, and so its output is random too. Run the box on a different sample from $\mathcal{D}$ and you get a different $\hat{\theta}$. In statistics the box is called an *estimator*, and the distribution of $\hat{\theta}$ over repeated draws of S is its **sampling distribution**.

6:22 Against this randomness stands a fixed target, a true parameter θ^* (or true hypothesis h^*) that we would like the box to output. It is unknown, but it is not random: there is no distribution over it. That is the frequentist stance of lecture 8, and the notation follows it —

hats for things estimated from data, stars for the true values we never see. Asked whether θ must be a vector, the section answers that it can be anything we estimate, a scalar, a matrix, or a function with no parameters at all.

Bias and variance as moments of the sampling distribution

10:26 Lecture 8 pictured bias and variance in the *data* view: curves through points, underfitting and overfitting. The section redraws them in the *parameter* view, which makes them precise. Take a model with two parameters, so that each $\hat{\theta}$ is a dot in the plane. Draw a training set of size m, run the algorithm, plot the dot; draw another training set of the same size, plot another dot; repeat. The cloud of dots is a sample from the sampling distribution. 13:51 **Bias** asks whether the cloud is centred on θ^*: it is $\mathbb{E}[\hat{\theta}] - \theta^*$, a statement about the first moment. **Variance** asks how spread out the cloud is, the second moment. They are, in the section's words, "just properties of the first and second moments of your sampling distribution".

Figure 8.6 draws the clouds for two real estimators on one regression problem, $y = \theta^{*\top} x + \varepsilon$ with $\theta^* = (1, 0.5)$ and two features correlated at 0.9 — the situation of square footage and number of bedrooms. Least squares is unbiased, and at $m = 10$ its cloud is centred on θ^* but enormous (total variance 1.47), stretched along the direction in which correlated features cannot be told apart: raising θ_1 while lowering θ_2 barely changes the fit. Ridge regression with $\lambda = 10$ on the same training sets is pulled towards the origin, a bias of length 0.52, and its variance is thirty times smaller, 0.045. That is lecture 8's claim that regularisation trades a little bias for a lot of variance, seen directly.

15:58 With ten times the data, every cloud tightens: least squares' variance falls from 1.47 to 0.110, roughly the factor of ten one expects from variance proportional to $1/m$. As $m \to \infty$ the variance goes to zero, and an estimator whose output converges to θ^* is called **consistent**. 17:56 An estimator with $\mathbb{E}[\hat{\theta}] = \theta^*$ at *every* m is called **unbiased**.

17:14 The rate at which variance falls with m the section calls *statistical efficiency*, a measure of how well an algorithm squeezes information from data, and it offers as examples one algorithm whose variance falls like $1/m^2$ and another like e^{-m}. Those rates are not available to an unbiased estimator of a parameter in a regular model: the Cramér–Rao bound puts a floor of order $1/m$ under its variance. (Irregular models are different — the right endpoint of a uniform distribution can be estimated with variance of order $1/m^2$ — but the models in this course are regular.) What efficiency usually measures is the *constant* in front of $1/m$. The mean and the median of a Gaussian sample are both unbiased estimates of its centre, both with variance proportional to $1/m$; but the median's is larger by a factor of $\pi/2$. The generator measures $m \cdot \mathrm{Var}$ at 1.01 for the mean and 1.575 for the median at $m = 101$, against $\pi/2 = 1.571$. Using the median wastes about a third of the data.

20:08 Informally, the section says, a high-bias algorithm is one that "no matter how much data or evidence you provide", stays away from θ^*. That runs two ideas together, and the bottom row of Figure 8.6 separates them. Bias is defined at a fixed m, as a student's question prompts the section to confirm at 21:35; staying away forever is *inconsistency*. Ridge with a fixed λ is biased at every m and yet consistent, because the penalty stays the same size while the data term grows in proportion to m: its bias is 0.52 at $m = 10$, 0.18 at 100, 0.030 at 1,000 and 0.002 at 10,000. 58:09 The same point answers a later question that the section leaves open, whether adding regularisation destroys the consistency of maximum likelihood. With a fixed penalty, it does not.

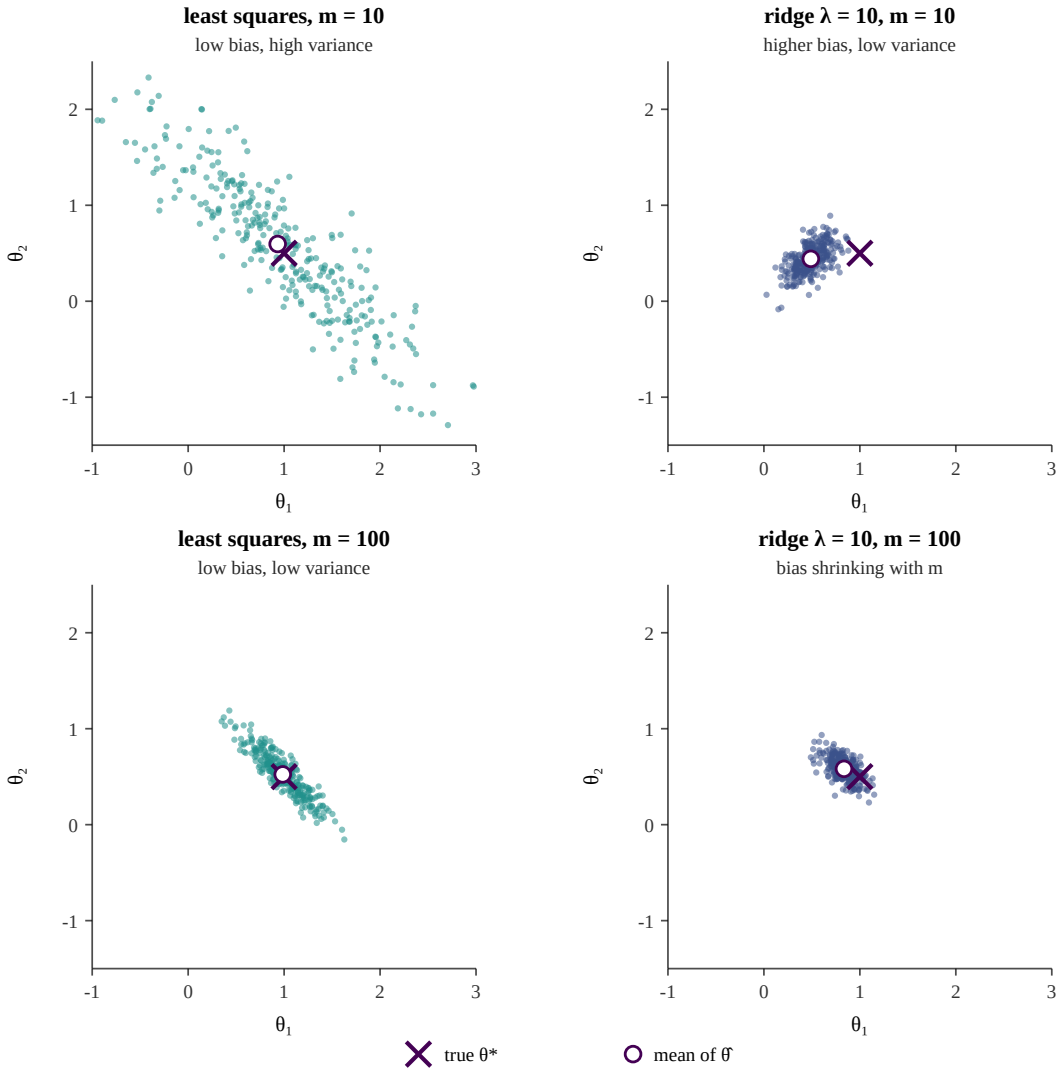


Figure 8.6.: Sampling distributions of ˆ for least squares (teal) and ridge with = 10 (indigo), 300 training sets per panel, on a two-feature regression with correlated features. The cross marks , *the ring the mean of* ˆ. *More data shrinks both clouds, and shrinks ridge's offset from* too.

20:58 Bias and variance are separate properties; any combination of the two can occur. 22:18 High variance has two standard remedies. More data shrinks the cloud directly. Regularisation shrinks it by restricting where $\hat{\theta}$ can go, usually at the price of moving its centre — which is the trade Figure 8.6 shows.

Approximation error and estimation error

25:02 The second picture is drawn in the space of all hypotheses. Somewhere in it is g, the best possible hypothesis, with the lowest expected error under $\mathcal{D}$ of any function whatsoever. 25:52 Our algorithm searches only a class $\mathcal{H}$ — all logistic regression models, say, or all SVMs — drawn as a region that need not contain g. 26:27 Inside the region is h^*, the best hypothesis *in the class*. And what training on finite data actually produces is $\hat{h}$, a different point of $\mathcal{H}$ for every training set.

27:51 For classification the error of a hypothesis is its probability of a mistake, and the training error is its frequency of mistakes on S:

$$\varepsilon(h) = P_{(x,y)\sim\mathcal{D}}\big(h(x) \neq y\big), \qquad \hat{\varepsilon}_S(h) = \frac{1}{m}\sum_{i=1}^{m} 1\{h(x^{(i)}) \neq y^{(i)}\}. \tag{8.4}$$

The first is called the **generalisation error** or risk; the second the **empirical risk**. 30:30 $\varepsilon(g)$ is the **Bayes error**, also called irreducible error: if the same x can come with different labels, no function gets all of them right. Then the error of what we learn splits into three:

$$\varepsilon(\hat{h}) = \underbrace{\varepsilon(g)}_{\text{Bayes error}} + \underbrace{\varepsilon(h^*) - \varepsilon(g)}_{\text{approximation error}} + \underbrace{\varepsilon(\hat{h}) - \varepsilon(h^*)}_{\text{estimation error}}. \tag{8.5}$$

31:35 The identity is a telescoping sum, but each term names a different cause. Approximation error is the price of restricting ourselves to $\mathcal{H}$, a property of the class alone. 32:20 Estimation error is the price of having only m examples, plus any quirks of the algorithm, and it shrinks as m grows.

34:49 The section then says this decomposition and the bias–variance decomposition are related but not the same: estimation error itself splits into a variance part and a bias part, and "bias" in the usual sense is approximation error plus that estimation bias. For 0–1 loss this is a picture rather than an equation — there is no clean additive bias–variance identity for classification error — but for squared error it is exact, and worth seeing. Let $g(x) = \mathbb{E}[y \mid x]$, let $\bar{h} = \mathbb{E}_S[\hat{h}]$ be the average learned hypothesis, and measure error by $\varepsilon(h) = \mathbb{E}(y - h(x))^2$. Then

$$\mathbb{E}_S\,\varepsilon(\hat{h}) = \underbrace{\sigma^2}_{\text{noise}} + \underbrace{\mathbb{E}_x\big(\bar{h}(x) - g(x)\big)^2}_{\text{bias}^2} + \underbrace{\mathbb{E}_{S,x}\big(\hat{h}(x) - \bar{h}(x)\big)^2}_{\text{variance}}, \tag{8.6}$$

and when $\mathcal{H}$ is a linear space, such as all straight lines, $g - h^*$ is orthogonal to it, so Pythagoras splits the bias: $\text{bias}^2 = \text{approximation error} + \mathbb{E}_x(\bar{h} - h^*)^2$, and the estimation error is the variance plus that same small term.

Figure 8.7 does this for straight lines fitted to eight points from a parabola, $g(x) = 4(x - \frac{1}{2})^2$, with noise $\sigma = 0.3$. The best line is flat, $h^* = \frac{1}{3}$, and the approximation error is 0.0889. The expected test error of the fitted line is 0.2597, which splits as $0.0900 + 0.0889 + 0.0808$ one way and $0.0900 + 0.0910 + 0.0786$ the other. The difference between the two middle terms, 0.0021, is the estimation bias: the average fitted line, $0.290 - 0.006x$, is not h^*, because least squares on a

misspecified model is biased at small m. The generator asserts that both totals agree and that the Pythagorean identity holds.

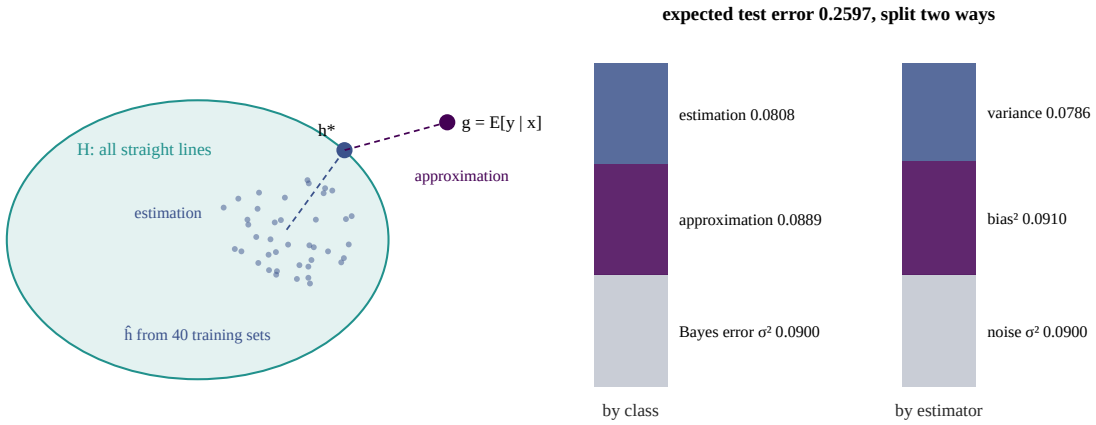


Figure 8.7.: Left: the hypothesis-space picture. The class H is a region; the best predictor g lies outside it; h* is the point of H closest to g; fitted hypotheses ĥ scatter inside H. Right: the expected test error of straight lines fitted to eight noisy points from a parabola, split into Bayes, approximation and estimation error, and into noise, squared bias and variance. The bars are to scale and their totals agree.

36:34 The picture also says how to fight each kind of error. To reduce bias, make $\mathcal{H}$ bigger, so that it reaches closer to g; but a bigger class gives $\hat{h}$ more room to wander, so variance tends to rise. 39:27 To reduce variance, make $\mathcal{H}$ smaller — regularisation does this, by penalising hypotheses with large θ — and unless the shrinking happens to move towards g, bias rises. 41:05 A student asks why not shrink the class towards h^*; because we do not know where h^* is, and if we did there would be nothing to learn.

Empirical risk minimisation and two tools

44:56 Nothing so far depended on what happens inside the box. To prove something, the section now fixes the algorithm: **empirical risk minimisation** returns the hypothesis with the lowest training error,

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\varepsilon}_S(h). \tag{8.7}$$

47:36 This is "what we've been doing so far in the course", with one honest caveat that the section mentions: logistic regression minimises the logistic loss, not the 0–1 training error of Equation 8.4, and SVMs minimise the hinge loss. Those losses are convex stand-ins for the count of mistakes, which is intractable to minimise directly. The theorems below are about the count of mistakes; they extend to the stand-ins with more work, which this section does not do.

49:25 Two questions organise the rest. How close is the training error of a hypothesis to its true error? And how close is the true error of the hypothesis ERM picks to that of the best in the class? 50:47 The first tool is the **union bound**: for any events $A_1, \dots, A_k$, independent or not,

$$P(A_1 \cup \cdots \cup A_k) \le P(A_1) + \cdots + P(A_k).$$

"If this looks trivial, it is trivial," the section says. 52:00 The second is **Hoeffding's inequality**, stated for the case needed here. Let $Z_1, \dots, Z_m$ be independent Bernoulli(ϕ) variables and $\hat{\phi} = \frac{1}{m}\sum_i Z_i$ their mean. Then for any $\gamma > 0$,

$$P(|\hat{\phi} - \phi| > \gamma) \le 2\exp(-2\gamma^2 m). \tag{8.8}$$

The probability that an average of coin flips misses its expectation by more than a margin γ falls exponentially in the number of flips. The general version holds for any bounded independent variables, and the bound on the range is essential.

The bound is easy to check, because the exact probability is a binomial tail. Figure 8.8 compares them for $\gamma = 0.1$. At $m = 100$ the exact probabilities are 0.035 for a fair coin and 0.0034 for $\phi = 0.15$, against a bound of 0.27. The bound holds at every m and ϕ the generator tries, with the exact tail never more than 0.23 of it. It has the right exponential rate, and it is loosest for coins far from fair, whose averages vary less — a looseness that matters here, because error rates of good classifiers are far from one half.

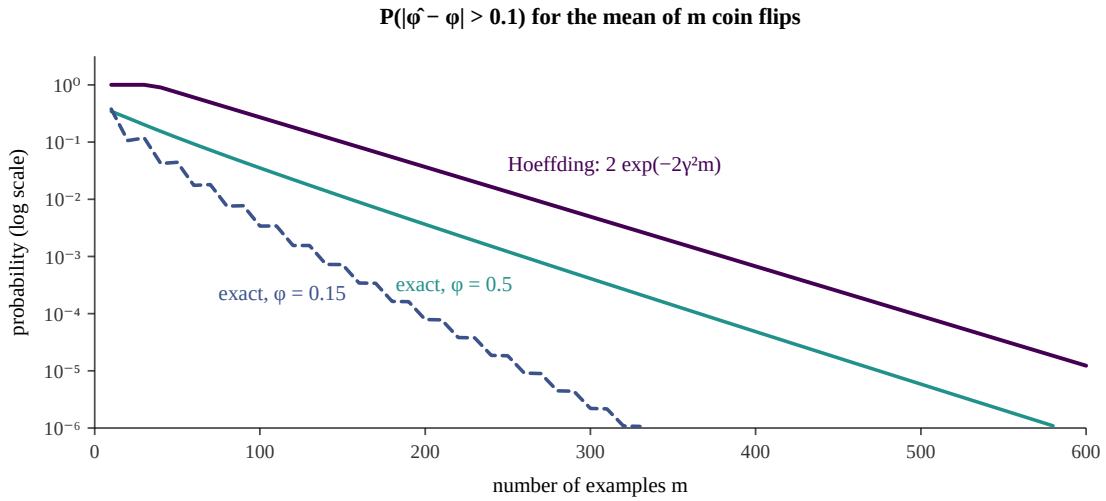


Figure 8.8.: The probability that the mean of m coin flips misses by more than 0.1: computed exactly for = 0.5 and = 0.15, and bounded by Hoeffding's inequality. The bound is valid everywhere and loose by an order of magnitude or more. The = 0.15 curve leaves the plot below 10 .

Uniform convergence for a finite class

59:54 The section's central picture puts every hypothesis in $\mathcal{H}$ on a horizontal axis and plots two curves above it: the true error $\varepsilon(h)$, a fixed curve, and the training error $\hat{\varepsilon}_S(h)$, a random curve that changes with every training set. Figure 8.9 computes it for a class small enough that the true error of every member is a closed form. Inputs are uniform on $[0, 1]$, the label is $1\{x \ge 0.4\}$ flipped with probability 0.15, and $\mathcal{H}$ contains ten threshold classifiers $h_t(x) = 1\{x \ge t\}$ with $t = 0.05, 0.15, \dots, 0.95$. The Bayes error is 0.15. The class has no threshold at 0.4, so its best members, $t = 0.35$ and $t = 0.45$, have error 0.185, an approximation error of 0.035.

1:05:19 For one hypothesis h fixed *in advance*, the training error is an average of m independent Bernoulli variables $1\{h(x^{(i)}) \ne y^{(i)}\}$ whose mean is $\varepsilon(h)$, so Hoeffding applies directly:

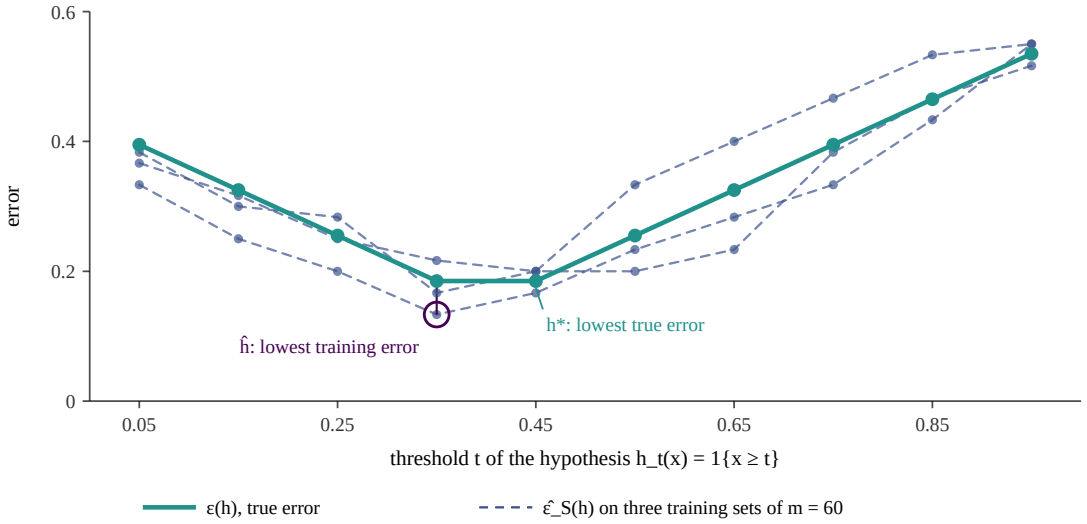


Figure 8.9.: True error (h) (solid) and training error on three samples of sixty examples (dashed) for each of ten threshold classifiers. The training-error curves scatter around the true curve. ERM on the first sample picks the hypothesis circled, whose training error is well below its true error: choosing the minimum selects a hypothesis whose training error happened to come out low.

$$P(|\hat{\varepsilon}_S(h) - \varepsilon(h)| > \gamma) \le 2\exp(-2\gamma^2 m).$$

On average over training sets the dashed curve passes through the solid one at every h, and with more data it hugs it more tightly. 1:08:16 But this is useless as it stands, and the section is precise about why: ERM does not fix h in advance. It looks at the training set and picks the hypothesis whose dashed curve is lowest, so $\hat{h}$ depends on S and its training error is no longer an unbiased average. It is the minimum of noisy scores, which lecture 8 showed is optimistic. The circled point in Figure 8.9 is exactly that.

1:09:32 The fix is to demand that the whole dashed curve be close to the solid one at once — **uniform convergence**. If no hypothesis deviates by more than γ, then neither does the one ERM picks, however it was chosen. 1:11:54 For a class of k hypotheses the union bound gives this cheaply: the event that *some* h deviates is the union of k events, each of probability at most $2\exp(-2\gamma^2 m)$, so

$$P(\exists\, h \in \mathcal{H} : |\hat{\varepsilon}_S(h) - \varepsilon(h)| > \gamma) \le 2k\exp(-2\gamma^2 m). \tag{8.9}$$

Equivalently, with probability at least $1-\delta$, where $\delta = 2k\exp(-2\gamma^2 m)$, every hypothesis has training error within γ of its true error.

The union bound is pessimistic, since it treats the k events as if they could not overlap, and nested thresholds overlap heavily. With $m = 200$ and $\gamma = 0.1$ the generator finds that some hypothesis among the ten deviates in 1.6 % of training sets. The bound for a single hypothesis is already 3.7 %, and the union bound is 37 %. The guarantees in learning theory are worst-case statements over every distribution, and they read best as statements about how quantities *scale*.

1:16:04 Three quantities are tied together — m, γ and δ — and fixing any two gives the third. Fixing γ and δ and solving for m gives the **sample complexity**: with probability at least $1-\delta$, every training error is within γ of the true error, provided

$$m \geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta}. \tag{8.10}$$

The striking feature is the logarithm. For $\gamma = \delta = 0.05$, ten hypotheses need 1,199 examples, a thousand need 2,120, and a million need only 3,501.

How good is the hypothesis ERM picks?

1:17:58 The second question compares $\hat{h}$ with the best in class. On the event that uniform convergence holds, a three-step chain does it, each step either uniform convergence or the definition of ERM:

$$\varepsilon(\hat{h}) \leq \hat{\varepsilon}_S(\hat{h}) + \gamma \leq \hat{\varepsilon}_S(h^*) + \gamma \leq \varepsilon(h^*) + 2\gamma.$$

The first step drops from the solid curve to the dashed one at $\hat{h}$. The second uses the fact that $\hat{h}$ has the lowest training error of any hypothesis, h^* included. The third climbs from the dashed curve back to the solid one at h^*. 1:22:29 Substituting γ from Equation 8.9 gives the theorem: with probability at least $1-\delta$,

$$\varepsilon(\hat{h}) \leq \Big(\min_{h \in \mathcal{H}} \varepsilon(h)\Big) + 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}. \tag{8.11}$$

This is bias and variance once more, in theorem form. The first term is the best the class can do, which can only fall as $\mathcal{H}$ grows; the second, which grows with k, is the price of searching a larger class with the same data. cs229-notes4 makes the same reading. On the threshold class the guarantee holds in every one of 3,000 runs at each m the generator tries, but it is very loose: at $m = 200$ and $\delta = 0.05$, 2γ is 0.245, while ERM picks a best-in-class threshold in every run.

Infinite classes and the VC dimension

1:23:40 Most classes of interest, linear classifiers among them, are infinite, and Equation 8.11 with $k = \infty$ says nothing. The section ran out of time here and only stated the result; cs229-notes4, section 4, gives more, and two pieces of it are worth having.

The first is a crude argument that gets the right answer. A class with d real parameters, stored as 64-bit floating-point numbers, has at most $k = 2^{64d}$ members, and $\log k = 64d \log 2$ is *linear* in d. So the sample complexity of ERM over a d-parameter class grows linearly in d. With $\gamma = \delta = 0.05$ the bound gives about 9,600 examples for $d = 1$ and 89,000 for $d = 10$, which is far too many in absolute terms and right in how it scales.

The second is the principled version. A class $\mathcal{H}$ **shatters** a set of points if, for every one of the 2^n ways of labelling them, some $h \in \mathcal{H}$ gets every label right. The **Vapnik–Chervonenkis dimension** $\mathrm{VC}(\mathcal{H})$ is the size of the largest set $\mathcal{H}$ can shatter. For lines in the plane it is 3, and Figure 8.10 shows why: three points in general position can be split by a line under all eight labellings, while no four points can. Any four points in general position either form a convex

quadrilateral, where the labelling that alternates around it defeats every line, or a triangle with a point inside, where labelling the inner point against the outer three does. The generator checks every labelling of each set by exhaustive search over line directions. The definition needs only *one* shattered set of size 3: three collinear points cannot be shattered, since no line puts the middle point on the other side from its two neighbours, and the VC dimension is still 3.

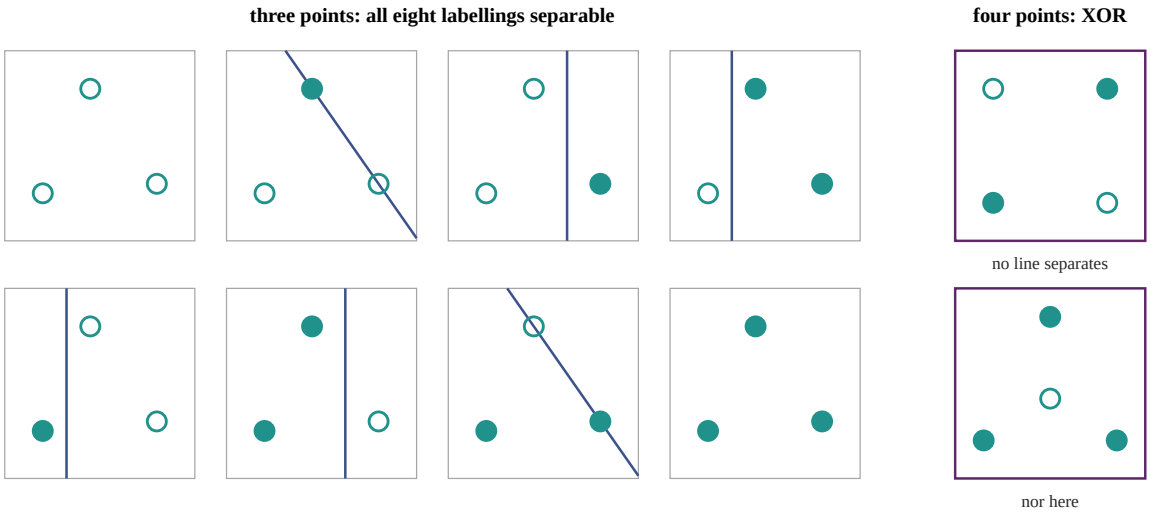


Figure 8.10.: Shattering with lines. Left: three points in general position under all eight labellings, each separated by a line (the all-same labellings need none). Right: the two shapes four points in general position can take, each with a labelling no line separates. The VC dimension of lines in the plane is therefore 3.

1:24:52 Vapnik's theorem then replaces $\log k$ with the VC dimension d: with probability at least $1-\delta$,

$$\varepsilon(\hat{h}) \le \varepsilon(h^*) + O\Big(\sqrt{\frac{d}{m}\log\frac{m}{d} + \frac{1}{m}\log\frac{1}{\delta}}\Big),$$

and the section's take-away is that the number of examples needed is of the order of the VC dimension. For most classes with a reasonable parametrisation the VC dimension is roughly the number of parameters — linear classifiers in n dimensions have $n+1$ — which is the formal version of Ng's rule of thumb in lecture 8 that unregularised models want at least as many examples as parameters.

The VC dimension also answers the question from lecture 8 about why an SVM in an infinite-dimensional feature space does not overfit. The VC dimension of all linear separators in that space is infinite. But Ng's answer was that the relevant class is the separators with a *large margin*, and for data inside a ball of radius R the VC dimension of separators with margin at least γ is bounded in terms of R^2/γ^2, however many dimensions the feature space has. That is the sense in which minimising $\|w\|^2$ is regularisation.

> **i** In the official notes
>
> cs229-notes4, sections 2 and 3, prove the finite-class results stated here, including the sample-complexity corollary; section 4 gives the 64-bit argument, the definition of shattering and Vapnik's theorem. Hoeffding's inequality is proved in the supplementary notes on

> Hoeffding's inequality, via Chernoff bounds and moment generating functions.

Part III.

Trees and Neural Networks

9. Decision Trees and Ensemble Methods

Lecture 9

Lecture video: Lecture 9: Decision Trees and Ensemble Methods · 1:20:41

Based on the lecture by Raphael Townshend, head TA — CS229, Stanford University, Autumn 2018.

A decision tree splits the input space into rectangles by asking one yes-or-no question about one feature at a time, and predicts a constant in each rectangle. It is the course's first genuinely non-linear model, it is easy to explain to anyone, and on its own it is not very accurate: a fully grown tree has low bias and very high variance. The second half of the lecture is about the cure, which is to average many trees. Averaging cuts variance by an amount the correlation between the trees controls, and that single formula explains bagging, the extra randomness of random forests, and why boosting, which works on bias instead, uses deliberately weak trees.

The lecture is given by Raphael Townshend, one of the course's head TAs, while Ng is travelling. It is almost entirely board work, built on one running example.

9.1. A model that cuts the space into boxes

1:24 Townshend, who is Canadian and likes to ski, poses a classifier: given a month and a latitude, can you ski there? Plot month (1 for January to 12 for December) against latitude ($+90°$ at the North Pole, $-90°$ at the South). The positives sit in two separate blocks, the northern winter at high northern latitudes and the southern winter, mid-year, at high southern ones, with nothing near the equator. No line separates that, and while a kernel might lift it into a space where a hyperplane does, the structure is plainly a set of boxes, and it would be simpler to find the boxes directly.

3:38 Finding the best set of regions is intractable in general. A decision tree finds a good one *greedily, top-down and recursively.* It starts with the whole space and asks the single question that best separates the classes — "is the latitude above 30°?" — which cuts the plane in two. It then treats each half as a fresh problem and asks another question there, "is the month before March?", and so on: a game of twenty questions with the data. Figure 9.1 shows the result on synthetic ski data. Seven leaves isolate both skiing seasons.

6:08 Formally, a split of a parent region R_p is a feature j and a threshold t, which divide it into two children:

$$R_1 = \{x \in R_p : x_j < t\}, \qquad R_2 = \{x \in R_p : x_j \geq t\}.$$

Each leaf predicts the most common class among the training examples that land in it. What remains is to say which (j, t) is "best".

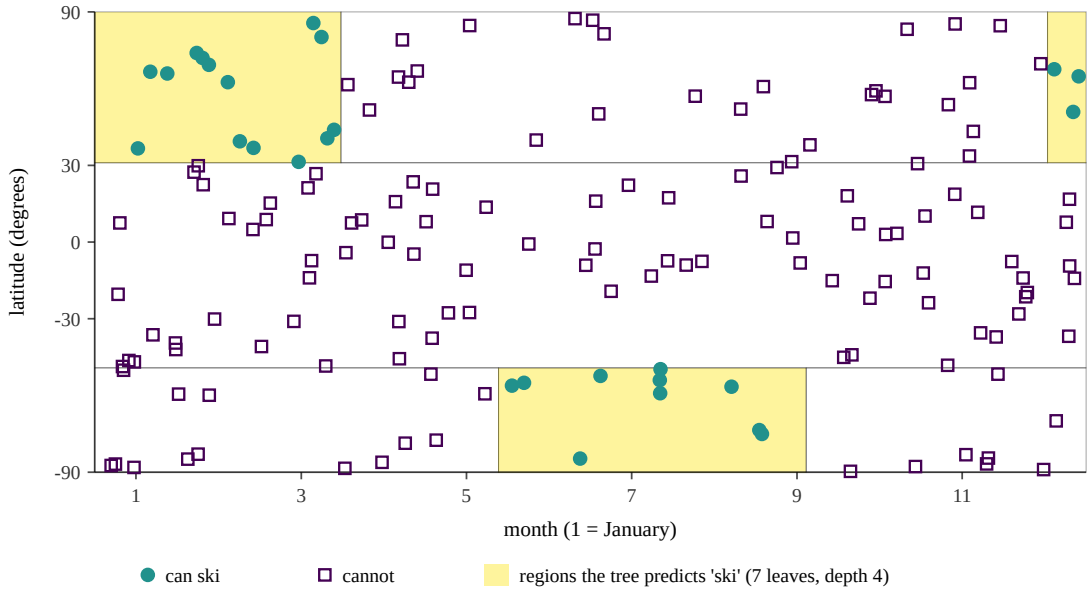


Figure 9.1.: Synthetic ski data: one can ski in the northern winter above 30° and in the southern winter below −30°. A tree grown greedily until its leaves are pure cuts the plane into axis-aligned rectangles; the shaded ones are those it labels “ski”.

9.2. Choosing a split

8:32 Define a loss $L(R)$ for a region and choose the split that reduces it the most, from the loss of the parent to the combined loss of the children. Since the parent's loss is fixed, this is the same as minimising the children's loss. With $\hat{p}_c$ the proportion of the region's examples in class c, the obvious loss is the misclassification rate, $L_{\text{mis}}(R) = 1 - \max_c \hat{p}_c$: predict the majority class and count everything else as an error.

12:29 Townshend's example shows why that loss is the wrong one. A region holds 900 positives and 100 negatives. One split sends 700 positives and 100 negatives one way and 200 positives the other; another sends 400 and 100 one way and 500 positives the other. The second split looks better — it isolates a pure region of 500 — but both children of both splits still predict “positive”, so each split makes exactly the 100 mistakes the parent made. Misclassification registers no gain for either, and a tree built on it would have no reason to prefer the split that obviously helps.

15:21 The fix is a loss that rewards purity even when it does not change any prediction. The standard choice is the **cross-entropy**,

$$L_{\text{cross}}(R) = -\sum_c \hat{p}_c \log_2 \hat{p}_c, \tag{9.1}$$

the number of bits needed, on average, to tell someone the class of an example when they already know the proportions: zero for a pure region, one bit for a 50/50 binary region. 51:56 Asked whether the logarithm must be base 2, Townshend is “pretty sure” it does not matter. It does not: changing the base multiplies every loss and every gain by the same constant, so the ranking of splits cannot change. The generator confirms the ratio is exactly $\ln 2$ for both splits.

One detail on the board needs care. The children's loss must be weighted by their sizes,

$$\frac{|R_1|\,L(R_1) + |R_2|\,L(R_2)}{|R_1| + |R_2|}, \tag{9.2}$$

as cs229-notes-dt writes it. The board writes $L(R_1)+L(R_2)$, which is correct for misclassification measured in *counts*, as in the example, but not for losses defined on *proportions* such as Equation 9.1. Unweighted, the sum can be made larger than the parent's loss by peeling off a single positive and a single negative — a region of two with a full bit of entropy — even though such a split barely changes anything. With the weighting, cross-entropy gives the lecture's two splits gains of 0.034 and 0.108 bits, preferring the second as intuition does.

21:44 The geometry explains why this works, and Figure 9.2 draws it. Plot a binary loss against $\hat{p}$, the share of positives. A parent's $\hat{p}$ is the size-weighted average of its children's, so the parent sits on the curve at that average, while the children's weighted loss sits on the *chord* between the children's points. For a strictly concave curve such as cross-entropy, the chord lies strictly below the curve whenever the children differ, so every split that changes the class proportions gains something. Misclassification is a tent, linear on each side of $\frac{1}{2}$. When both children fall on the same side as the parent, the chord lies on the curve and the gain is exactly zero, which is what happened in the example. Over random splits of random regions the generator finds this case — positive cross-entropy gain, zero misclassification gain — in about half of all splits.

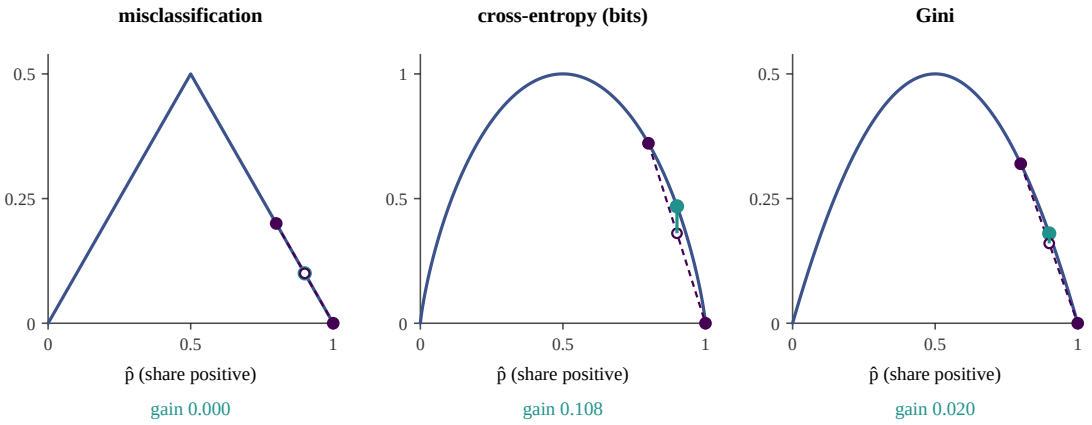


Figure 9.2.: Three impurities against the share of positives. The 900/100 parent sits at 0.9; the 400/100 and 500/0 children sit at 0.8 and 1.0, and their average loss is the midpoint of the dashed chord. The gain is the vertical gap between the parent's point (teal) and the midpoint (ring): zero for misclassification, positive for cross-entropy and Gini.

29:36 The **Gini** impurity $\sum_c \hat{p}_c(1-\hat{p}_c)$ has the same shape and is the other common choice. It is tempting to treat the two as interchangeable, and usually they are; but the generator, offering both the same ten random candidate splits of a 60/40 region 2,000 times, finds that they choose different splits in 5.8 % of cases. The family resemblance is the concavity, and that, Townshend says, is what the losses that work in practice have in common.

9.3. Regression trees and categorical features

31:03 Nothing above depends on the labels being classes. To predict snowfall instead of "can ski", let each leaf R_m predict the mean of its targets, $\hat{y}_m = \frac{1}{|R_m|}\sum_{i\in R_m} y^{(i)}$, and measure the region by squared error, $L_{\text{sq}}(R_m) = \frac{1}{|R_m|}\sum_{i\in R_m}(y^{(i)}-\hat{y}_m)^2$. These are **regression trees**, and splitting

proceeds exactly as before. 34:34 Asked how the best split is actually found, Townshend says it is found by brute force, efficiently: sort the examples along each feature, and every distinct gap between consecutive values is a candidate threshold that can be scored with running sums in one pass.

35:06 Trees also take **categorical** features naturally. If location is one of *north*, *equator* or *south*, a split is a question such as "is the location in the northern hemisphere?" — in general, "is the category in subset A?". 36:27 The catch is the number of subsets. The lecture counts 2^q for q categories; the number of distinct binary splits is $2^{q-1} - 1$, since a subset and its complement define the same split and the empty set defines none. Either way it is exponential. 37:01 For binary classification there is a well-known escape, which the lecture mentions without proof: sort the categories by their proportion of positives, and the best split is always one of the $q - 1$ splits of that sorted list into a prefix and a suffix. The generator checks this against brute force over all subsets for 500 random six-category problems under cross-entropy, and the two always agree.

> **In the official notes**
>
> cs229-notes-dt, sections 2 to 4.1, give the recursive partitioning algorithm, the size-weighted split criterion and the treatment of categorical variables.

9.4. Regularising a tree

37:39 Grown until every leaf is pure, a tree can put each training example in a region of its own. That overfits, and trees are in this sense high-variance models. 38:24 The usual controls are heuristics: stop splitting a node with fewer than a minimum number of examples, cap the depth, or cap the number of nodes.

39:07 A fourth heuristic, stopping when the best split decreases the loss by less than some threshold, is one Townshend calls "very tempting" and advises against. Interactions between features can make the first question worthless on its own and the second one decisive. In the ski data the latitude question alone helps little; the latitude question followed by the month question nails it. The extreme case is XOR: label a point positive when exactly one of its two coordinates exceeds a threshold. The generator builds a balanced XOR set of 64 points, on which *every* single split leaves the cross-entropy exactly where it was, so a tree that splits only on positive gain stops at the root, with 32 mistakes. Taking the zero-gain split anyway, the two children are each solved by one more split, with no mistakes at depth 2.

40:07 The better approach is to grow the tree fully and then **prune** it backwards: remove a split if doing so does not hurt the misclassification error on a held-out validation set — the dev set of lecture 8, used to choose the size of the tree.

9.5. Runtime

42:54 Let n be the number of examples, f the number of features and d the depth of the tree. Prediction walks one path from root to leaf, so it costs $O(d)$, and for a reasonably balanced tree d is about $\log_2 n$. Training is the surprise. 44:46 Each example lands in exactly one node per level, so it takes part in at most d split searches, and in each it is scanned once per feature. The total is $O(nfd)$. Since nf is the size of the data matrix, training a tree costs about as much as

reading the data d times. (For continuous features the scan needs the examples sorted along each feature; the sort is done once up front, or the bound picks up a logarithmic factor.)

9.6. What trees cannot do

47:58 The main weakness is **additive structure**. Label points positive above the line $x_2 = x_1$. Logistic regression represents that boundary exactly with one weight per feature. A tree can only approximate it with a staircase of axis-aligned steps, and every step costs leaves. On 400 training points, a depth-1 tree reaches 75.8 % test accuracy, depth 3 reaches 89.0 %, and a tree grown until its leaves are pure stops at 18 leaves and 96.9 %, the staircase being as fine as the training data allow (Figure 9.3).

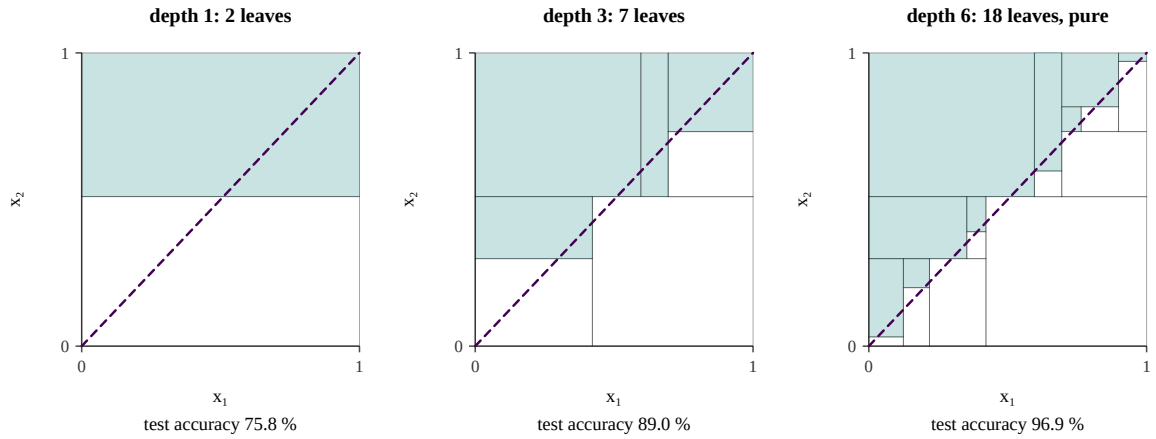


Figure 9.3.: Points above the dashed diagonal are positive. Trees of depth 1 and 3, and one grown until every leaf is pure, each trained on 400 points, shade the region they call positive. The boundary is always a staircase, and following the diagonal more closely takes more leaves and more data.

49:36 The scorecard, then. Trees are easy to explain — twenty questions — and short ones can be read directly. They handle categorical features and they are fast. Against that, they have high variance, they are bad at additive structure, and, mostly for those two reasons, their predictive accuracy is low. 51:09 "I know what you guys are thinking," Townshend says: why spend half a lecture on a model that, on its own, is not very good? Because trees are the ideal raw material for ensembles, which (as of 2018) underpin many of the methods winning Kaggle competitions.

9.7. Why averaging helps

53:11 Let $X_1, \dots, X_n$ be independent random variables with variance σ^2. Their average has variance σ^2/n: each independent source divides the variance further. 55:00 Models trained on related data are not independent, though. If the X_i are identically distributed with pairwise correlation ρ, then

$$\operatorname{Var}\Big(\frac{1}{n}\sum_{i=1}^{n} X_i\Big) = \rho\,\sigma^2 + \frac{1-\rho}{n}\,\sigma^2. \tag{9.3}$$

The second term vanishes as n grows; the first does not. With $\rho = 1$ averaging does nothing at all, and in general the variance can never fall below $\rho\sigma^2$ (Figure 9.4). 57:01 So the game has two moves: average many models to shrink the second term, and make them as decorrelated as possible to shrink the first. The generator checks Equation 9.3 by simulation for $\rho = 0, 0.3, 0.6$.

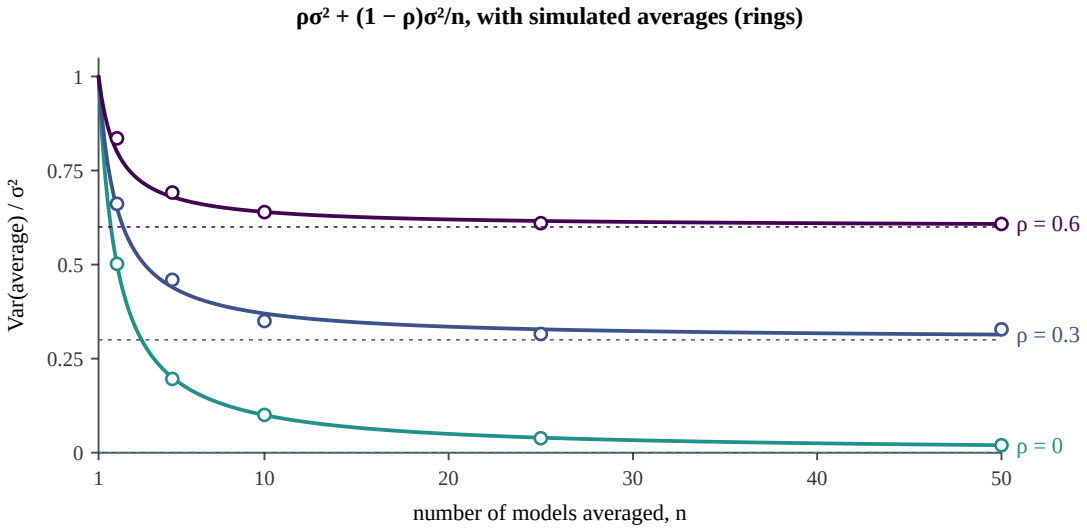


Figure 9.4.: The variance of an average of n equally correlated models, relative to one model's. Independent models (= 0) average away to nothing; correlated ones level off at , the dotted lines. Rings are simulated averages.

57:48 The most decorrelated ensembles come from different algorithms — averaging a neural network, a random forest and an SVM is a common Kaggle tactic that works well — or from different training sets, which is rarely an option, since collecting one was hard enough. Bagging approximates the second; boosting does something else.

9.8. Bagging

1:00:20 **Bagging** stands for *bootstrap aggregation*. The bootstrap is a statistician's device for measuring the uncertainty of an estimate. The training set S was sampled from a population P; ideally we would draw fresh training sets from P, but we cannot. So pretend S *is* the population, and draw new training sets Z from it: n examples sampled from S *with replacement*, which is what sampling from a population means. 1:03:25 Bagging takes M bootstrap samples $Z_1, \dots, Z_M$, trains a model G_m on each, and averages:

$$G(x) = \frac{1}{M} \sum_{m=1}^{M} G_m(x).$$

1:05:06 In the terms of Equation 9.3, the bootstrap lowers ρ (the models see different data) and M controls the second term. A consequence Townshend stresses is that more bootstrap models cannot make the ensemble overfit more: they only shrink the second term, so the usual practice is to keep adding models until the error stops falling. Figure 9.5 shows exactly that.

1:07:33 Asked how far bootstrapping can push ρ down, he has no formula, only the reason there is a floor: every bootstrap sample is drawn from the same S, and each contains about two-thirds of it. The exact fraction of distinct examples is $1 - (1 - 1/n)^n \to 1 - 1/e \approx 0.632$, which the

generator confirms by sampling. 1:08:29 That missing third is also the price: each model trains on fewer distinct examples, so bagging slightly raises bias. In Figure 9.5 a single tree on a bootstrap sample has test error 1.52 against 1.08 for one tree on the full data, but by three or four trees the average has overtaken the full-data tree, and a hundred bagged trees reach 0.75.

1:11:23 This is why trees and bagging belong together. A deep tree has low bias and high variance; bagging trades a little bias for a large cut in variance; and so a model whose error is mostly variance gains the most. Bagging a high-bias model would gain little.

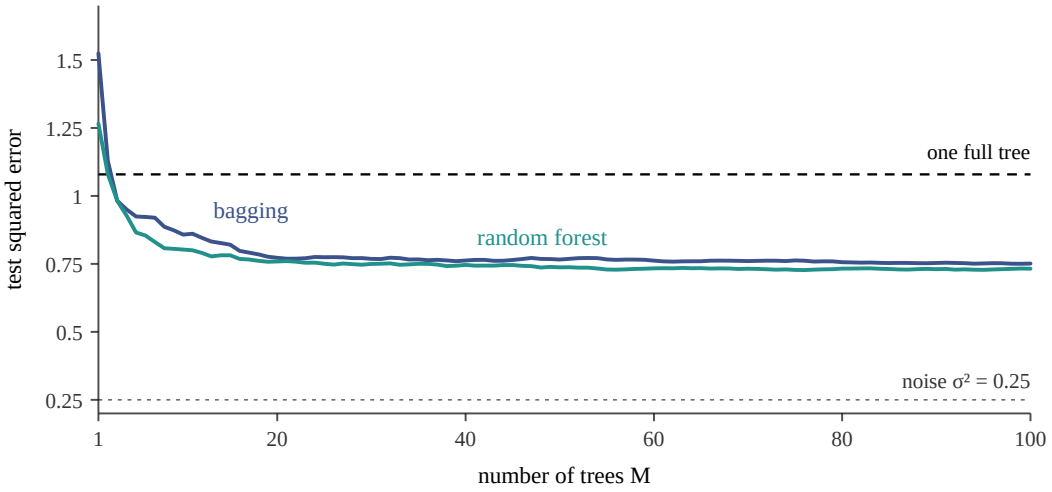


Figure 9.5.: Test error of an average of M regression trees grown on bootstrap samples, for plain bagging and for a random forest that considers three of eight features at each split. Both fall steeply and then flatten; neither rises as trees are added. One full tree on all the data is far worse; the dotted line is the noise floor.

The method is due to Breiman (1996).

9.9. Random forests

1:12:29 A **random forest** is bagged trees with one more source of randomness: at each split, the tree may only consider a random subset of the features. 1:13:56 Townshend gives two reasons this helps, which are the same reason. It lowers ρ. And concretely, if one feature is a very strong predictor, every bagged tree will split on it first whatever its bootstrap sample, so the trees are alike from the root down; forbidding it at some splits forces them apart.

The decorrelation is real and easy to measure. On the problem of Figure 9.5 — eight noisy measurements, five of one latent variable and three of another — the correlation between two bagged trees' predictions across training sets is about 0.18; letting each split see only three features drops it to 0.07, with each tree's own variance roughly unchanged, so $\rho\sigma^2$ falls from about 0.14 to 0.05. The test error of 100 trees falls from 0.738 with all features to about 0.70 with two to four.

But the argument above is only about variance, and the lecture does not mention the other side. Forbidding a feature can also make a tree worse. On a second problem where one feature carries most of the signal and the others are weak, restricting each split to two of six features roughly doubles each tree's variance — many splits must make do with weak features — and

the forest's test error rises from 0.44 to 0.51; with one feature per split it reaches 0.74. Random feature selection pays when the features it forces the trees to use are nearly as good as the ones it hides, as with redundant or correlated measurements. The number of features per split is a hyperparameter to tune like any other, which is how Breiman (2001) treated it.

9.10. Boosting

1:15:16 Bagging fights variance. **Boosting** works in the opposite direction: it reduces *bias*, by building an ensemble additively, each new model trained to fix what the previous ones got wrong. It starts from deliberately weak models — typically **decision stumps**, trees of depth one that ask a single question. A stump has high bias and low variance, which is the right raw material for a method that lowers bias. (The lecture's sentence at 1:16:44 runs these two the wrong way round, saying that restricting to depth one decreases bias before saying it increases it; the second is right.)

1:16:59 The first stump draws its line and misclassifies some examples. Boosting increases the weight of those examples and trains the next stump on the reweighted set, which pushes it to get the hard ones right; and so on, each example's weight reflecting how often the ensemble so far has got it wrong. Figure 9.6 runs it on the diagonal boundary that defeats a single tree. 1:18:17 Each stump G_m also gets a vote α_m, larger for more accurate stumps, and the final classifier is the weighted vote $G(x) = \text{sign}(\sum_m \alpha_m G_m(x))$. The algorithm is **AdaBoost**, due to Freund and Schapire (1997). In the form the official notes give, with labels $y \in \{-1, +1\}$ and weights starting at $w_i = 1/n$, each round:

1. fits a stump G_m to the weighted data;
2. computes its weighted error $\text{err}_m = \sum_i w_i \, 1\{y_i \neq G_m(x_i)\} / \sum_i w_i$;
3. sets its vote to the log-odds $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$;
4. multiplies the weight of every misclassified example by e^{α_m}.

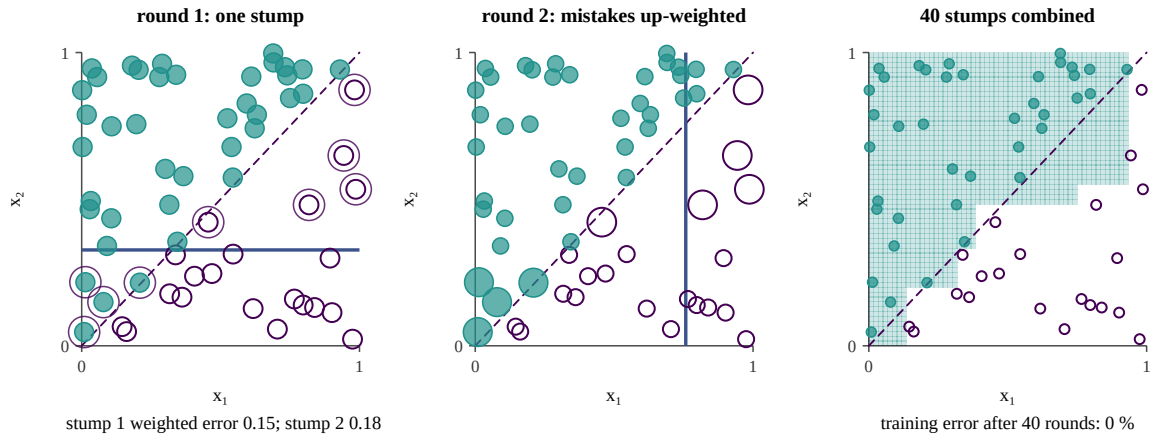


Figure 9.6.: AdaBoost with decision stumps on 60 points labelled by the diagonal. Left: the first stump, with its nine mistakes ringed. Middle: point areas show the weights before round 2, the mistakes now heavy, and the second stump that responds to them. Right: the region the weighted vote of 40 stumps calls positive, an additive staircase that classifies every training point correctly.

The weights look arbitrary, and the notes justify them: AdaBoost is *forward stagewise additive modelling* under the exponential loss $e^{-yf(x)}$, adding one stump at a time and choosing each

stump's coefficient to minimise that loss with the earlier ones fixed. That derivation gives a coefficient of $\frac{1}{2}\log((1-\mathrm{err}_m)/\mathrm{err}_m)$ — half the vote above. The generator runs both versions side by side for 40 rounds: they choose the same stump every round, their coefficients differ by a factor of exactly 2, and since doubling every vote does not change the sign of the sum, they make the same prediction everywhere on a fine grid.

On the diagonal data the first stump has weighted error 0.15 and the second 0.18. Training error is 15 % after two rounds, 5 % after five, 1.7 % after ten, and zero from round twenty, each stump being only a little better than chance — by round 40 the newest stump's weighted error is 0.35. Freund and Schapire's bound guarantees this kind of progress: training error is at most $\prod_m 2\sqrt{\mathrm{err}_m(1-\mathrm{err}_m)}$, which shrinks geometrically as long as every stump beats chance by some margin. The generator asserts the bound at every round; it is 0.068 at round 20 and 0.023 at round 40.

The additive structure that trees lack (Section 9.6) is exactly what boosting adds, since the ensemble is a sum. That cuts both ways, and the notes say so where the lecture does not: because each new model depends on the earlier ones, adding rounds raises the risk of overfitting, unlike adding trees to a bagged ensemble. The number of rounds is chosen on held-out data. 1:20:08 The same stagewise idea with other losses gives gradient boosting, and with engineering on top, XGBoost.

> **In the official notes**
>
> cs229-notes-ensemble, section 1, covers the bootstrap, aggregation and their combination with trees; section 2 gives AdaBoost, states that it is forward stagewise additive modelling under the exponential loss (leaving the derivation to *The Elements of Statistical Learning*, section 10.4), and sketches gradient boosting.

10. Introduction to Neural Networks

Lecture 10

Lecture video: Lecture 10: Introduction to Neural Networks · 1:20:14

Based on the lecture by Kian Katanforoosh — CS229, Stanford University, Autumn 2018.

A neural network is logistic regression repeated and stacked. One neuron computes a linear function of its inputs and passes the result through a non-linearity; a layer is several neurons reading the same inputs; a network is layers feeding one another. This lecture builds that up one step at a time from a classifier the course already has, and at each step asks the same three questions: what the output means, what the loss must therefore be, and how many parameters there are. It ends with the forward pass written in matrices, for one example and for a whole batch, and with the reason gradients are computed from the last layer backwards, which the next chapter works out in full.

The two deep-learning lectures are given by Kian Katanforoosh, who teaches CS230, the Stanford deep-learning course. 1:02 He opens with the usual account of why the methods took off when they did: GPUs made the computation affordable, the digitised world made large datasets available, and these models are flexible enough to keep improving as the data grows, which older methods mostly stop doing.

10.1. Logistic regression as one neuron

3:05 The running task is to say whether an image contains a cat: output a number close to 1 if it does and close to 0 if it does not. A colour image of 64×64 pixels is three such grids, one per red, green and blue channel, so it is $64 \cdot 64 \cdot 3 = 12{,}288$ numbers. Logistic regression needs a vector, so the first step is simply to flatten the image into a column vector $x \in \mathbb{R}^{12288}$, and then

$$\hat{y} = \sigma(wx + b), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}. \tag{10.1}$$

This is the model of Section 3.6 with the intercept split out: $\theta^\top x$ with a constant feature becomes $wx + b$, where b plays the role of θ_0 and w holds the rest. 6:03 Since $\hat{y}$ is a scalar and x is $12{,}288 \times 1$, w must be a *row* vector, $1 \times 12{,}288$. The lecture asks for that shape before anything else, and the habit of checking shapes first carries the whole of the next chapter.

Training is the familiar loop: initialise w and b, minimise the logistic loss

$$L(\hat{y}, y) = -\big[\, y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \,\big] \tag{10.2}$$

by gradient descent, then predict with the fitted parameters. The loss is the negative log-likelihood of the Bernoulli model, so minimising it is maximum likelihood. What Katanforoosh stresses is its role: the function we actually want, the one that recognises cats, has parameters we cannot write down, and the loss is the proxy we can optimise in their place.

10:26 The model has 12,288 weights and one bias. There is a quicker way to count that the lecture uses from here on: draw the model with one edge from each input to the neuron, and there is one weight per edge and one bias per neuron. It also shows something unwelcome, that the parameter count grows with the size of the input. A larger image means a larger w, and the lecture flags this as something to change later; convolutional networks, which share weights across the image, are the change.

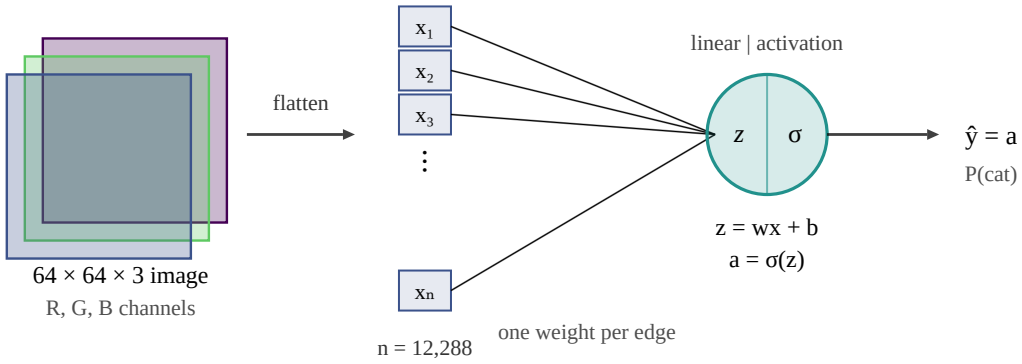


Figure 10.1.: Logistic regression drawn as a neuron. The image is flattened into x, every entry reaches the neuron along its own weighted edge, and the neuron computes a linear part z followed by the activation σ.

11:34 Two slogans come out of this. The first is that *a neuron is a linear part plus an activation*: $z = wx + b$, then $a = g(z)$, with the sigmoid as one choice of g among several. The second is that *a model is an architecture plus parameters*. When a team says it has shipped a model, it has shipped two files, one describing the computation and one holding the numbers that computation uses. The rest of the lecture changes the architecture and lets the parameters, still called W and b, grow.

10.2. Three animals, three neurons

13:42 Now the task is to detect a cat, a lion and an iguana. The obvious extension is to put three neurons side by side, each connected to every pixel, each computing its own logistic regression:

$$\hat{y}_k = a_k^{[1]} = \sigma(w_k^{[1]} x + b_k^{[1]}), \qquad k = 1, 2, 3.$$

The square-bracket superscript names the *layer* and the subscript the neuron within it. This network has one layer, whose neurons do not talk to one another, and its output $\hat{y}$ is now a vector of three numbers. 17:08 Each neuron has its own weights, so the parameter count triples, to $3n + 3 = 36{,}867$ for $n = 12{,}288$.

The labels must change with the output. 18:42 An image of a cat gets $(1, 0, 0)$, a lion $(0, 1, 0)$, an iguana $(0, 0, 1)$, and the loss is the sum of three logistic losses, one per entry. Which neuron ends

up detecting cats is decided by nothing more than this labelling convention: the first neuron is only ever compared with the first entry, so it becomes the cat detector, and had the convention put cats second, the second neuron would have.

20:37 The same observation answers the question of an image showing a cat *and* a lion. Label it $(1, 1, 0)$ and nothing breaks, because the three neurons are trained independently; each sigmoid depends only on its own weights, and nothing forces the outputs to add up to anything. 22:32 As a student points out, the network is not yet a network in any interesting sense: it is three logistic regressions that happen to share an input.

23:49 The case where this design is the wrong one is when the classes exclude each other: a cell image showing exactly one of many diseases, say. The lecture's argument is that a very rare class, the one iguana in a zoo of thousands of lions and cats, leaves its neuron almost untrained. That needs one qualification. The iguana neuron does receive a gradient on every image: the derivative of its logistic loss with respect to its logit is $\sigma(z_3) - y_3$, which on an image without an iguana is $\sigma(z_3) > 0$ and pushes the neuron towards saying "no iguana". What the neuron lacks is positive examples, and a softmax does not create those either. The real reason to prefer a softmax for exclusive classes is that it builds the exclusivity into the model, which is the subject of the next section.

10.3. Softmax: one class per image

25:10 Add the constraint that each image shows exactly one of the three animals. Write the linear part of neuron k as $z_k^{[1]} = w_k^{[1]} x + b_k^{[1]}$, drop the three sigmoids, and replace them by a single function of all three logits:

$$\hat{y}_k = \frac{e^{z_k^{[1]}}}{\sum_{j=1}^{3} e^{z_j^{[1]}}}. \tag{10.3}$$

This is the softmax of softmax regression, derived as a generalised linear model in Section 4.7. 28:27 Its outputs are positive and sum to one, so they form a probability distribution over the classes. It can no longer say "probably a cat and probably a lion": raising one output lowers the others. Labels are one-hot, $(1, 0, 0)$, $(0, 1, 0)$ or $(0, 0, 1)$, and the prediction is the class with the largest output. 30:03 The constraint also bakes in an assumption that one of the three animals is always present; if images with none of them can occur, a fourth class for "no animal" has to be added. The parameter count does not change, since the softmax has no parameters of its own: still $3n + 3$.

33:17 A student asks what happens if this network is trained on the multi-hot label $(1, 1, 0)$ anyway. The lecture's guess is that it will settle near $(\frac{1}{2}, \frac{1}{2}, 0)$, and that is exactly right: with the cross-entropy loss below, the best a distribution can do against two targets of 1 is split its mass equally between them, and the generator, training a softmax on that label by gradient descent, ends at $(0.5000, 0.5000, 0.0000)$. The fit can never be exact, because the outputs sum to 1 and the label sums to 2.

34:07 The loss has to change too. For the three independent sigmoids the derivative of the summed loss with respect to $w_2^{[1]}$ involves only the second term, since the other two do not depend on $w_2^{[1]}$. Under a softmax every output depends on every logit through the denominator, so the loss is the *softmax cross-entropy*

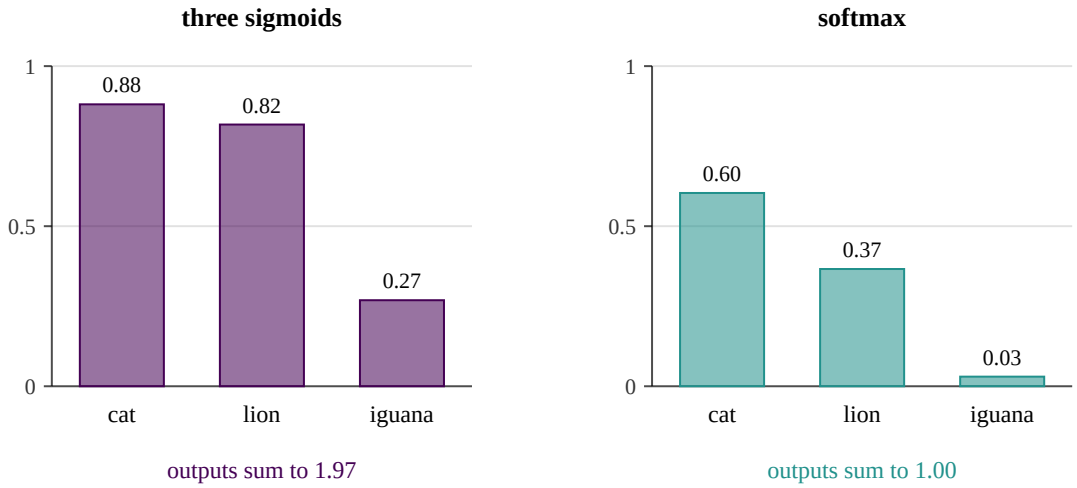


Figure 10.2.: The same logits, $z = (2.0, 1.5, -1.0)$, through three independent sigmoids and through a softmax. The sigmoids can report two animals at once and their outputs need not sum to anything; the softmax shares one unit of probability among the classes.

$$L(\hat{y}, y) = -\sum_{k=1}^{3} y_k \log \hat{y}_k, \tag{10.4}$$

which for a one-hot y is just $-\log$ of the probability given to the right class. The lecture leaves its derivative as an exercise and suggests it is harder than the sigmoid case. It looks harder and is not. The coupling through the denominator cancels, and for a one-hot label the gradient with respect to the logits is

$$\frac{\partial L}{\partial z_k^{[1]}} = \hat{y}_k - y_k, \tag{10.5}$$

the same form as the logistic loss of one sigmoid neuron, $\sigma(z) - y$. The generator checks both against finite differences on 500 random logit vectors. This shared form is not a coincidence: both are canonical-link GLMs (Section 4.4), and it is the reason the next chapter's backpropagation starts from "prediction minus label" at the output whichever of the two is used.

10.4. Predicting a number

37:17 Suppose instead the network has to estimate the cat's age. One answer, from the room, is to treat each age as a class and use a softmax, since a cat has exactly one age. The other is to make it a regression, and that changes two things. The sigmoid at the output has to go, since it confines $\hat{y}$ to $(0, 1)$. With the identity in its place, the one-neuron model is linear regression. 39:29 A better choice here is the *rectified linear unit*,

$$\mathrm{ReLU}(z) = \max(0, z),$$

which is linear for positive inputs and zero for negative ones, a natural fit for a quantity that cannot be negative. The loss changes as well, to the absolute or squared error between $\hat{y}$ and y. The lecture's reason is that these are easier to optimise for regression than the logistic loss would be; the reason from the earlier chapters is that squared error is the log-likelihood of a Gaussian output, just as the logistic loss is the log-likelihood of a Bernoulli one, and a student's suggestion of a Poisson output for counts would pick the matching loss in the same way.

10.5. Hidden layers

41:16 Back to cat or no cat, but with more neurons: a layer of three, feeding a layer of two, feeding one output neuron. 43:00 The only hard constraint on the architecture is at the end, where the output layer must have one neuron per number to be predicted. The parameter count follows from one weight per edge and one bias per neuron:

$$\underbrace{3n+3}_{\text{layer 1}}+\underbrace{2\cdot 3+2}_{\text{layer 2}}+\underbrace{2+1}_{\text{layer 3}}=3n+14=36{,}878.$$

Adding two layers cost eleven parameters; 99.97 % of the network is still the first layer, because that is the layer that touches all 12,288 pixels.

44:31 Some vocabulary. A *layer* is a set of neurons that are not connected to one another. Katanforoosh calls the first layer of neurons the input layer, the last the output layer, and the one between a *hidden* layer; the official notes, and this book, instead treat the input x as layer 0 and count only layers of neurons, which gives the same three. His reason for "hidden" is that the layer sees neither the input nor the label, only what the previous layer hands it. The more common reading is about the training data: it supplies values for the inputs and the outputs and never for these units, so what they compute is not specified by anyone.

45:31 What they do compute, in a network trained on enough cat images, is usually described as a hierarchy: neurons early on respond to edges, later ones combine edges into ears and eyes, and the output combines those into a face. 1:03:48 A student later asks why the first layer should find edges, and the lecture's answer is that those neurons see only pixels, which is too granular to see more. For the network on the board that argument does not hold. Each neuron of a fully connected first layer is a weighted sum of *all* 12,288 pixels, a template the size of the whole image, and nothing prevents that template from looking like a face. The edge story comes from convolutional networks, whose first-layer neurons each look at a small patch of the image; for them a local pattern such as an edge is about all a single neuron can represent, and there the argument is right.

46:44 The second example makes the point about hidden units from the other side. Predict a house's price from its size, number of bedrooms, zip code and the wealth of the neighbourhood. An engineer wiring the network by hand would give each hidden neuron a job: size and bedrooms say how large a family fits, the zip code says how walkable the area is, the zip code and wealth say how good the nearby school is, and those three predict the price better than the raw inputs. 48:51 A *fully connected* layer instead links every input to every neuron of the next layer and lets training decide what each hidden neuron computes. It often finds better features than the hand-designed ones, at the price of not saying what they are, which is why neural networks get called black boxes. Training only from input to label, with nothing constrained in between, is what *end-to-end learning* means.

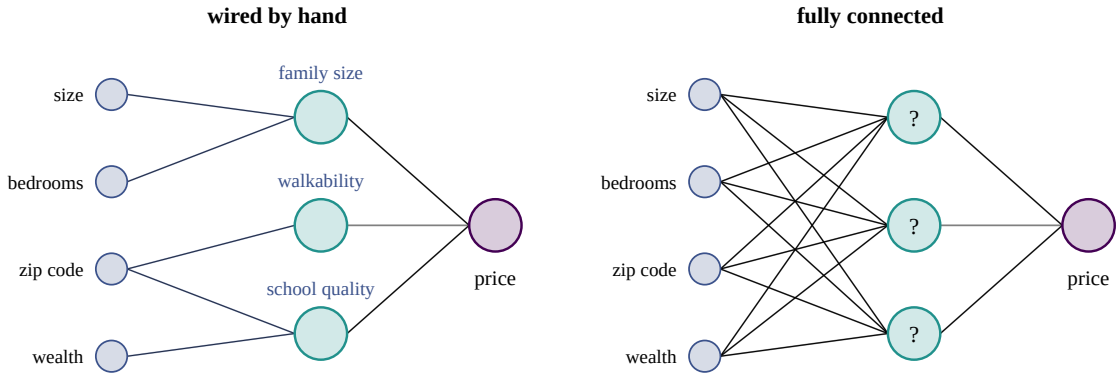


Figure 10.3.: Four inputs, three hidden neurons and a price. Left: the edges chosen from domain knowledge, so that each hidden neuron computes a named quantity. Right: a fully connected layer, where every hidden neuron sees every input and its meaning is left to training.

10.6. Forward propagation and shapes

51:31 For the three-layer cat network, pushing one example from input to output is six lines, each layer a linear part and an activation:

$$\begin{aligned} z^{[1]} &= W^{[1]}x + b^{[1]}, & a^{[1]} &= \sigma(z^{[1]}), \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]}, & a^{[2]} &= \sigma(z^{[2]}), \\ z^{[3]} &= W^{[3]}a^{[2]} + b^{[3]}, & \hat{y} = a^{[3]} &= \sigma(z^{[3]}). \end{aligned} \tag{10.6}$$

This is *forward propagation*. Where Section 10.2 wrote one equation per neuron, these write one per layer: $W^{[1]}$ stacks the three neurons' weight rows into a matrix, and the sigmoid is applied entry by entry.

53:13 The shapes follow from two facts, that $z^{[l]}$ has one entry per neuron in layer l and that $W^{[l]}$ must turn the previous layer's activations into it. So $z^{[1]}$ is 3×1 and x is $n \times 1$, which forces $W^{[1]}$ to be $3 \times n$; $a^{[1]}$ is 3×1 and $z^{[2]}$ is 2×1, so $W^{[2]}$ is 2×3, one entry per edge between the layers; and $W^{[3]}$ is 1×2. In general $W^{[l]}$ is $n_l \times n_{l-1}$ and $b^{[l]}$ is $n_l \times 1$, where n_l is the width of layer l. Katanforoosh's advice is to know every shape in the network before writing a line of code, and it is the tool that makes the next chapter's derivatives come out right.

10.7. Vectorising over a batch

54:59 Recall the two ends of gradient descent from Section 2.6: stochastic gradient descent updates after every example, with a noisy direction; batch gradient descent updates after seeing all of them, with an accurate one. Either way, a network with many layers does a lot of arithmetic per example, and the way to make that fast is to process many examples in one set of matrix operations rather than in a loop.

55:59 Stack m examples as the columns of an $n \times m$ matrix $X = [x^{(1)} \cdots x^{(m)}]$, using round-bracket superscripts for the example and square brackets for the layer. Then the first layer for the whole batch is

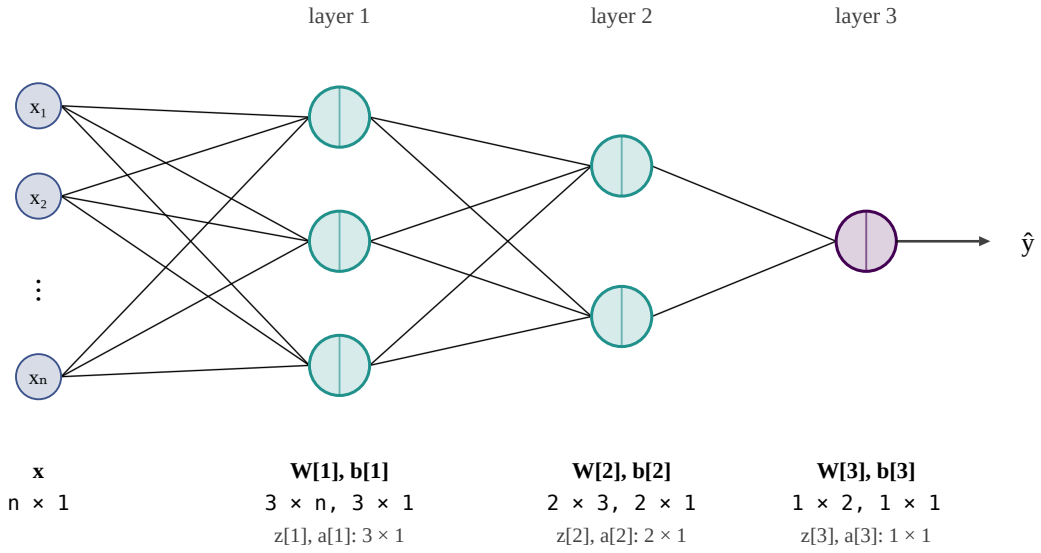


Figure 10.4.: The 3-2-1 network with the shape of every parameter and activation. Each weight matrix has one row per neuron in its layer and one column per neuron, or input, in the layer before.

$$Z^{[1]} = W^{[1]}X + b^{[1]}, \tag{10.7}$$

and $Z^{[1]}$ is $3 \times m$, its i-th column the $z^{[1]}$ of example i. $W^{[1]}$ is still $3 \times n$: 59:21 feeding the network a thousand images does not give it more parameters. 59:36 But that leaves Equation 10.7 adding a $3 \times m$ matrix to a 3×1 vector, which is not defined. The fix is *broadcasting*: read $b^{[1]}$ as the $3 \times m$ matrix whose columns are all $b^{[1]}$, so the same bias is added to every example. NumPy does this automatically when the shapes are compatible. The generator checks that the broadcast batch computation gives column for column what the per-example loop gives. The remaining lines of Equation 10.6 become capital-letter versions in the same way; the lecture leaves writing them out, with their shapes, as an exercise.

Two student questions in this stretch are worth keeping. 1:03:05 One asks how this differs from principal component analysis, which also builds new features from the inputs. PCA is unsupervised: it finds the directions along which the inputs vary most and never looks at a label. The hidden layer is trained to predict y, and a direction of large input variance that does not help with that is of no interest to it.

1:05:53 The other asks how to choose the number of layers and neurons. Nobody knows in advance, so the honest answer is to train several architectures and compare them on the validation set, as in Section 8.4. Experience narrows the search: a harder problem needs more data and a deeper network, and judging which problem is harder is a skill. Telling cats from non-cats seems harder than telling day from night, since cats come in many breeds and nights do not, though even day or night has its trap, a photo taken indoors next to a small window.

10.8. The cost, and why backpropagation starts at the end

1:09:06 With the architecture fixed, what is left is to find its parameters $W^{[1]}, b^{[1]}, \ldots, W^{[3]}, b^{[3]}$. The objective over a batch is the average loss,

$$J(\hat{Y}, Y) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}), \tag{10.8}$$

with L the logistic loss of Equation 10.2. The lecture keeps the words apart: the *loss* is the error on one example, the *cost* the average over a batch. 1:11:13 Differentiation is linear, so the gradient of J is the average of the gradients of the per-example losses, and it is enough to work out derivatives of L and average at the end. The generator confirms this numerically.

1:12:55 Gradient descent then updates every layer,

$$W^{[l]} \leftarrow W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}}, \qquad b^{[l]} \leftarrow b^{[l]} - \alpha \frac{\partial J}{\partial b^{[l]}}, \qquad l = 1, 2, 3, \tag{10.9}$$

which needs six derivatives. 1:14:13 Asked which to compute first, the room suggests $W^{[1]}$, and Katanforoosh argues for $W^{[3]}$. It is the closest to the loss, so its effect on J passes through the fewest functions, while $W^{[1]}$ affects J through every layer above it. Splitting the derivative along the chain of the forward pass,

$$\frac{\partial L}{\partial W^{[3]}} = \frac{\partial L}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial W^{[3]}},$$

every factor is easy, and the first two together are $\partial L / \partial z^{[3]}$. 1:17:33 The next layer down needs exactly that quantity again:

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial W^{[2]}},$$

so it is computed once and reused, and the same happens one layer further down with $\partial L / \partial z^{[2]}$. That is the whole idea of *backpropagation*: start at the loss, apply the chain rule one layer at a time, and carry the error backwards instead of recomputing it. 1:18:42 The forward equations say which path the chain must take. $z^{[3]}$ depends on $W^{[2]}$ only through $a^{[2]}$, so the chain for $W^{[2]}$ runs through $a^{[2]}$ and not through $W^{[3]}$, which is a dead end. The derivation, with the shapes that make the products well defined, is the first half of Section 11.1 in the next chapter.

> **In the official notes**
>
> cs229-notes-deep_learning, section 1, builds the network from the housing example; section 2 vectorises the output computation and the computation over training examples; section 3.1 counts the parameters of the three-layer network ($3n + 14$) and explains why they cannot all start at zero. The backpropagation derivation is in section 3 and in the separate cs229-notes-backprop.

11. Backprop and Improving Neural Networks

Lecture 11

Lecture video: Lecture 11: Backprop and Improving Neural Networks · 1:16:38

Based on the lecture by Kian Katanforoosh — CS229, Stanford University, Autumn 2018.

Backpropagation is the chain rule applied one layer at a time from the loss downwards, with each layer's intermediate result handed to the layer below instead of being recomputed. For the three-layer network of the previous chapter it comes down to one recurrence: the error at the output is prediction minus label, and each layer's error is the next layer's error carried back through the transposed weights and multiplied entry by entry by the slope of the activation. The first third of the lecture derives that by hand. The rest is about why a network that computes its gradients correctly can still fail to train, and the standard remedies: the choice of activation, normalised inputs, initial weights of the right size, mini-batches and momentum.

The lecture continues the previous one, again given by Kian Katanforoosh, on the same running example: a 64×64 colour image flattened into $x \in \mathbb{R}^n$, a layer of three sigmoid neurons, a layer of two, and one sigmoid output that says whether the image shows a cat (Section 10.6).

11.1. The gradient at the output

5:41 The cost is the average of the per-example logistic losses, and since differentiation is linear it is enough to differentiate one loss and average at the end (Section 10.8). Write the loss for one example with the output layer spelled out,

$$L = -\Big[\, y \log \sigma(W^{[3]} a^{[2]} + b^{[3]}) + (1-y) \log\Big(1 - \sigma(W^{[3]} a^{[2]} + b^{[3]})\Big)\Big],$$

and differentiate with respect to $W^{[3]}$, the parameter closest to it. Each term is a logarithm of a sigmoid of a linear function, so the chain rule gives three factors: $1/a^{[3]}$ from the logarithm, $a^{[3]}(1-a^{[3]})$ from the sigmoid, whose derivative is $\sigma'(z) = \sigma(z)(1-\sigma(z))$, and the derivative of $W^{[3]}a^{[2]} + b^{[3]}$ with respect to $W^{[3]}$. The second term is the same with $1/(1-a^{[3]})$ and a minus sign. 15:46 The sigmoid factors cancel against the logarithm's, the cross terms cancel against each other, and what is left is

$$\frac{\partial L}{\partial W^{[3]}} = (a^{[3]} - y)\, a^{[2]\top}. \tag{11.1}$$

The result "doesn't look that bad," as Katanforoosh puts it, and it is the same prediction-minus-label form as logistic regression (Equation 10.5), with the activations of the layer below standing in for the input.

12:05 A student asks where the transpose comes from. The answer is a shape argument, and it is the tool for the rest of the derivation. L is a scalar and $W^{[3]}$ is 1×2, so the derivative of one with respect to the other has one entry per entry of $W^{[3]}$ and must be 1×2 as well. In scalar calculus the derivative of wa with respect to w is a; here $a^{[2]}$ is 2×1, which is the wrong shape, so the answer has to be $a^{[2]\top}$. Either learn the matrix-calculus identities by heart, or learn to check shapes. The bias works the same way and gives $\partial L/\partial b^{[3]} = a^{[3]} - y$. Averaging over a batch, the cost's gradient is $\frac{1}{m}\sum_i (a^{[3](i)} - y^{(i)})\, a^{[2](i)\top}$, which in the matrix notation of Section 10.7 is the single product $\frac{1}{m}(A^{[3]} - Y)A^{[2]\top}$: the sum over examples is done by the matrix multiplication.

11.2. The chain rule, one layer down

19:05 For $W^{[2]}$ the lecture does not start over. It decomposes the derivative along the path that connects $W^{[2]}$ to the loss in the forward equations:

$$\frac{\partial L}{\partial W^{[2]}} = \underbrace{\frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}}_{\partial L/\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial W^{[2]}}.$$

20:24 The path has to go through $a^{[2]}$. $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$ also depends on $W^{[3]}$ and $b^{[3]}$, but differentiating with respect to those leads nowhere: they do not depend on $W^{[2]}$, and the error has to travel through variables that do. 22:04 Asked which part of this is already known, a student says the first three factors and then corrects it to two, which is right: since $\partial L/\partial W^{[3]} = (\partial L/\partial z^{[3]})\, a^{[2]\top}$, Equation 11.1 says $\partial L/\partial z^{[3]} = a^{[3]} - y$. The remaining factors are each one step of the forward pass. $\partial a^{[2]}/\partial z^{[2]}$ is the sigmoid's derivative, $a^{[2]} \circ (1 - a^{[2]})$; $\partial z^{[2]}/\partial W^{[2]}$ brings in $a^{[1]\top}$ as before; and 25:24 $\partial z^{[3]}/\partial a^{[2]}$ is $W^{[3]}$ or its transpose, which the shapes decide: $z^{[3]}$ is a scalar and $a^{[2]}$ is 2×1, so the factor is the 2×1 vector $W^{[3]\top}$.

28:01 Written in the order the chain rule produced them, the factors are 1×1, 2×1, 2×1 and 1×3, and they do not multiply. The derivation so far was a bookkeeping of which quantities appear, not of how they combine. Two facts put them together. The sigmoid acts on each entry of $z^{[2]}$ separately, so its derivative enters as an entry-by-entry product, written $\circ$, and that is true of every activation. And the result must have the shape of $W^{[2]}$, 2×3, which fixes the order of the rest:

$$\frac{\partial L}{\partial W^{[2]}} = \Big(W^{[3]\top}(a^{[3]} - y) \circ a^{[2]} \circ (1 - a^{[2]})\Big)\, a^{[1]\top}. \tag{11.2}$$

Inside the parentheses is a 2×1 vector, $\partial L/\partial z^{[2]}$, and the outer product with the 1×3 vector $a^{[1]\top}$ gives 2×3. The lecture leaves $W^{[1]}$ as an exercise; it comes out the same way, and the pattern is worth stating once in general. Write $\delta^{[l]} = \partial L/\partial z^{[l]}$ for the error at layer l of an N-layer network with activation g. Then

$$\delta^{[N]} = a^{[N]} - y, \qquad \delta^{[l]} = \big(W^{[l+1]\top}\delta^{[l+1]}\big) \circ g'(z^{[l]}), \qquad \frac{\partial L}{\partial W^{[l]}} = \delta^{[l]}\, a^{[l-1]\top}, \qquad \frac{\partial L}{\partial b^{[l]}} = \delta^{[l]}, \tag{11.3}$$

with $a^{[0]} = x$. The first equation holds for a sigmoid output with the logistic loss and for a softmax with cross-entropy alike. The generator implements exactly these four lines for the 3-2-1 sigmoid network and compares every one of its parameters' gradients with finite differences on fifty random networks; the worst disagreement is 4×10^{-10}.

31:21 Katanforoosh's view is that hardly anyone derives these by hand any more, since programming frameworks differentiate automatically, but that it is worth having done once, both to understand what the framework is doing and to read papers that write gradients out. For the rigorous version he points to the lecture notes.

11.3. What the forward pass must keep

32:04 Every factor in Equation 11.3 other than δ itself is something the forward pass already computed: the activations $a^{[1]}, a^{[2]}, a^{[3]}$, the values $g'(z^{[l]})$ depend on, and the weights. Discard them and the backward pass would have to re-run the forward pass from x each time it needed one, going forwards again to reach $a^{[2]}$, then again to reach $a^{[1]}$. So the forward pass *caches* what it computes and the backward pass reads it. This is a trade of memory for computation, and the memory is not small: every activation of every layer for every example in the batch is held until the backward pass reaches it.

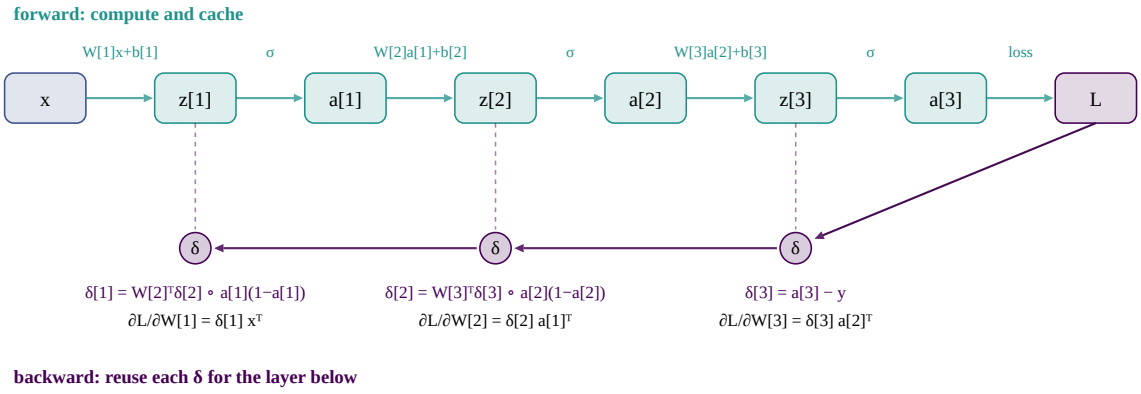


Figure 11.1.: Forward pass on top, left to right, caching each value. Backward pass below, right to left: the output error $\delta^{[3]} = a^{[3]} - y$ is carried down by $W^{[l+1]\top}$ and the activation's slope, and each layer's weight gradient is its δ times the cached activation that fed it.

11.4. Activation functions

34:44 Getting the gradients right is necessary and nowhere near sufficient; a network built and differentiated correctly usually still trains badly, and the rest of the lecture is the set of practical fixes. The first is the choice of activation. There are three in common use:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \qquad \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}, \qquad \mathrm{ReLU}(z) = \max(0, z),$$

with derivatives $\sigma(1-\sigma)$, $1-\tanh^2$, and the indicator $\mathbb{1}\{z>0\}$. 37:08 The sigmoid's strength is that its output reads as a probability. Its weakness, which a student names, is *saturation*: for large positive or negative z the curve is flat, its derivative is nearly zero, and since that derivative is a factor in every δ below it, layers early in the network get almost no gradient and barely move. The derivative is small even at its best. Its maximum is $\frac{1}{4}$, at $z=0$, so every sigmoid layer shrinks the gradient passing through it by at least a factor of four, and ten of

them shrink it by at least 4^{10}, about a million. tanh saturates in the same way at both ends, but its slope at the centre is 1 and its output is centred on zero.

38:52 ReLU does not saturate for positive inputs: its slope there is exactly 1, so it passes the gradient through unchanged rather than scaling it down. Its slope for negative inputs is exactly 0, which has its own cost the lecture does not get to: a ReLU neuron whose input is negative for every example receives no gradient at all and stops learning. That trade has worked out well enough that ReLU is the usual choice for hidden layers.

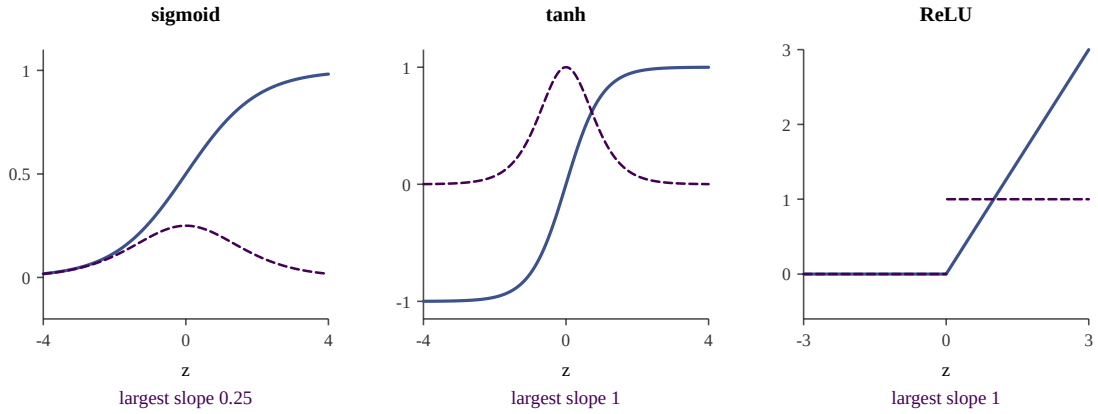


Figure 11.2.: Sigmoid, tanh and ReLU (solid) with their derivatives (dashed). Only ReLU keeps a slope of 1 for large inputs; the sigmoid's slope never exceeds 0.25.

39:53 For the *output* layer the choice is made by the range of what is predicted, not by trainability: a sigmoid for a probability in $(0, 1)$, tanh for values in $(-1, 1)$ such as the rewards of reinforcement learning, a ReLU for a non-negative quantity such as a house price, the identity for an unrestricted one. 45:18 Within a hidden layer the convention is one activation for every neuron, so that a layer is described as, say, a fully connected ReLU layer. People have tried mixing activations inside a layer without finding a reason to, and the choice is treated as a hyperparameter like any other, settled by experiment rather than by theory.

11.5. Why the network needs a non-linearity

41:06 A student asks why these particular functions, and Katanforoosh answers by taking the activation away. With the identity in every neuron of the 3-2-1 network the forward pass substitutes into itself:

$$\hat{y} = W^{[3]}\Big(W^{[2]}\big(W^{[1]}x + b^{[1]}\big) + b^{[2]}\Big) + b^{[3]} = \underbrace{W^{[3]}W^{[2]}W^{[1]}}_{W}\,x + \underbrace{W^{[3]}W^{[2]}b^{[1]} + W^{[3]}b^{[2]} + b^{[3]}}_{b}. \tag{11.4}$$

43:52 Three layers of parameters compute exactly what one layer computes, an affine function $Wx + b$, and adding layers changes nothing: however deep, the network is a linear regression. The capacity of a network to represent complicated functions comes entirely from its activations. The lecture's phrasing is that any function but the identity will do; strictly, the problem is any affine activation, since $g(z) = cz + d$ collapses in the same way, and what is needed is a non-linear one. The generator confirms the collapse on random inputs to machine precision.

11.6. Normalising the inputs

48:16 If the inputs are large, $z = Wx + b$ is large and the sigmoids saturate before training starts. The simple remedy is to standardise the inputs using the training set: compute the mean and variance of each feature,

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \qquad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu\right)^2,$$

entry by entry, and replace every x by $(x - \mu)/\sigma$. Subtracting μ centres the cloud of points on the origin; dividing by σ gives every feature the same spread.

51:18 The second reason is the shape of the cost. When features have very different scales, or sit far from zero, the cost is a long narrow valley in parameter space, and gradient descent, which moves perpendicular to the contours, crosses the valley again and again while making slow progress along it. After standardising, the contours are closer to circles, the gradient points nearly at the minimum, and descent goes almost straight there. Figure 11.3 makes the comparison on a least-squares problem with two features, one centred at 2 and the other on a small scale. The ratio of the cost's curvatures along its two axes is 29 with the raw inputs and 1 after standardising, and gradient descent, each run at its best fixed learning rate, needs 100 steps against 2. Standardising equalises scales, not correlations: afterwards the curvature matrix of a least-squares cost is the features' correlation matrix, so two strongly correlated features still leave a long valley, tilted at 45°, which decorrelating transforms such as the PCA of a later chapter address.

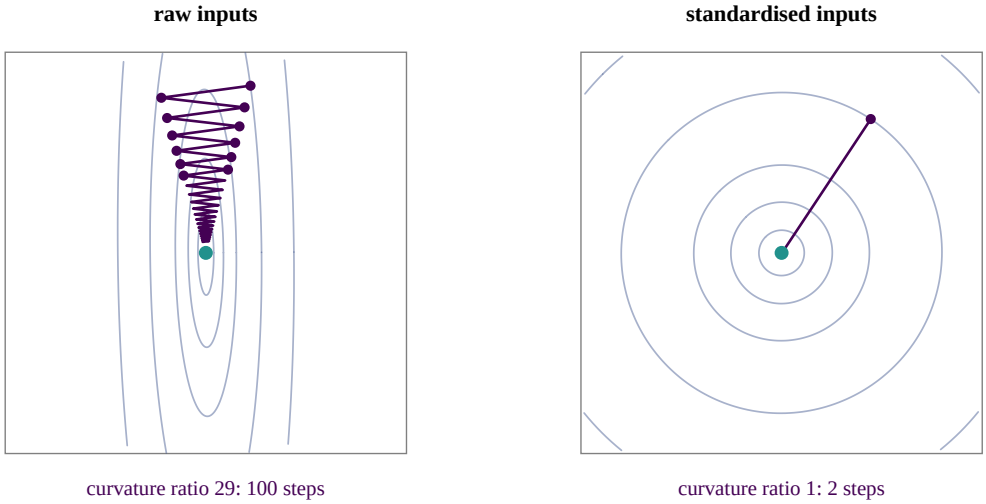


Figure 11.3.: Contours of a least-squares cost in its two weights and the path of gradient descent, with raw inputs (left) and the same inputs standardised (right). Each run uses its best fixed learning rate.

53:03 One rule matters more than the formulas: μ and σ are computed on the training set and then reused, unchanged, on the dev and test sets. The network has learned to expect inputs transformed in one particular way, and normalising the test set by its own statistics would feed it a differently transformed input.

11.7. Vanishing and exploding gradients

54:19 To see why depth makes all of this harder, take a network of ten layers with two neurons each, identity activations and zero biases. The output is then the input multiplied by every weight matrix in turn,

$$\hat{y} = W^{[N]} W^{[N-1]} \cdots W^{[1]} x.$$

56:05 If every $W^{[l]}$ is a little larger than the identity, say $1.5\,I$, the product is $1.5^N I$, which for $N = 10$ is about 58; if every one is $0.5\,I$, it is $0.5^{10} \approx 0.001$. Small departures from 1 compound geometrically. 57:16 The same products appear in the backward pass, because Equation 11.3 multiplies by $W^{[l+1]\top}$ and by g' once per layer, so the gradients reaching the early layers either explode or vanish. With sigmoids the g' factors alone contribute at most $\frac{1}{4}$ each (Section 11.4). Nothing in the gradient computation is wrong; the arithmetic is simply unforgiving over many layers.

11.8. Initialising the weights

58:24 The lecture's goal is to keep each layer's multiplier "close to 1", which needs a little care, because that is a statement about the layer and not about individual weights. A neuron computes $z = \sum_{i=1}^{n} w_i x_i$, a sum of n terms, and the more inputs it has, the larger z gets unless the weights shrink to compensate. Make that precise: if the weights and inputs are independent with mean zero, then

$$\operatorname{Var}(z) = n \operatorname{Var}(w) \operatorname{Var}(x),$$

so z has the same variance as its inputs when $\operatorname{Var}(w) = 1/n$, or standard deviation $1/\sqrt{n}$. The lecture's code multiplies standard normal draws by $\sqrt{1/n^{[l-1]}}$, where $n^{[l-1]}$ is the number of inputs to layer l. The generator checks the variance identity by simulation, getting 0.96 against 1.

1:02:07 Katanforoosh reports, as an empirical observation, that a ReLU network trains better with 2 in place of 1. There is a clean reason, given by He et al. (2015): a ReLU zeroes the negative half of a symmetric input, which halves the mean square of what it passes on, so the variance needs to start twice as large to come out the same. Figure 11.4 shows how quickly this matters. In a 30-layer ReLU network with weights of variance $1/n$, the signal loses half its mean square at every layer and ends about 2^{-15} of its starting size; with $4/n$ it grows by the same factor; with $2/n$ it stays level.

1:02:38 The lecture then presents "Xavier initialisation" and "Glorot initialisation" as two different schemes. They are one person's name: the method is from Glorot and Bengio (2010), whose first author is Xavier Glorot. The paper asks for two things at once, that activations keep their variance going forwards, which needs $n_{\text{in}} \operatorname{Var}(w) = 1$, and that gradients keep theirs going backwards, which needs $n_{\text{out}} \operatorname{Var}(w) = 1$. Both cannot hold unless the layer is square, and the compromise it proposes is

$$\operatorname{Var}(w) = \frac{2}{n_{\text{in}} + n_{\text{out}}},$$

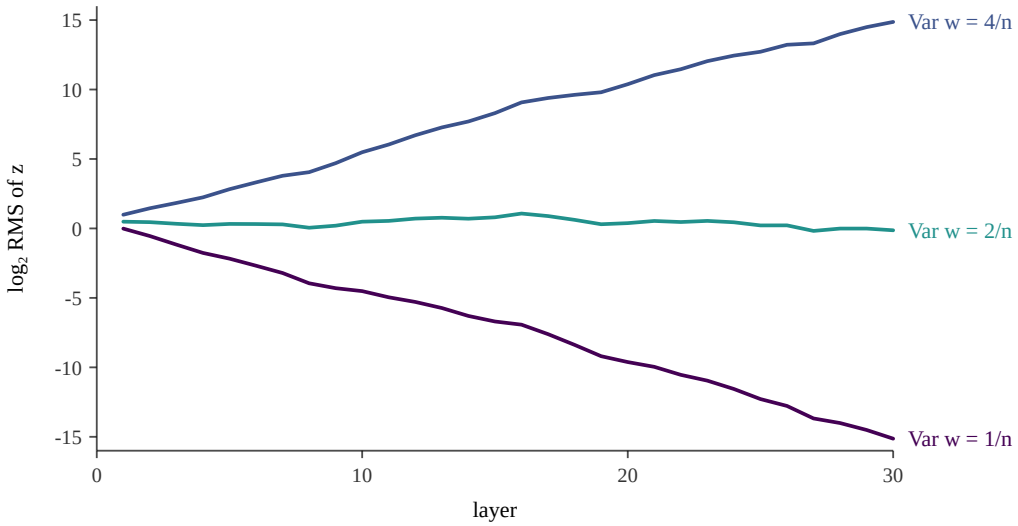


Figure 11.4.: The root-mean-square size of the pre-activations at each layer of a 30-layer ReLU network, 64 units wide, for three weight variances. Only $2/n$ holds the signal steady.

the reciprocal of the *arithmetic* mean of the fan-in and fan-out (the lecture says a geometric average). This is the formula the official notes give, under the name "Xavier/He", though He's $2/n$ is a different rule for a different activation. Glorot and Bengio derived theirs for tanh-like units; for ReLU networks, He's is the one that holds the variance.

1:04:02 Finally, the initial weights must be *random*, and not merely small. If all the weights of a layer start equal, every neuron in it computes the same function, receives the same gradient, and is updated identically, forever: the layer behaves as a single neuron, however wide it is. The generator trains the 3-2-1 network from equal weights for 6,000 steps of stochastic gradient descent, and the three hidden neurons' weights end up identical to the last bit. Starting from zero is the extreme case, and the official notes work it through. Random initial values break the symmetry, so that the neurons start in different places and can learn different things; biases have no such problem and can start at zero.

11.9. Mini-batch gradient descent

1:05:16 The two versions of gradient descent seen so far sit at opposite extremes. Batch gradient descent vectorises well but makes one update per pass through the data, and with a million images one update is very slow. Stochastic gradient descent updates after every example, quickly but noisily, and gives up vectorisation. *Mini-batch* gradient descent takes the middle. Split the training set into batches, written with curly braces, $X^{\{1\}}, \dots, X^{\{T\}}$, of 1,000 examples each, and loop:

1. pick a batch $(X^{\{t\}}, Y^{\{t\}})$;
2. forward-propagate the whole batch and compute its cost $J^{\{t\}} = \frac{1}{1000} \sum_i L(\hat{y}^{(i)}, y^{(i)})$;
3. backpropagate to get the gradients of $J^{\{t\}}$;
4. update $W^{[l]}$ and $b^{[l]}$ for every layer.

Each update is a matrix computation over a thousand examples, which a GPU does at once, and a thousand examples already give a useful estimate of the direction to move in. 1:09:30 The price is noise. Plotted against iterations, the cost of batch gradient descent falls smoothly, while that of mini-batch descent falls on average but wobbles, since each step follows one batch's estimate of the gradient. The smaller the batch, the larger the wobble, and quantitatively: for batches drawn at random, the variance of the gradient estimate is inversely proportional to the batch size. On a toy regression the generator measures 9.9, 0.98 and 0.10 for batches of 1, 10 and 100.

11.10. Momentum

1:11:24 Return to the long narrow valley of Section 11.6. Gradient descent zig-zags across it because the steep direction dominates every gradient, and the useful step would be large along the valley and small across it. *Momentum* gets there by looking at the history of updates. 1:13:27 Averaged over a few steps, the components across the valley alternate in sign and cancel, while the components along it all point the same way and add up. Stepping along the average rather than the latest gradient damps the zig-zag and builds up speed along the floor. In the lecture's form,

$$v \leftarrow \beta v + (1-\beta)\,\frac{\partial J}{\partial W}, \qquad W \leftarrow W - \alpha v, \tag{11.5}$$

where v, the velocity, is an exponentially weighted average of past gradients, and β, often around 0.9, sets how long a memory it has. 1:15:42 It costs one more line of code and one more variable per parameter. The physical picture is a heavy ball rolling downhill: its velocity cannot reverse at every bump, and $1-\beta$ plays the part of friction.

A computation sharpens the picture in a way the lecture does not. In this form, at the *same* learning rate as gradient descent, momentum helps only modestly: on a quadratic 100 times steeper across the valley than along it, gradient descent at its best fixed rate takes 346 steps to converge and momentum with $\beta = 0.9$ at that rate takes 272. The large gain comes from what momentum makes possible. Gradient descent diverges as soon as α exceeds $2/\lambda_{\max}$, here 0.02, while the averaging raises the limit to $2(1+\beta)/((1-\beta)\lambda_{\max})$, here 0.38. At $\alpha = 0.06$, three times the rate that makes gradient descent blow up, momentum converges in 90 steps.

1:15:58 The optimisers used most in practice, RMSProp and Adam, go further and give every parameter its own effective learning rate, scaled by a running average of its squared gradients. Adam combines that scaling with momentum. The lecture's closing point about them is a sociological one: a new optimiser has to prove itself across a wide range of problems before people adopt it, which is why so few are in common use.

> **i** In the official notes
>
> cs229-notes-backprop derives the backward recurrence for a network of any depth, with the matrix calculus the lecture replaces by shape analysis. cs229-notes-deep_learning, section 3, covers backpropagation for the three-layer network; section 3.1 the parameter count, why zero and equal initialisations fail, and the variance-scaled initialisation; section 3.2 stochastic and mini-batch gradient descent and momentum; and section 3.3, which the lecture did not reach, L2 regularisation and parameter sharing.

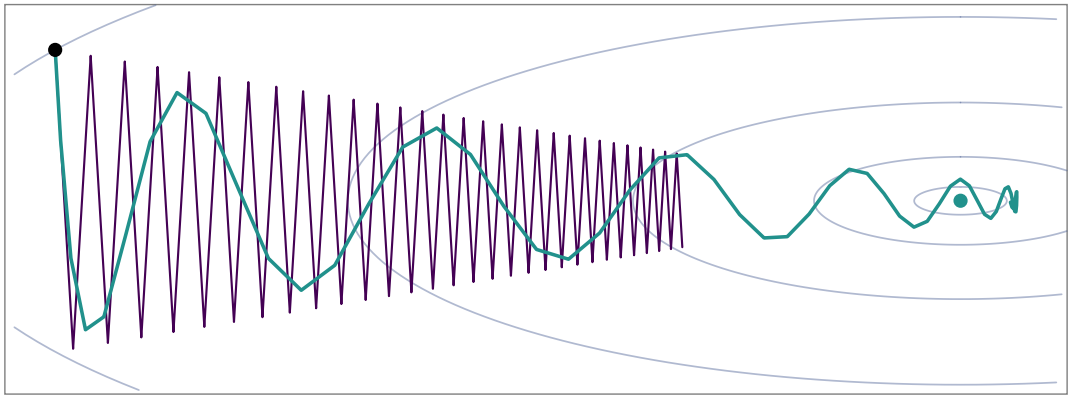


Figure 11.5.: The first 60 steps of gradient descent (at its best fixed rate) and of momentum (at a rate three times past gradient descent's stability limit) on a quadratic 100 times steeper vertically than horizontally.

Part IV.

Applying Machine Learning

12. Debugging ML Models and Error Analysis

Lecture 12

Lecture video: Lecture 12: Debugging ML Models and Error Analysis · 1:18:55

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

A learning algorithm that does not work can be fixed in a dozen ways, and most teams pick one at random. This lecture is about picking on evidence. Each diagnostic in it makes the same move: state a hypothesis about what is wrong, then find a measurement that comes out differently depending on whether the hypothesis holds. Learning curves separate an algorithm that overfits from one that underfits. Comparing two parameter vectors on the training objective tells a failing optimiser apart from an objective that is the wrong thing to optimise. Error analysis works out which stage of a pipeline is costing the most accuracy, and ablative analysis works out which component is actually earning its keep.

Ng is explicit that this is advice for building systems that work, not for writing research papers, and that it compresses what is otherwise learned from years of projects into a handful of worked scenarios. The chapter keeps his scenarios: a spam filter, an autonomous helicopter and a face-recognition door lock.

12.1. What to try next

4:42 The running example is a spam filter. You have hand-picked a hundred words as features, trained logistic regression with a penalty on $\|\theta\|^2$ (which Ng also calls Bayesian logistic regression, since the penalty is a Gaussian prior on θ; see Section 8.3), and it has 20 per cent test error. That is one mistake in every five emails, which is not good enough.

The list of things a team might try writes itself. Get more training examples. Use fewer features, since some of the hundred are probably irrelevant, or more features, since a hundred may be too few. Add features from the email header, including the routing information that records which servers the message passed through on its way to you. Run gradient descent for more iterations, or switch to Newton's method. Change λ. Abandon logistic regression for an SVM or a neural network.

8:25 Ng's observation is that teams usually pick one of these almost at random, going on what someone read the night before or what worked on the last project, and then spend days or weeks finding out whether it helped. Just writing the list down and weighing the options already puts a team ahead of most. The real fix is to work out first what kind of problem you have, because each item on the list treats only one kind. More data never hurts, but in some situations it barely helps, and finding that out by collecting it is expensive.

He also notes that he says "test error" in what follows but means error on a dev set, since decisions made by looking at the test set use it up (Section 8.4).

12.2. Learning curves

12:00 The diagnostic Ng says he uses on nearly every project is bias against variance (Section 8.1). An algorithm with high variance does much better on its training set than on the dev set. An algorithm with high bias does badly on both. The tool that tells these apart is the *learning curve*: training error and dev error against the number of training examples m, with a horizontal line at the error you need to reach for the application to be useful. The earlier bias–variance plots had model complexity on the horizontal axis. This one holds the model fixed and varies the amount of data.

Two facts set the shape. Dev error usually falls as m grows. Training error usually *rises*, because a small training set is easy to fit: logistic regression fits one email perfectly and five almost certainly, and a straight line passes through any two points. Only with many examples does the model start to miss some of them. Since training error is normally below dev error, the two curves approach each other from opposite sides.

Figure 12.1 draws both regimes for least-squares regression, where the expected curves have closed forms. With p Gaussian features, noise variance σ^2 and $m > p+1$ examples, the expected training and test mean squared errors are

$$\mathbb{E}[\text{train}] = \sigma^2\Big(1 - \frac{p}{m}\Big), \qquad \mathbb{E}[\text{test}] = \sigma^2\Big(1 + \frac{p}{m-p-1}\Big). \tag{12.1}$$

Any signal the model cannot represent acts as extra noise, so in an underfitting model σ^2 is replaced by the noise plus the variance it leaves unexplained. The generator checks both formulas against least squares fitted to simulated data.

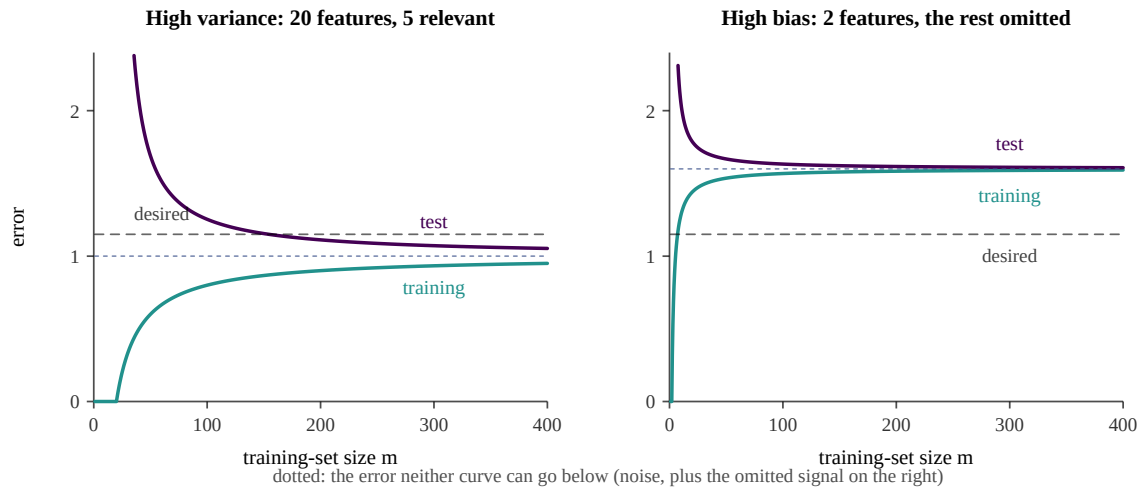


Figure 12.1.: Expected training and test error of least squares against training-set size, computed from Equation 12.1. Left: 20 features, of which 5 matter. Right: 2 features when the target needs more.

The left panel shows high variance: twenty features of which only five matter, so the fit chases noise. Ng reads two signals from a plot like this. The weaker one is that the dev error is still falling at the right edge, which suggests more data would help, but extrapolating a curve by eye is unreliable. The stronger one is the large gap between the curves. Training error sits near the

target while dev error is well above it, and as the curves have to close that gap, more data will bring the dev error down. In this example it reaches the target at 155 examples.

The right panel shows high bias: two features when more are needed. Here the training error is already above the target, and since training error rises with m and dev error sits above training error, no amount of data brings the dev error below the target. The curves have nearly met and both have levelled off. Collecting more data in this situation might help slightly but can never get you where you need to go.

21:12 A student asks whether these shapes depend on the data. Ng's answer is that they assume the training, dev and test sets come from one distribution, that learning theory suggests the dev error falls roughly as $1/\sqrt{m}$, and that it need not fall to zero, because some examples are ambiguous even to an expert: a blurry X-ray, or a medical record that does not say whether the patient will survive the year.

The $1/\sqrt{m}$ is worth qualifying. It is the rate in the *uniform convergence* bound of the learning-theory chapter, which is a worst-case guarantee across a whole hypothesis class. A particular algorithm can approach its limit faster. For least squares, Equation 12.1 puts the test error's excess over the noise at $\sigma^2 p/(m-p-1)$, which falls as $1/m$. Doubling the training set from 200 to 400 examples cuts the excess in the left panel from 0.112 to 0.053, a factor of 0.47, where $1/\sqrt{m}$ would predict 0.71. The practical effect is that a learning curve can flatten sooner than the bound suggests, which makes Ng's warning against extrapolating curves by eye even more apt.

12.3. Matching the fix to the diagnosis

19:08 With the diagnosis in hand, the first four items on the spam list sort themselves out. More training examples fix high variance: they are what moves the left panel's dev curve down, and they do nothing much for the right panel. A smaller set of features also fixes high variance. Ng's picture is a tenth-order polynomial that wiggles through every point, where dropping the high-order terms removes the wiggle. A larger set of features fixes high bias, as when a straight line is not doing well and you add a quadratic term. Header features are another way of adding information, so they are something to try against bias.

24:16 He adds advice on how to start, which the earlier chapter on Naive Bayes already raised (Section 6.4). On a new application it is very hard to know in advance what will go wrong, so implement something quick and dirty, such as plain logistic regression, run the bias–variance diagnosis on it, and let the result decide whether the next step is a bigger model or more data. The exception is a domain where you already have experience. Ng has done enough speech recognition to know roughly how much data a system needs, and someone who has read the face-recognition literature can build on it. Without that experience, the diagnosis on a simple system is the cheapest way to learn where the difficulty lies. His slides call the alternative, a careful up-front design of exactly the right features and architecture, a risk of "premature statistical optimisation". The term parallels premature optimisation in software: effort spent on a part of the system that turns out not to be the bottleneck.

12.4. Optimisation algorithm or objective

26:30 The second diagnostic addresses the last four items on the list. Change the spam example: logistic regression now has 2 per cent error on spam and 2 per cent on non-spam. The first is tolerable, since you read the occasional spam message. The second is not, because it loses one

important email in fifty. A teammate's linear SVM gets 10 per cent error on spam but 0.01 per cent on non-spam, which is acceptable. You would still rather ship logistic regression, because it is cheaper to run and easier to update as new email arrives. What should you do?

Two quantities are in play. What you care about is a weighted accuracy on the dev set,

$$a(\theta) = \frac{\sum_i w^{(i)}\, 1\{h_\theta(x^{(i)}) = y^{(i)}\}}{\sum_i w^{(i)}},$$

with a larger weight $w^{(i)}$ on the non-spam emails. What logistic regression maximises is something else, the regularised log-likelihood

$$J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)} \mid x^{(i)}; \theta) - \lambda \|\theta\|^2.$$

34:09 A student asks the natural question: why not maximise $a(\theta)$ directly? Because it is a sum of indicator functions. As θ moves, a stays constant until some example crosses the decision boundary and then jumps, so its gradient is zero almost everywhere and gradient ascent has nothing to follow. The generator confirms this: at random parameter vectors, a finite-difference gradient of the weighted accuracy is exactly zero. Ng adds that maximising it exactly is NP-hard. In the learning-theory literature this is the hardness of agnostically learning halfspaces: even finding a halfspace that classifies slightly better than chance on data that some halfspace classifies almost perfectly is intractable (Guruswami and Raghavendra, 2009). So we optimise J as a proxy, and the question is whether the proxy or its optimisation is letting us down.

Write θ_{SVM} for the SVM's parameters, put in the same (θ, x) form as logistic regression, and θ_{BLR} for logistic regression's. We know that $a(\theta_{\text{SVM}}) > a(\theta_{\text{BLR}})$. Two hypotheses could explain this. Either the optimiser has not found the maximum of J, or J is the wrong function to maximise. 34:40 The diagnostic is to evaluate logistic regression's own objective at both parameter vectors and see which is higher.

Case 1: $J(\theta_{\text{SVM}}) > J(\theta_{\text{BLR}})$. Logistic regression was trying to maximise J, yet the SVM, which was not even trying, found a point where J is higher. So θ_{BLR} is not the maximiser, and the problem is the optimisation algorithm.

Case 2: $J(\theta_{\text{SVM}}) \le J(\theta_{\text{BLR}})$. Logistic regression did better on J, yet the SVM did better on a. A higher value of J therefore does not mean a higher weighted accuracy, so maximising J is not the right thing to do. The problem is the objective.

40:09 The remaining fixes on the list now sort the same way. Running gradient descent longer and switching to Newton's method fix the optimisation algorithm, since Newton's method maximises the same J, often much faster. Changing λ changes the objective, although Ng notes that in practice it is usually tuned for bias and variance instead. Switching to an SVM replaces the objective altogether. A student asks what to do once the objective is known to be wrong, and the honest answer is that nothing is automatic: you brainstorm objectives. In this example one obvious candidate is to put the weights $w^{(i)}$ into the log-likelihood, but the diagnostic only says which category of fix to look in.

The two cases are not symmetric, and the slides claim more for case 2 than the inequality delivers. Case 1 is a proof: θ_{SVM} is a point where J is higher than where the optimiser stopped, so the optimiser has not finished. Case 2 proves that the objective is wrong, but the slides go further and read it as "BLR succeeded at maximizing J". The inequality says only that logistic regression beat one particular point, and nothing forces that point to be anywhere near the

maximum. Figure 12.2 runs the test along a training run on a toy spam problem. An SVM with errors on non-spam weighted ten times wins on weighted accuracy, 0.927 against 0.855. Logistic regression starts from $\theta = 0$, and after a *single* gradient step its J already exceeds the SVM's, -159.6 against -160.0, so case 2 fires. Gradient ascent then needs another 31 steps to get within one per cent of the optimum at -86.5.

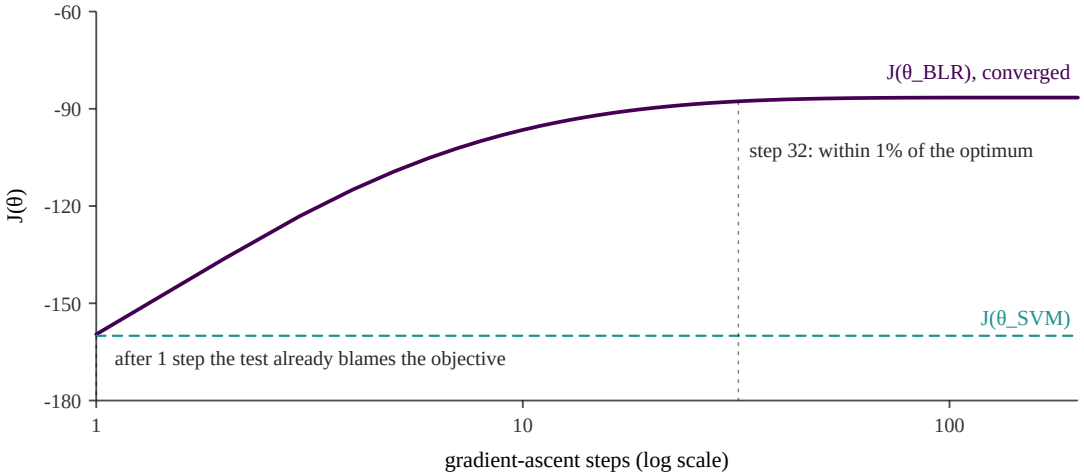


Figure 12.2.: Logistic regression's objective along gradient ascent on a toy spam problem (log step axis), against its value at the SVM's parameters. The SVM wins on weighted accuracy at every step.

The SVM's J is low in this example partly because its parameters were never chosen to make good probabilities. Scaling θ_{SVM} leaves every prediction, and hence a, unchanged but changes J. Here the best rescaling only lifts $J(\theta_{\text{SVM}})$ from -160.0 to -158.8, which is still nowhere near the optimum. So read case 2 as "the objective is wrong", which it shows, and not as "the optimiser is fine", which it does not. Both can be true at once. Ng's own practice in the helicopter story below, re-running the diagnostics after each fix, covers exactly that.

12.5. The helicopter

43:17 The more complex example is the autonomous helicopter from Ng's group, built in three steps. First build a simulator, because a simulated helicopter can crash a thousand times for free. Then choose a cost function. Ng keeps it simple for now: the squared distance $\|x - x_{\text{desired}}\|^2$ between where the helicopter is and where it should be, summed over the flight. Finally run a reinforcement-learning algorithm, which the course covers in its last lectures, to find controller parameters θ_{RL} that minimise the expected cost $J(\theta)$ in the simulator.

The resulting controller flies much worse than a human pilot. Each of the three steps suggests a fix, and 48:10 each fix could absorb a PhD. Simulators can always be made more accurate: airflow around a helicopter is complicated enough that parts of it are still not fully understood, so there is almost unlimited room to improve one. The cost function may be wrong. A hovering helicopter is not level, for example: the tail rotor pushes the body sideways and the pilot tilts against it, so penalising position alone may not capture what good flight is. And the reinforcement-learning algorithm may not be finding the minimum of J. Ng knows of a PhD thesis spent on helicopter simulation. That is a fine outcome if the goal is the thesis, and a poor one if the goal is a helicopter that flies.

49:50 The reasoning that cuts this down is the same as in the spam example. Suppose all three steps were sound: the simulator is accurate, the RL algorithm minimises J in it, and minimising J means flying well. Then θ_{RL} would fly well on the real helicopter. It does not, so at least one of the three is false. The job is to find which, much as one isolates a bug in a large program by bisecting it instead of reading the whole code base. Figure 12.3 lays out the two tests.

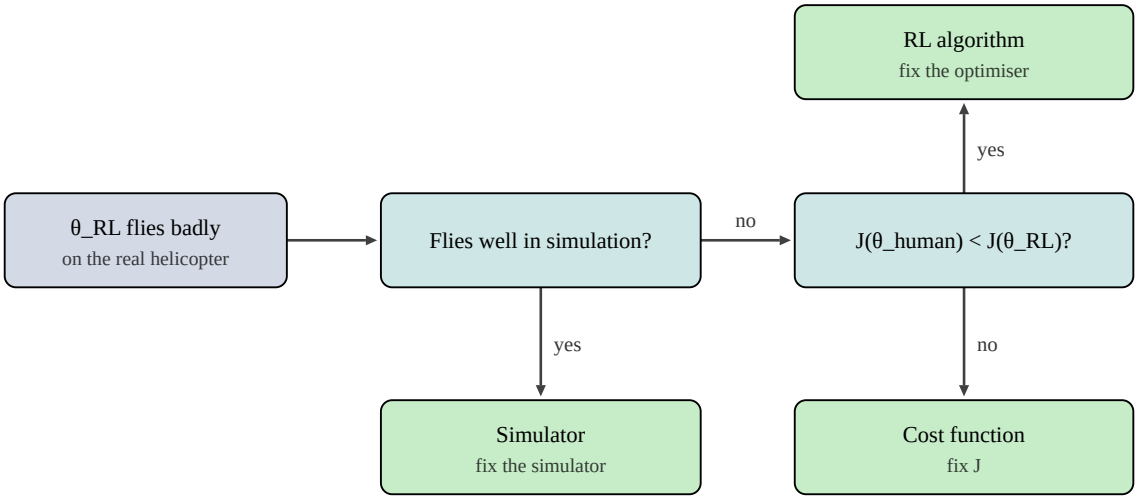


Figure 12.3.: The helicopter diagnostics. First check the controller in simulation, then compare the human pilot with the controller on the cost the RL algorithm minimises.

51:48 First, fly θ_{RL} in the simulator. If it flies well there and badly in reality, the RL algorithm did its job on the problem it was given, and the simulator is what differs from the real world. A student asks whether the reverse can happen, flying badly in simulation and well in reality. Ng says it is rare, and when it happens he still works on the simulator. His group deliberately adds noise to its simulators so that a controller robust to the simulated noise is more likely to be robust to real disturbances, and a controller that does well in reality but not in simulation can mean that too much noise was added.

54:40 If the controller also flies badly in simulation, compare it with the human on the RL algorithm's own cost: record the human pilot's flight and evaluate the same squared error. If $J(\theta_{\mathrm{human}}) < J(\theta_{\mathrm{RL}})$, the human achieved a lower cost than the algorithm found, so the algorithm is not minimising J, and that is the part to improve. If $J(\theta_{\mathrm{human}}) \geq J(\theta_{\mathrm{RL}})$, the algorithm reached a cost at least as low as the human's while flying worse, so a low J does not mean good flight, and the cost function is wrong. The same asymmetry as in the previous section applies here, now with the human as the witness point.

58:10 In the actual project, Ng says, the bottleneck moved around. For a few weeks the diagnostics pointed at the simulator; once it was good enough they pointed at the RL algorithm, and later at the cost function. Running them again after each round of work is what kept the team on the part that mattered. The cost function turned out to be the hard one, because it is not obvious how to write down mathematically what a good banked turn is. His student Adam Coates found a way through learning the target trajectory from several imperfect expert demonstrations, work published as Learning for Control from Multiple Demonstrations (Coates, Abbeel and Ng), which was presented in the award-paper session of ICML 2008.

12.6. Error analysis

1:01:24 Many applications are not one learning algorithm but a pipeline of them. Ng's example is a camera by a door that unlocks for authorised people. The image goes through background removal, which with a fixed camera can be as simple as keeping the pixels that changed. Then a face detector finds the face, separate components segment the eyes, nose and mouth (the eyes being a particularly strong cue for identity), and logistic regression on those features decides whether to open the door. Ng adds that pipelines like this are what small data sets call for. His teams often get good results from a hundred examples, but only through more careful design than a single end-to-end model needs.

When such a system reaches 85 per cent accuracy, which component should you work on? 1:06:04 Ng tells the story of someone who chose background removal, which can be made almost arbitrarily sophisticated by modelling the trees and bushes moving behind the subject, and wrote a whole PhD thesis on it, without improving the face recognition he set out to build.

1:07:50 Error analysis answers the question by substituting ground truth. On the dev set, replace the output of background removal with a perfect one, labelled by hand, and measure the system's accuracy. Then also give it perfect face locations, then perfect eye, nose and mouth segmentations, and finally perfect labels, which must give 100 per cent. Figure 12.4 shows the lecture's numbers.

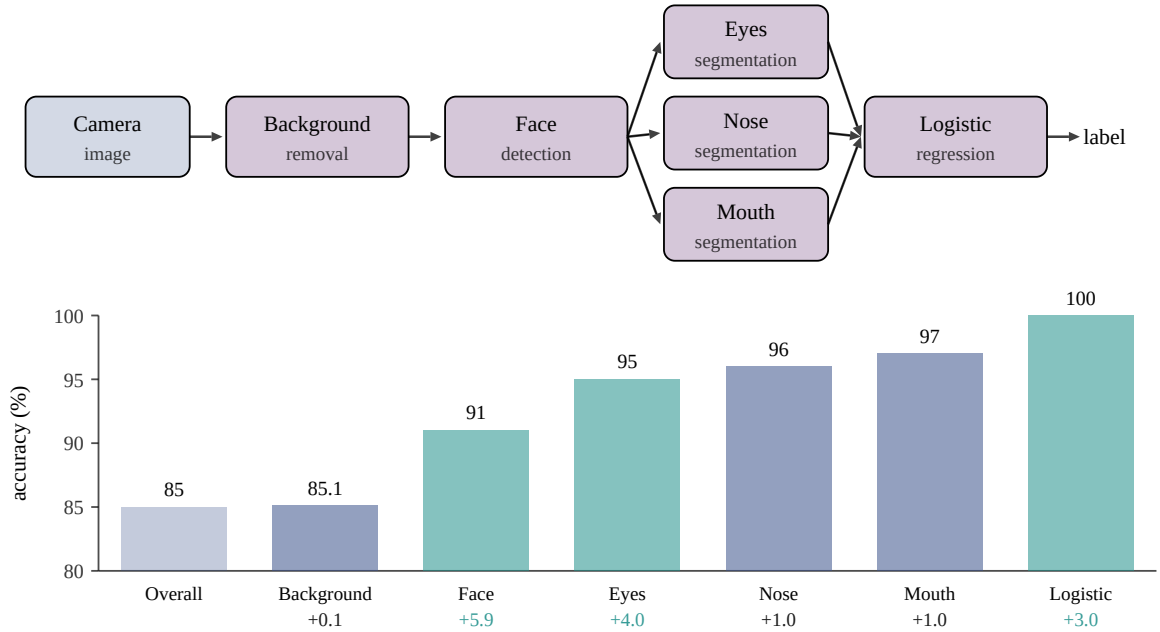


Figure 12.4.: The face-recognition pipeline (top) and the system's accuracy as each component, from left to right, is cumulatively replaced by ground truth (bottom). The lecture's illustrative numbers.

Perfect background removal is worth 0.1 points, so a thesis on it can change the door lock's accuracy by at most that much. Perfect face detection is worth almost six points and perfect eye segmentation another four, so those are the components to work on, in parallel if the team is big enough. The fifteen points between 85 and 100 per cent are divided among the stages, and each stage's share bounds what improving it could gain.

Two student questions sharpen this. 1:11:13 Should the substitution be cumulative, as in the figure, or one component at a time? Ng's answer is that either will do, that they usually agree, and that if you suspect they will not, you should run both. Ablation, below, gives a concrete case where they do not. 1:13:21 What if giving a component perfect output makes the system *worse*? That should be rare, and Ng's advice is not to gloss over it. Treat it like a button that misbehaves only when the mouse hovers over it: an anomaly to understand, not one to dismiss because users will not notice. He also notes that the numbers are a guide to prioritisation, to be combined with a judgement of how hard each component would be to improve. They do not settle the decision by formula.

12.7. Ablative analysis

1:14:35 Error analysis explains the gap between where a system is and perfection. *Ablative analysis* explains the gap between where it is and a much worse baseline. Back to spam: suppose a good classifier has been built by adding clever components to plain logistic regression. These are spelling correction (spammers misspell words to get them past the tokeniser), features of the sending host, email header features, a parser for the email text, a JavaScript parser and features from images. Plain logistic regression gets 94 per cent and the full system 99.9. What accounts for the difference?

Remove components one at a time, cumulatively, and record the accuracy after each removal:

Table 12.1.: The lecture's ablation, cumulative from the top.

Removed	Accuracy	Drop
nothing (full system)	99.9 %	
spelling correction	99.0 %	0.9
sender host features	98.9 %	0.1
email header features	98.9 %	0.0
email text parser features	95.0 %	3.9
JavaScript parser	94.5 %	0.5
features from images	94.0 %	0.5

The text-parser features account for most of the gain. For a paper or a project report this is the result to state: the report then explains *why* the system works, not just that it does. If you are going to work on spam for years, it tells you where to invest. 1:18:21 The last question of the lecture asks how to choose the order of removal. There is no systematic way, Ng says, so if in doubt use the non-cumulative version: remove one component, put it back, remove the next.

1:17:03 He says that the two versions usually give similar insight. Usually they do, but there is a common case where they cannot, and Table 12.1 hints at it. Removing the header features costs nothing *after* the sender-host features are gone, which is what happens when two components carry largely the same information. The generator builds this case exactly. Two classes are Gaussian with a shared covariance, and there are four features: "sender host" and "email header" are separate measurements of the same signal (correlation 0.95), "text parser" is a strong independent feature and "JavaScript" a weak one. For this model the best classifier on any subset of features has accuracy $\Phi(\Delta/2)$, where Δ is the Mahalanobis distance between the class means on that subset, so every ablation can be computed in closed form.

Table 12.2.: Accuracy lost when a feature is removed, on a toy with two redundant features. The full model gets 0.788; with no features, 0.5.

Feature	Leave one out	Cumulative, sender host first	Cumulative, header first	Averaged over orders
sender host	0.001	0.001	0.055	0.070
email header	0.001	0.055	0.001	0.070
text parser	0.087	0.172	0.172	0.130
JavaScript	0.004	0.060	0.060	0.019

Leave-one-out says the sender-host and header features are each worth a tenth of a point, since either one can stand in for the other. Yet removing both together costs 5.6 points. Cumulative removal hands the whole 5.5 to whichever of the two is removed second, so the conclusion depends on an order that was chosen arbitrarily. The final column, the drop averaged over every removal order (the Shapley value from cooperative game theory), splits the credit evenly. Its entries add up to the full 28.8 points, as any single removal order's do. Leave-one-out's add up to only 9.4, because the credit the two redundant features share is counted for neither.

Order also matters for weak features: JavaScript looks worthless with everything else present and worth six points once the stronger features are gone. None of this makes ablation less useful. It means that when two components could plausibly duplicate each other, you should remove them together as well as separately before concluding either is dead weight. Error analysis has the same problem in reverse when stages interact: giving one stage perfect input may help little until another stage is fixed too.

> **In the official materials**
>
> This lecture has no class notes. Its slides are Ng's *Advice for applying machine learning*, and the course's error-analysis section handout covers the same pipeline example.

Part V.

Unsupervised Learning

13. Expectation-Maximization Algorithms

Lecture 13

Lecture video: Lecture 13: Expectation-Maximization Algorithms · 1:20:31

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

In unsupervised learning the training set is $\{x^{(1)}, \dots, x^{(m)}\}$ with no labels, and the job is to find structure in it. This lecture builds two algorithms that alternate between a guess about hidden structure and a fit given that guess. *k-means* guesses which cluster each point belongs to and moves each cluster's centre to the mean of its points. The *EM algorithm* for a mixture of Gaussians does the same with probabilities instead of hard assignments. The second half derives EM in general. Each E-step builds a lower bound on the log-likelihood that touches it at the current parameters, and each M-step maximises that bound, so the likelihood can never decrease. Jensen's inequality is the tool that makes the bound.

The same derivation carries the next lecture's factor analysis, which is why Ng gives it in full rather than stopping at the intuitive version.

13.1. k-means

2:19 Ng opens with an animation (Figure 13.1). Given unlabelled points that seem to form two groups, place two *cluster centroids* at random. Then repeat two steps: colour every point by the centroid it is closer to, and move each centroid to the mean of the points of its colour. After a few rounds the colours stop changing and the centroids stop moving, and the algorithm has converged.

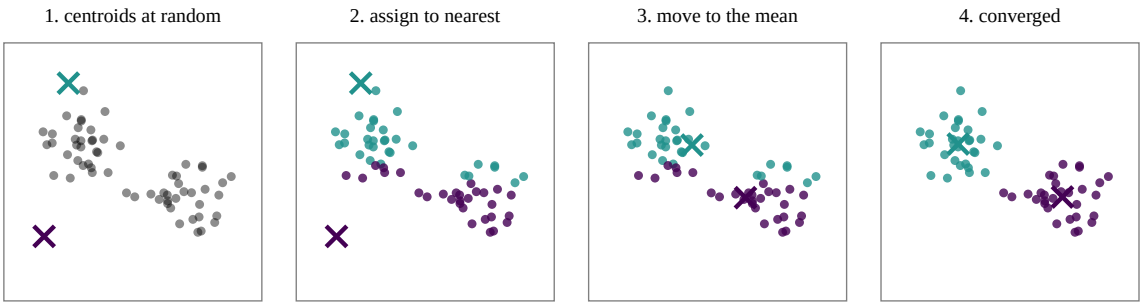


Figure 13.1.: k-means on 60 unlabelled points: two centroids placed badly, the first assignment, the first update, and the converged state.

5:42 Written out, the algorithm is:

1. Initialise the cluster centroids $\mu_1, \dots, \mu_k \in \mathbb{R}^n$.

2. Repeat until convergence:
   - for every i, set $c^{(i)} := \arg\min_j \|x^{(i)} - \mu_j\|^2$;
   - for every j, set $\mu_j := \dfrac{\sum_i 1\{c^{(i)} = j\}\, x^{(i)}}{\sum_i 1\{c^{(i)} = j\}}$.

The first inner step assigns each example to its nearest centroid. The second moves each centroid to the average of the examples assigned to it. 6:23 The animation drops the initial centroids at arbitrary positions, but Ng points out that the usual practice is to set them to k training examples chosen at random. In two dimensions this hardly matters, but in high dimensions a random vector tends to land far from all the data and capture no points.

The most common use Ng names is market segmentation: group a website's users into types that can be targeted differently, without anyone having labelled what the types are. The Google News clustering of stories from the first lecture is another.

13.2. The cost k-means descends

11:45 k-means is guaranteed to converge, and the proof is a cost function. Define the *distortion*

$$J(c, \mu) = \sum_{i=1}^{m} \left\| x^{(i)} - \mu_{c^{(i)}} \right\|^2, \tag{13.1}$$

the total squared distance from each point to the centroid it is assigned to. The assignment step minimises J over c with μ fixed, since each term is minimised separately by choosing the nearest centroid. The update step minimises J over μ with c fixed, because the point that minimises a sum of squared distances to a set of points is their mean. So k-means is coordinate descent on J, and J never increases. Since it is bounded below by zero, it must converge. The generator tracks J after every half-step of every run and confirms it never rises. In Figure 13.1 it falls from 476.9 to 31.3 in three rounds. On very large data sets, Ng adds, people often stop once the decrease in J has become small, without waiting for convergence.

8:49 A student asks whether the norm in the circulated notes is an L_1 norm. Ng says it is L_2, the default whenever a norm is written without a subscript, and adds that with or without the square "the algorithm is the same either way". For the assignment step that is right. Squaring is monotone on non-negative numbers, so the nearest centroid is the same under $\|\cdot\|$ and $\|\cdot\|^2$, and the generator confirms identical assignments for 200 random sets of centroids. For the update step it is not. The mean minimises the sum of *squared* distances. The point that minimises the sum of plain distances is a different one, the geometric median, so the convergence argument above only works for the squared cost. The generator finds random starts where moving to the means *raises* the unsquared cost: 26 of 2,000. So k-means does descend a cost, but it is the squared one, Equation 13.1.

13:33 Ng is most often asked how to choose k. He usually chooses it by hand, for two reasons. Unsupervised data is often genuinely ambiguous: the same picture can look like two clusters to one person and four to another. And criteria that pick k automatically, such as AIC and BIC, answer a question the application did not ask. The better guide is what the clusters are for. If the marketing team can run four campaigns but not a hundred, use four clusters. In image compression, choose k by how much compression you want.

15:41 k-means can stop at a local minimum of J, since coordinate descent only guarantees that J stops decreasing. The fix Ng gives is to run it many times from different random initialisations

and keep the run with the lowest J. On three well-separated clusters, the generator's 100 restarts, each initialised at three random training examples, ended at seven different final costs, from 92.6 to 378.2, and the best was reached 92 times. Taking the lowest cost over a handful of restarts is therefore cheap insurance.

13.3. Density estimation and anomalies

16:18 A related but different problem is *density estimation*: model $p(x)$ from unlabelled data. Ng's example comes from friends who worked on aircraft engines. As each engine comes off the assembly line, measure its vibration and the heat it produces. If a new engine's measurements fall where $p(x)$ is very small, flag it and send it for inspection before it is fitted to a plane. The same scheme finds cell towers whose network traffic has turned strange, and computers sending unusual traffic.

19:46 The interesting anomaly in his sketch is one where *neither* feature is unusual on its own. Many engines vibrate that much, and many run that hot, but few do both. Only a joint model of $p(x)$ catches it. And the data he draws are L-shaped, which no single textbook distribution fits. They look instead like two Gaussian blobs, perhaps engines from two production processes, which is the idea of the next section.

13.4. The mixture of Gaussians

24:23 The model introduces a *latent* random variable z, meaning one that exists but is never observed. For each example,

$$z^{(i)} \sim \text{Multinomial}(\phi), \qquad x^{(i)} \mid z^{(i)} = j \;\sim\; \mathcal{N}(\mu_j, \Sigma_j), \tag{13.2}$$

where $\phi_j = p(z^{(i)} = j)$, $\phi_j \geq 0$, $\sum_j \phi_j = 1$. In words: pick one of k Gaussians with probabilities ϕ, then draw x from it. The density of x is the mixture $p(x) = \sum_j \phi_j \, \mathcal{N}(x; \mu_j, \Sigma_j)$, which with enough components can take almost any shape, such as an L, or a narrow peak beside a wide hump (Figure 13.2, left).

This is Gaussian discriminant analysis (Section 5.3) with two cosmetic differences: z takes k values instead of two, and each component has its own covariance Σ_j by convention, where GDA shared one. The real difference is that GDA's labels $y^{(i)}$ were observed and $z^{(i)}$ are not. 28:12 If the $z^{(i)}$ *were* observed, maximum likelihood would give GDA's closed-form estimates, $\phi_j = \frac{1}{m}\sum_i 1\{z^{(i)} = j\}$ and μ_j the mean of the examples with $z^{(i)} = j$, and Σ_j their covariance. Not knowing which Gaussian each example came from is the whole difficulty.

13.5. EM for the mixture: soft k-means

30:54 EM resolves the difficulty with the same bootstrap as k-means. Guess the $z^{(i)}$, fit the parameters as if the guesses were labels, and repeat. The guesses are probabilities rather than hard labels.

E-step. For each i and j, compute the posterior probability that example i came from component j,

$$w_j^{(i)} := p(z^{(i)} = j \mid x^{(i)}; \phi, \mu, \Sigma) = \frac{\phi_j \, \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}{\sum_{l=1}^{k} \phi_l \, \mathcal{N}(x^{(i)}; \mu_l, \Sigma_l)}, \tag{13.3}$$

which is Bayes' rule, exactly as GDA used it to classify.

M-step. Re-estimate the parameters with the GDA formulas, with each indicator $1\{z^{(i)} = j\}$ replaced by $w_j^{(i)}$:

$$\phi_j := \frac{1}{m} \sum_i w_j^{(i)}, \qquad \mu_j := \frac{\sum_i w_j^{(i)} x^{(i)}}{\sum_i w_j^{(i)}}, \qquad \Sigma_j := \frac{\sum_i w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^\top}{\sum_i w_j^{(i)}}. \tag{13.4}$$

36:30 The replacement is principled: $w_j^{(i)}$ is the expected value of the indicator under the current posterior, since the expectation of an indicator is the probability of its event. 37:10 Ng's intuition is that this is k-means with *soft assignment*. k-means gives each point entirely to the nearest centroid, even when it is only slightly nearer. EM gives it, say, 0.8 to one Gaussian and 0.2 to the other, and each mean becomes a weighted average in which each point counts with that strength. A student asks how to think about the weights. Ng's reading is that each point is assigned with total strength one, split across the components. In the engine story, a point with a 10 per cent chance of coming from process one contributes a tenth of an engine to process one's Gaussian.

The analogy can be made exact. Give all components the same fixed covariance $\sigma^2 I$ and equal ϕ_j, and let $\sigma^2 \to 0$. Then the posterior in Equation 13.3 goes to 1 for the nearest mean and 0 for the others, and the mean update becomes k-means' update. In the generator's one-dimensional example, the largest gap between EM's soft assignments and k-means' hard ones is 3×10^{-11} once $\sigma^2 = 0.01$.

34:06 Ng describes the E-step for two classes as a sigmoid: the probability of the left Gaussian falls smoothly from one to zero across the gap between the means. That is GDA's result (Section 5.4), and it holds when the components *share* a covariance. With equal variances the posterior is exactly $1/(1 + e^{-(ax+b)})$, which the generator checks to 10^{-12}. With separate covariances, the convention the model was just given, $\log(w_1/w_2)$ is quadratic in x rather than linear, and the posterior need not be monotone. In Figure 13.2 the left component is narrow and the right one wide. The posterior of the left component rises to 0.955 near $x = -2.1$ and falls back to 0.003 at $x = -5$, because far out on either side the wide Gaussian is the likelier source. EM handles this without any change. It only means "points on the left go to the left cluster" is not what the E-step computes.

38:56 Once fitted, the mixture gives $p(x) = \sum_j p(x, z = j)$ for any new engine, and a low value flags an anomaly. Ng closes the section with a memory from a summer at AT&T Bell Labs, of a colleague who had just run EM on a mixture of three Gaussians for the first time and could not stop showing people what it did.

13.6. Jensen's inequality

48:07 The argument so far was heuristic: guess the latent variables, then pretend they were observed. To show that EM really is maximum-likelihood estimation, Ng needs Jensen's inequality. Let f be convex, meaning $f''(x) \geq 0$, and X a random variable. Then

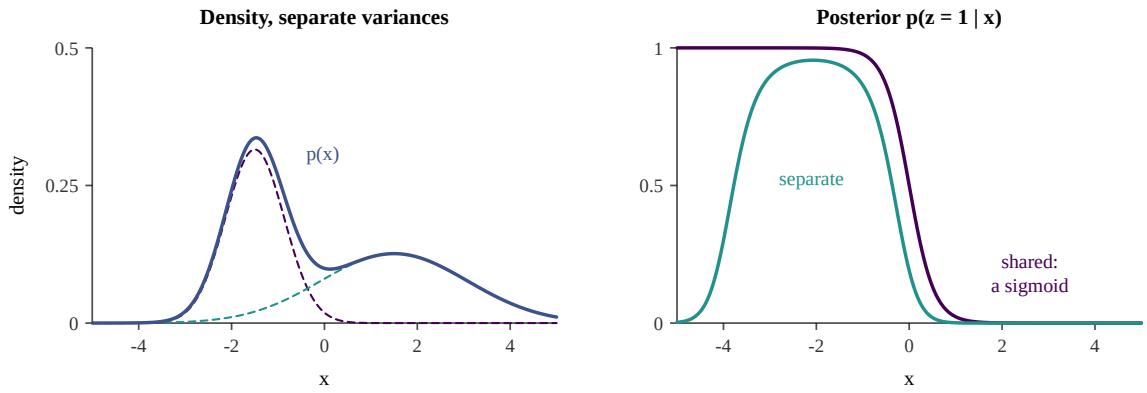


Figure 13.2.: A mixture of a narrow and a wide Gaussian: the components (dashed) and $p(x)$ (left), and the E-step posterior of the left component (right) with the variances shared and separate.

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)]. \tag{13.5}$$

His example takes $X = 1$ or $X = 5$, each with probability one half. Then $\mathbb{E}[X] = 3$, and $\mathbb{E}[f(X)]$ is the average of $f(1)$ and $f(5)$, which is the midpoint of the chord joining those two points of the curve (Figure 13.3). A convex function lies below its chords, so the chord's midpoint is above $f(3)$. Ng admits that he and his friends do not remember which way the inequality goes. They redraw this picture every time.

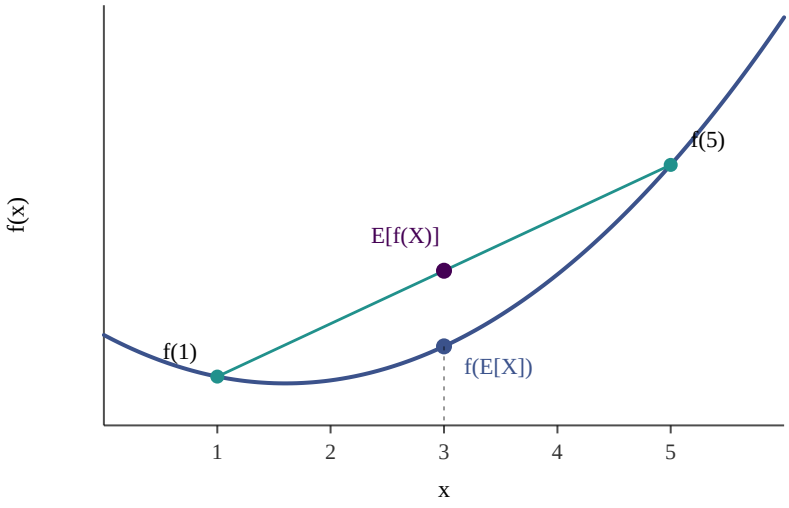


Figure 13.3.: Jensen's inequality: the chord's midpoint, $\mathbb{E}[f(X)]$, lies above the curve at the mean, $f(\mathbb{E}[X])$.

53:23 EM also needs the equality case. If f is *strictly* convex, $f'' > 0$, then equality holds only if X is constant, i.e. $X = \mathbb{E}[X]$ with probability one, which is the measure theorist's way of saying it. A straight line is convex but not strictly, and for it equality holds for any X. 54:47 A student asks about a curve with $f(1) = f(3)$. Equal end values alone change nothing: the chord is then flat, and a strictly convex curve still dips below it in between. Equality for a non-constant X needs the function to be straight between the values X takes, for instance flat between 1 and 3, and such a function is not strictly convex. The strictness is the hypothesis that rules this out.

For EM the inequality is used upside down. A concave function is the negative of a convex one ($f'' \leq 0$), and for concave f, $f(\mathbb{E}[X]) \geq \mathbb{E}[f(X)]$, with equality for strictly concave f only when X is constant. The function it is applied to is log, which is strictly concave.

13.7. EM as maximising a lower bound

57:30 Take any model $p(x, z; \theta)$ with z latent and θ all the parameters (for the mixture, ϕ, μ, Σ). The log-likelihood of the data marginalises z out:

$$\ell(\theta) = \sum_{i=1}^{m} \log p(x^{(i)}; \theta) = \sum_{i=1}^{m} \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta).$$

The sum inside the logarithm is what makes this hard. 1:19:12 A student asks at the end why we do not just maximise ℓ directly. Setting its derivatives to zero gives equations with no closed-form solution for the mixture, or for factor analysis next time, whereas each step of EM is in closed form. Gradient ascent on ℓ is possible, but it needs a learning rate, and EM needs none.

59:58 Ng gives the picture first (Figure 13.4). The E-step builds a lower bound on ℓ that has two properties: it lies below ℓ everywhere, and it equals ℓ at the current parameters $\theta^{(t)}$. The M-step jumps to the maximum of the bound. At that point ℓ is at least as high as the bound, which is at least as high as it was at $\theta^{(t)}$, where it equalled $\ell(\theta^{(t)})$. Repeat, and ℓ climbs until EM reaches a local optimum. As with k-means, which local optimum it finds depends on the starting point.

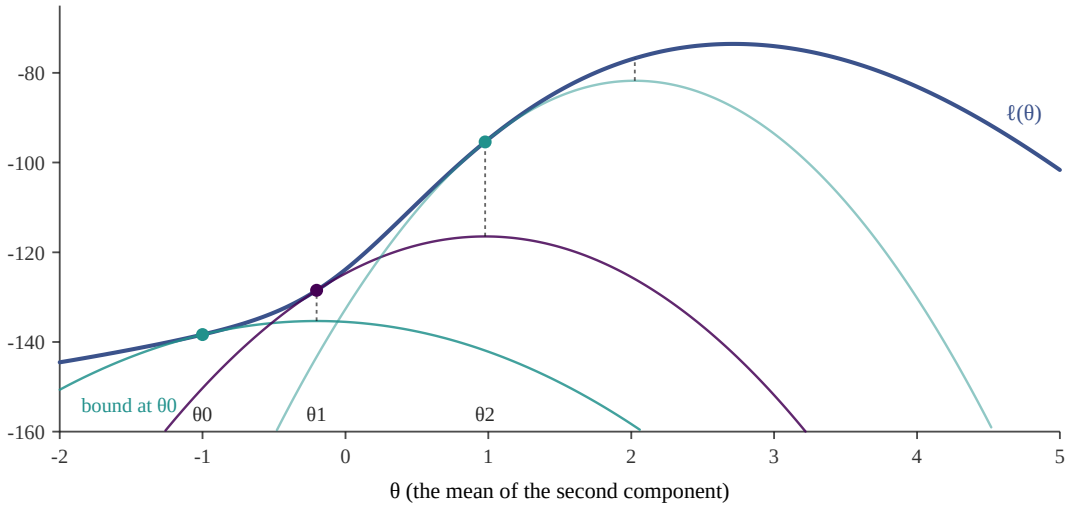


Figure 13.4.: EM on a one-parameter mixture: the log-likelihood $\ell(\theta)$ and the lower bounds built at the first three iterates. Each touches ℓ at the current iterate; the M-step moves to its peak.

1:04:21 To build the bound, pick for each example any distribution Q_i over the values of $z^{(i)}$, multiply and divide by it, and read the inner sum as an expectation over $z^{(i)} \sim Q_i$:

$$\begin{aligned}\ell(\theta) &= \sum_i \log \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} = \sum_i \log \mathbb{E}_{z^{(i)} \sim Q_i}\left[\frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}\right] \\ &\geq \sum_i \mathbb{E}_{z^{(i)} \sim Q_i}\left[\log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}\right] = \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}.\end{aligned} \tag{13.6}$$

The inequality is Jensen's for the concave log: the log of an expectation is at least the expectation of the log. The right-hand side, viewed as a function of θ with the $x^{(i)}$ fixed and the $z^{(i)}$ summed out, is the green curve of Ng's picture. Today it is usually called the *evidence lower bound*, or ELBO.

1:11:54 The bound holds for every choice of Q_i. The E-step chooses the Q_i that make it *tight* at the current θ. By the equality case of Jensen's inequality, that requires the random variable inside to be constant, so $p(x^{(i)}, z^{(i)}; \theta)/Q_i(z^{(i)})$ must not depend on $z^{(i)}$. In other words, $Q_i(z^{(i)}) \propto p(x^{(i)}, z^{(i)}; \theta)$. Normalising to sum to one gives

$$Q_i(z^{(i)}) = \frac{p(x^{(i)}, z^{(i)}; \theta)}{\sum_z p(x^{(i)}, z; \theta)} = p(z^{(i)} \mid x^{(i)}; \theta),$$

the posterior of the latent variable given the data, which is exactly the $w_j^{(i)}$ of Equation 13.3. The general algorithm is then:

1. *E-step:* for each i, set $Q_i(z^{(i)}) := p(z^{(i)} \mid x^{(i)}; \theta)$.
2. *M-step:* set $\theta := \arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \dfrac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$.

For the mixture of Gaussians, the M-step's maximisation is the weighted GDA fit of Equation 13.4, so the heuristic of the previous section was EM all along.

The generator checks each step of this argument on the one-parameter model of Figure 13.4, a mixture of $\mathcal{N}(0, 1)$ and $\mathcal{N}(\theta, 1)$ with equal weights, where only the second mean is unknown. At each of the first four iterates, the bound equals ℓ to 10^{-9}, has the same slope (so the curves are tangent rather than merely touching), and lies below ℓ at every point of a fine grid. Its maximiser found by grid search agrees with the closed-form M-step. From $\theta^{(0)} = -1$ the iterates run $-0.20, 0.98, 2.03, 2.47$ and converge to 2.72, and ℓ rises at every step, from -138.4 to -73.5.

There is a caveat about "maximum likelihood" for mixtures. When each component has its own variance, the likelihood has no maximum at all. Centre one component exactly on a training point and shrink its variance. That point's density grows without bound while the other components keep the remaining points' densities finite, so $\ell \to \infty$. The generator shows ℓ rising without limit as the variance falls through 10^{-8}, 10^{-16}, … 10^{-128}. EM does not usually find these degenerate solutions, because it climbs from where it starts, but a component that captures a single point can drift into one. In practice this is guarded against by restarting, or by putting a floor or a small prior under the covariances. "Converges to a local optimum" is the accurate claim, and for mixtures the global optimum is not a useful target.

> **In the official notes**
>
> k-means and its convergence as coordinate descent are in cs229-notes7a. The mixture of Gaussians and its EM updates, including the Σ_j step the lecture skipped, are in cs229-notes7b. Jensen's inequality, the general derivation, the proof that ℓ increases monotonically, and the view of EM as coordinate ascent on the lower bound are in cs229-notes8, sections 1

to 3. The original general treatment is Dempster, Laird and Rubin (1977), and the k-means iteration is Lloyd's (1982).

14. EM Algorithm and Factor Analysis

Lecture 14

Lecture video: Lecture 14: EM Algorithm and Factor Analysis · 1:19:48

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Factor analysis is a density model for data with many more dimensions than examples. For example, a hundred temperature sensors give a hundred-dimensional reading, but you may have only thirty days of data. An ordinary Gaussian cannot be fitted to such data, because its maximum-likelihood covariance is singular. The model assumes that each observation is a low-dimensional hidden cause z, mapped linearly into the high-dimensional space, plus independent noise in every coordinate. Because z is hidden, the model is fitted with EM, and doing so means working out the conditional distributions of a joint Gaussian.

The lecture first closes off the previous one. It shows how the abstract E- and M-steps of Section 13.7 become code for the mixture of Gaussians, and it recasts EM as coordinate ascent. It then spends the rest of the hour on factor analysis.

14.1. From the general algorithm to the mixture updates

3:00 The general M-step says to maximise the lower bound over θ with the Q_i held fixed. For the mixture of Gaussians, $Q_i(z^{(i)} = j)$ is the responsibility $w_j^{(i)} = p(z^{(i)} = j \mid x^{(i)}; \phi, \mu, \Sigma)$ that the E-step computes and stores. The joint density factorises as $p(x^{(i)} \mid z^{(i)} = j)\, p(z^{(i)} = j) = \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)\, \phi_j$. So the quantity the M-step maximises is a double sum that can be written out in full:

$$J(\phi, \mu, \Sigma) = \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \frac{\dfrac{1}{(2\pi)^{n/2} |\Sigma_j|^{1/2}} \exp\!\Big(-\tfrac{1}{2}(x^{(i)} - \mu_j)^\top \Sigma_j^{-1} (x^{(i)} - \mu_j)\Big)\, \phi_j}{w_j^{(i)}}. \tag{14.1}$$

Ng's point is that this objective is what gives you the updates. The rule "move each mean to the weighted average of the points" was an intuition in the last lecture, and here it is derived. Differentiating Equation 14.1 with respect to μ_j and setting the result to zero gives

$$\mu_j = \frac{\sum_i w_j^{(i)} x^{(i)}}{\sum_i w_j^{(i)}},$$

and the same procedure gives Σ_j.

For ϕ, "differentiate and set to zero" does not quite work, and the board skips over why. The only terms in ϕ are $\sum_i \sum_j w_j^{(i)} \log \phi_j$, whose derivative with respect to ϕ_j is $\sum_i w_j^{(i)}/\phi_j$. That derivative is positive for every value of ϕ_j, so it never vanishes. Taken alone, the objective wants every ϕ_j to be infinite. What stops it is the constraint $\sum_j \phi_j = 1$, so the step needs a Lagrange multiplier. With the multiplier, the solution is $\phi_j = \frac{1}{m}\sum_i w_j^{(i)}$. The generator confirms this on a three-component one-dimensional mixture. At the closed-form M-step, the unconstrained derivative equals exactly $m = 200$ for all three components (the multiplier), and none of 300 random perturbations of (ϕ, μ, σ^2) that stay on the simplex raises J.

14:11 Ng then gives a second view of EM that the factor-analysis derivation leans on. Treat the lower bound as a function of *both* arguments,

$$J(\theta, Q) = \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \;\le\; \ell(\theta) \quad \text{for every } \theta \text{ and } Q.$$

The E-step maximises J over Q with θ fixed, and the maximum is the choice that makes $J = \ell$. The M-step maximises J over θ with Q fixed. EM is therefore *coordinate ascent* on J: it maximises over one block of variables, then the other, and alternates. This view also tells you how to check that an implementation is working. J cannot decrease in either step, and since it equals $\ell(\theta)$ after every E-step, the log-likelihood you print at each iteration must not decrease either. If it ever goes down, there is a bug. On the mixture above, the generator asserts that $J = \ell$ after each E-step to 10^{-8}, and that ℓ never falls over 40 iterations: from -544.2 it reaches -388.9 after three iterations and -358.1 at the end.

When z is continuous, as it will be in factor analysis, every step of the derivation in Section 13.7 goes through with the sum over $z^{(i)}$ replaced by an integral. Jensen's inequality does not care which kind of expectation it is applied to.

14.2. Fewer examples than dimensions

17:46 A mixture of Gaussians suits data like Ng's two-dimensional picture with a hundred points: $m \gg n$, and each cluster has plenty of points to estimate its covariance. Factor analysis is for the opposite regime, where $m \approx n$ or $m \ll n$. Ng's running example is a Stanford PhD student's project with temperature sensors placed around campus buildings: a hundred sensors and thirty days of readings, so $n = 100$ and $m = 30$. The goal was anomaly detection by density estimation, as in Section 13.3. A window left open on a winter day produces a reading with low $p(x)$, and in practice the point was saving energy rather than detecting fires. To do this you need a joint density over all hundred sensors, because neighbouring sensors are strongly correlated, and you have thirty examples to fit it with.

22:01 The obvious model is a single Gaussian with its maximum-likelihood estimates $\mu = \frac{1}{m}\sum_i x^{(i)}$ and $\Sigma = \frac{1}{m}\sum_i (x^{(i)} - \mu)(x^{(i)} - \mu)^\top$. If $m \le n$, this Σ is singular. The reason is that it is a sum of m rank-one matrices whose vectors $x^{(i)} - \mu$ sum to zero, so its rank is at most $m - 1$. The generator checks the exact statement on random data for $n = 2, 5, 10$ and every m from 2 to $n + 2$: the rank is $\min(m - 1, n)$, so the estimate is invertible exactly when $m \ge n + 1$. A singular Σ breaks the density in two ways. The normaliser contains $|\Sigma|^{1/2} = 0$ and the exponent contains Σ^{-1}, which does not exist, so evaluating $p(x)$ gives $0/0$ at best. Ng's picture with $m = n = 2$ shows the second problem (Figure 14.1, first panel). The fitted Gaussian has its contours squashed to zero width along the segment joining the two points. All of its probability

mass lies on that line, so any test point slightly off the line gets density zero. That is a numerical failure, and it is also a bad model of the data.

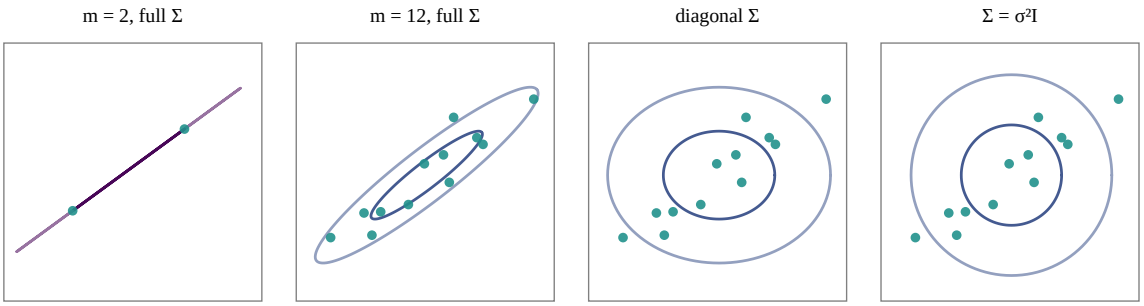


Figure 14.1.: Maximum-likelihood Gaussians on two-dimensional data, with the one- and two-standard-deviation contours. With two points (left) the full covariance is singular and the Gaussian collapses onto the line through them. With twelve strongly correlated points (sample correlation 0.93) the full covariance follows the tilt of the data, the diagonal covariance gives axis-aligned ellipses that miss it, and the spherical covariance gives circles.

26:53 Ng adds that one of the earliest applications of factor analysis had the same shape: psychological testing, with a hundred questionnaire items and thirty subjects, where the question is how attributes like verbal and mathematical ability are correlated.

14.3. Restricting the covariance

28:15 One way out is to fit fewer parameters. If Σ is constrained to be diagonal, the maximum-likelihood estimate is what you would guess: each feature gets its own sample variance, $\Sigma_{jj} = \frac{1}{m}\sum_i (x_j^{(i)} - \mu_j)^2$. That makes n parameters in place of about $n^2/2$, and they can be fitted from any $m \geq 2$. The contours are axis-aligned ellipses. The price is that the model assumes the features are uncorrelated, and for a hundred sensors in one room that is plainly false (Figure 14.1, third panel).

31:36 A stronger restriction is $\Sigma = \sigma^2 I$, with one parameter and circular contours. Its estimate is the average squared deviation over every coordinate of every example, which is the mean of the diagonal model's n variances. A student asks why anyone would bother with a model that is worse still. Ng's answer is that it is a building block: factor analysis uses a diagonal covariance for its noise, not for the data.

34:38 Another student asks about the standard fix, a Wishart prior on Σ. In practice this means adding a constant λ to the diagonal of the maximum-likelihood estimate. Ng's verdict is that it removes the singularity "in a technical way" but is not the best model for many data sets. The generator puts that verdict to a test at the end of the chapter (Section 14.6). What Ng wants instead is a model that captures the main correlations, such as the room warming and cooling as a whole or one side heating up under the lamps, without paying for a full covariance matrix.

He follows this with an aside about small data that is worth keeping. The large technology companies, he says, all work on the same few big-data problems (translation, speech, face recognition), while many real applications in healthcare and manufacturing have a hundred or a thousand examples. The best algorithm for a thousand examples is often not the best for a

million, and the field does not yet know how to make one algorithm work along the whole range. He mentions using a modified factor analysis on a manufacturing problem that same week.

14.4. The factor analysis model

40:32 Like the mixture of Gaussians, the model is a joint density $p(x, z) = p(x \mid z)\, p(z)$ with z hidden. This time z is continuous and of lower dimension than x:

$$\begin{aligned} z &\sim \mathcal{N}(0, I), \qquad z \in \mathbb{R}^d,\ d < n, \\ x &= \mu + \Lambda z + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \Psi), \end{aligned} \tag{14.2}$$

with z and ε independent. The parameters are the mean $\mu \in \mathbb{R}^n$, the *factor loading* matrix $\Lambda \in \mathbb{R}^{n \times d}$, and a **diagonal** $\Psi \in \mathbb{R}^{n \times n}$. Equivalently, $x \mid z \sim \mathcal{N}(\mu + \Lambda z, \Psi)$.

43:30 Ng's reading of the model uses the room. Suppose three strong forces drive the temperatures in the room: the temperature outside, the lights on the left side, and the lights on the right. Then $d = 3$, and z holds those three numbers. Each sensor reads a linear combination of the three, weighted by where it sits, so a sensor near the door leans on the outdoor term. On top of that each sensor adds noise of its own, such as a gust of wind or a bad thermistor. The assumption that Ψ is diagonal says that this noise is independent across sensors once the shared factors are accounted for. All correlation between sensors has to come through Λz.

46:21 The lecture's example makes this concrete with $d = 1$, $n = 2$, $m = 7$, $\Lambda = (2, 1)^\top$, $\mu = 0$ and $\Psi = \mathrm{diag}(1, 2)$ (Figure 14.2). Seven draws of z fall on the real line. Multiplying by Λ places them on the line $x_2 = x_1/2$ in the plane. Around each of these points, the noise ε has axis-aligned elliptical contours, taller than wide in the ratio $\sqrt{2} : 1$. Drawing one sample from each gives the observed points. Only those crosses are data. The z values, and the points on the line, are latent.

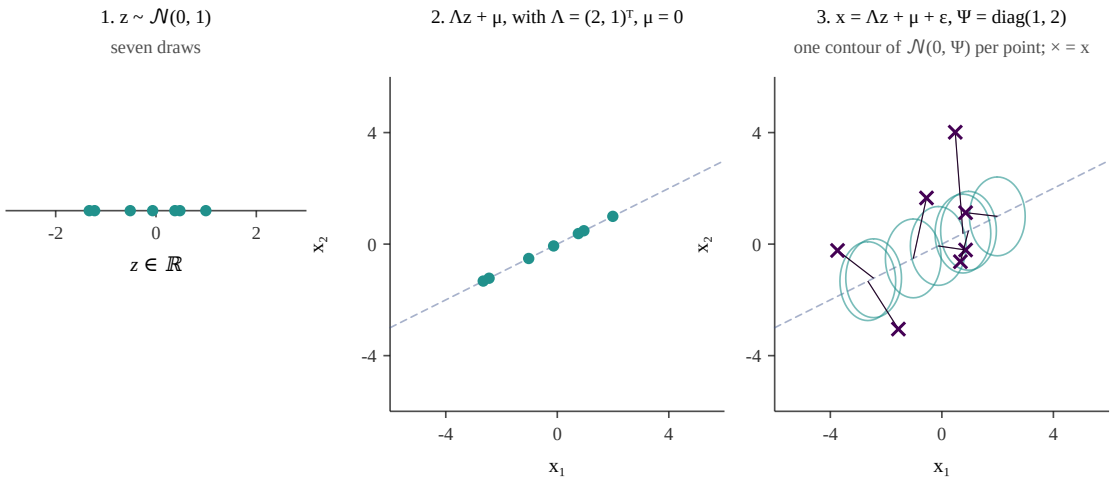


Figure 14.2.: The factor-analysis model as a generative process, with the lecture's parameters. Seven draws of z (left) map through $\Lambda z + \mu$ onto a line in the plane (centre). Independent axis-aligned noise with covariance Ψ moves each one to the observed x, the crosses (right). The thin lines join each latent point to its observation.

51:05 Ng repeats the construction with $d = 2$ and $n = 3$, carrying the five sampled z points on a sheet of cardboard and holding it up into three-dimensional space. The data then lie close to a

two-dimensional "pancake" in $\mathbb{R}^3$, with some fuzz off it. Factor analysis takes data in a hundred dimensions and models it as lying near a three-dimensional subspace, plus independent noise in each coordinate. 54:57 Asked what happens when the data do not lie near a subspace, he concedes that the model is then not the best one. He adds that in a hundred dimensions you rarely know, because you cannot look.

14.5. Marginals and conditionals of Gaussians

56:43 Fitting the model needs two facts about multivariate Gaussians. Partition a Gaussian vector into blocks $x_1 \in \mathbb{R}^r$ and $x_2 \in \mathbb{R}^s$:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right).$$

The first fact is that the marginal of x_1 is $\mathcal{N}(\mu_1, \Sigma_{11})$. You can prove it by integrating x_2 out of the joint density, but the result is the obvious one: the mean and covariance of x_1 are the corresponding blocks. The second fact is that the conditional is also Gaussian,

$$x_1 \mid x_2 \sim \mathcal{N}(\mu_{1|2}, \Sigma_{1|2}), \qquad \mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2), \qquad \Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. \tag{14.3}$$

This one is less obvious, and the lecture states it without proof. The conditional mean is a linear regression of x_1 on x_2. The conditional covariance does not depend on the value of x_2 that was observed.

1:02:32 Applying this to factor analysis starts with the observation that (z, x) is jointly Gaussian, since x is a linear function of the Gaussians z and ε. Its mean is $(0, \mu)$. Ng derives the lower-right block of the covariance and leaves the others to the notes. Expanding

$$\mathbb{E}[(x-\mu)(x-\mu)^\top] = \mathbb{E}[(\Lambda z + \varepsilon)(\Lambda z + \varepsilon)^\top] = \Lambda\,\mathbb{E}[zz^\top]\,\Lambda^\top + \mathbb{E}[\varepsilon\varepsilon^\top] = \Lambda\Lambda^\top + \Psi.$$

Here the cross terms vanish because z and ε are independent with mean zero. The same expansion gives $\mathbb{E}[z(x-\mu)^\top] = \Lambda^\top$, so

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ \mu \end{bmatrix}, \begin{bmatrix} I & \Lambda^\top \\ \Lambda & \Lambda\Lambda^\top + \Psi \end{bmatrix} \right). \tag{14.4}$$

For the lecture's example, Equation 14.4 gives the covariance of (z, x_1, x_2) as rows $(1, 2, 1)$, $(2, 5, 2)$ and $(1, 2, 3)$. A Monte Carlo estimate from 200,000 samples in the generator agrees to within 0.015 in every entry.

By the marginal rule, $x \sim \mathcal{N}(\mu, \Lambda\Lambda^\top + \Psi)$. This explains why the model escapes the singularity: Ψ is diagonal with positive entries, so $\Lambda\Lambda^\top + \Psi$ is positive definite however small m is. The covariance has $nd + n$ free parameters: a rank-d part that carries the correlations, plus a diagonal part. 1:10:09 The log-likelihood $\sum_i \log \mathcal{N}(x^{(i)}; \mu, \Lambda\Lambda^\top + \Psi)$ is perfectly well defined, but setting its gradient to zero has no closed-form solution in Λ and Ψ. That is why the lecture turns to EM.

14.6. EM for factor analysis

1:11:03 In the mixture of Gaussians, Q_i was a list of k numbers. Here $z^{(i)}$ is continuous, so the question is how to represent Q_i in a computer. The answer is that $Q_i(z^{(i)}) = p(z^{(i)} \mid x^{(i)})$ is Gaussian, by Equation 14.3 applied to Equation 14.4, so it is stored as a mean vector and a covariance matrix:

$$\mu_{z^{(i)}|x^{(i)}} = \Lambda^\top(\Lambda\Lambda^\top + \Psi)^{-1}(x^{(i)} - \mu), \qquad \Sigma_{z^{(i)}|x^{(i)}} = I - \Lambda^\top(\Lambda\Lambda^\top + \Psi)^{-1}\Lambda. \tag{14.5}$$

The generator checks Equation 14.5 against Bayes' rule applied directly, multiplying the prior on z by the likelihood of x on a grid of 12,000 points. For three different observations in the lecture's example, the two agree to 10^{-6} in mean and variance. As the formula says, the posterior variance is the same for every observation: $1/(1 + 4 + \frac{1}{2}) = 0.1818$. In code, the $n \times n$ inverse is avoided with the matrix-inversion lemma, $\Sigma_{z|x} = (I + \Lambda^\top\Psi^{-1}\Lambda)^{-1}$, which is $d \times d$.

1:13:39 The M-step derivation is long, and Ng gives only its two tricks. The first is to recognise expectations instead of expanding integrals. The objective contains terms like $\int Q_i(z^{(i)})\, z^{(i)}\, dz^{(i)}$. Substituting the Gaussian formula for Q_i and integrating is the wrong approach. The integral is $\mathbb{E}_{z^{(i)}\sim Q_i}[z^{(i)}]$, which is $\mu_{z^{(i)}|x^{(i)}}$, already computed in the E-step. 1:18:13 The second is to expand a Gaussian density when a logarithm sits in front of it. Then the log cancels the exponential and leaves a quadratic in the parameters, whose expectation under Q_i is again a quadratic that can be maximised in closed form. The resulting update for Λ is

$$\Lambda = \left(\sum_{i=1}^{m}(x^{(i)} - \mu)\,\mu_{z^{(i)}|x^{(i)}}^\top\right)\left(\sum_{i=1}^{m}\left[\mu_{z^{(i)}|x^{(i)}}\mu_{z^{(i)}|x^{(i)}}^\top + \Sigma_{z^{(i)}|x^{(i)}}\right]\right)^{-1}, \tag{14.6}$$

with μ the sample mean. Ψ is the diagonal of a similar expression, given in the official notes. Equation 14.6 has the form of the normal equations for regressing x on z, except that the unknown $z^{(i)}$ are replaced by their posterior moments. The bracket in the second factor is $\mathbb{E}[z^{(i)}z^{(i)\top}]$ under Q_i. That is the mean's outer product *plus* the posterior covariance, because $\mathbb{E}[zz^\top]$ is not $\mathbb{E}[z]\,\mathbb{E}[z]^\top$.

The official notes warn that dropping $\Sigma_{z|x}$ here is a common mistake, which treats the posterior mean as if it were the true z. They do not say what the mistake costs, so the generator runs both versions (Figure 14.3). The data are 30 examples in $n = 100$ dimensions, drawn from a three-factor model; the sample covariance has rank 29, as Section 14.2 says it must. Correct EM increases ℓ on every one of 200 iterations, from -5744.2 at the start to -3254.7 after 10 iterations and -3203.0 at 200. Over the last 50 iterations it is still gaining 1.3 nats. The shortcut version is deceptive. It climbs *faster* at first and reaches -3221.0 at iteration 4, ahead of the correct algorithm. Then the likelihood falls on every remaining iteration, 196 of 200 in all. Correct EM overtakes it at iteration 38, and the shortcut finishes at -3235.3. The shortcut is not coordinate ascent on any bound, so the monitoring rule of Section 14.1 is exactly what catches it: an implementation whose log-likelihood ever decreases has a bug, even if its first few iterations look better than the correct one.

The same run tests Ng's remark about the Wishart fix from Section 14.3. On 300 held-out examples, factor analysis scores an average log-likelihood of -129.0 per example. The full Gaussian with λI added scores -138.5, even though λ was chosen from a grid using the test set itself, which is an unfair advantage. The diagonal Gaussian scores -198.8. The comparison does favour factor analysis, because the data were generated from a factor model. What it shows is narrower than "factor analysis is better" in general. When the correlations really are low-rank,

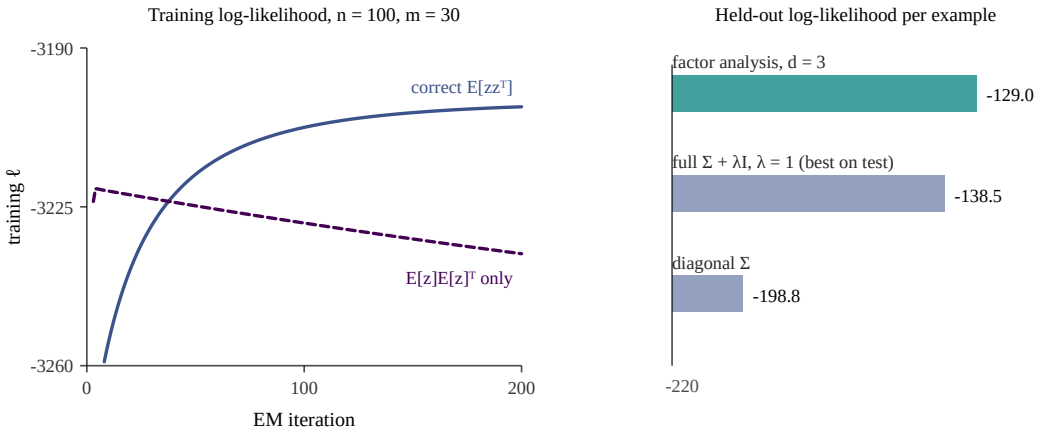


Figure 14.3.: EM for factor analysis with $n = 100$, $m = 30$, $d = 3$. Left: training log-likelihood per iteration for the correct M-step and for the shortcut that uses $\mathbb{E}[z]\mathbb{E}[z]^\top$ in place of $\mathbb{E}[zz^\top]$. The first few iterations climb from far below and are clipped. The shortcut leads early, then declines on every iteration. Right: average log-likelihood of 300 held-out examples under factor analysis, a full Gaussian with λI added to its covariance (with λ chosen on the test set itself, which favours it), and a diagonal Gaussian.

adding λI makes the covariance invertible without capturing that structure, and the diagonal model throws the correlations away entirely.

> **In the official notes**
>
> Restrictions on Σ, the marginal and conditional formulas with a derivation of the conditional, the joint distribution of (z, x), and the full M-step for μ, Λ and Ψ are in cs229-notes9, sections 1 to 4. The mixture-of-Gaussians M-step with the Lagrange multiplier for ϕ, and the coordinate-ascent view of EM, are in cs229-notes8, sections 2 and 3. The EM algorithm for factor analysis was given by Rubin and Thayer (1982).

15. PCA and ICA

Lecture 15

Lecture video: Lecture 15: PCA and ICA · 1:18:35

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Principal components analysis (PCA) reduces n-dimensional data to $k \ll n$ dimensions by projecting it onto the k directions along which it varies most. Those directions are the top k eigenvectors of the data's covariance matrix. Like factor analysis in the previous lecture, PCA assumes the data lie near a low-dimensional subspace. Unlike factor analysis it has no probabilistic model: it does not define $p(x)$, it just finds the subspace. Ng spends as long on when *not* to use PCA as on how it works. The last quarter of the lecture sets up independent components analysis (ICA), which looks for statistically independent directions instead of high-variance ones. The algorithm itself is in lecture 16.

15.1. One-dimensional data living in two dimensions

2:09 Ng's first example is a school that measured its children's heights twice in the same month, once in centimetres and once in inches, with each value rounded to a whole unit. Plotted against each other, the two numbers lie almost exactly on a line. The data have $n = 2$, but they are really one-dimensional: a child's height plus a little rounding noise perpendicular to the line. PCA's job is to find that line, the principal axis of variation, and replace each point by its position along it.

Because centimetres and inches measure the same thing, he offers two less contrived examples. In the first, two vibration sensors made by different manufacturers sit on different parts of a factory machine. Both measure how hard the machine is shaking, which is useful for preventive maintenance, since the vibration pattern changes before a machine fails. 5:50 In the second, from his autonomous-helicopter work, each pilot is scored on skill and on enjoyment of flying. The two might both reflect one underlying aptitude. Going from two dimensions to one is only for the whiteboard. In practice, Ng says, PCA takes something like 10,000 dimensions to 100 or 1,000.

7:29 Before running PCA, the data are standardised. Subtract the mean of each feature, then divide each feature by its standard deviation, so that every feature has mean 0 and variance 1. Section 15.4 explains why both steps matter.

15.2. Two definitions of the best direction

10:00 Ng draws two candidate lines through the standardised height data: the 45-degree line, which is clearly right, and the perpendicular line, which is clearly wrong. He then gives two ways to say what makes the first one right (Figure 15.1). Project every point orthogonally onto a line. For the good line, the projection segments are short, so the sum of squared distances from the points to their projections is small. The projected points are also spread out along the line, while on the bad line they bunch together near the origin. So PCA could be defined as *minimising the squared projection distance* or as *maximising the variance of the projections*.

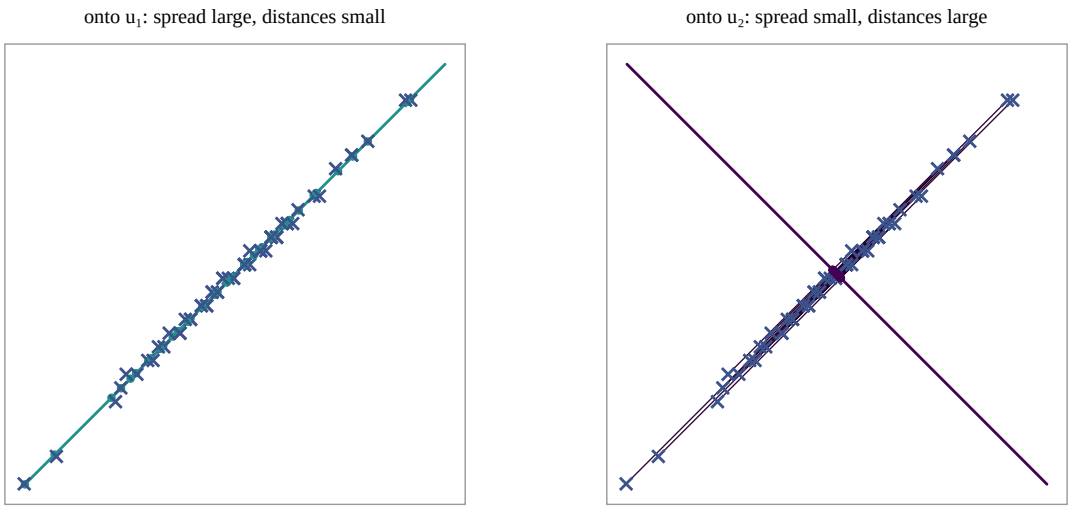


Figure 15.1.: Sixty standardised height measurements (crosses) projected orthogonally onto the first principal direction u_1 (left) and onto the perpendicular direction u_2 (right). The projections are the dots, joined to their points by thin segments. On u_1 the projections spread out and the segments are short; on u_2 the projections collapse onto the origin and the segments are long.

12:00 The two definitions are the same, and the reason is Pythagoras. For a unit vector u and a centred point $x^{(i)}$, the squared length of the point splits into the squared length of its projection plus the squared distance to it:

$$\|x^{(i)}\|^2 = (u^\top x^{(i)})^2 + \|x^{(i)} - (u^\top x^{(i)})\,u\|^2 .$$

The left side does not depend on u. Averaging over the data, projected variance plus mean squared distance is a constant, so maximising one minimises the other. The generator checks this at 3,600 directions around the half-circle. The two terms sum to the same constant to 10^{-12}, and the direction with the largest variance is exactly the direction with the smallest distance. On the height data that direction is 45.0°. The projected variance there is 1.9986 and the mean squared distance is 0.0014, out of a total of 2, because the correlation between the two measurements is 0.9986. Ng leaves the proof to the homework and recalls a paper that collected many equivalent derivations of PCA. Historically these two are the founding pair: Pearson (1901) posed the problem as fitting the closest line or plane to a set of points, and Hotelling (1933) posed it as finding components that account for as much of the variance as possible.

15.3. The principal direction is an eigenvector

13:14 Ng derives PCA from the variance definition. For a unit vector u, the projection of $x^{(i)}$ onto it has length $u^\top x^{(i)}$, so the problem is

$$\max_{u:\ \|u\|=1} \frac{1}{m}\sum_{i=1}^{m}\left(u^\top x^{(i)}\right)^2 = \max_{u:\ \|u\|=1} u^\top\Big(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)\top}\Big)u = \max_{u:\ \|u\|=1} u^\top \Sigma\, u, \tag{15.1}$$

where Σ is the covariance matrix of the standardised data. Ng apologises that the symbol is the same as the one for summation. 19:13 The maximiser is the *principal eigenvector* of Σ, the one with the largest eigenvalue. The argument uses a Lagrangian, $\mathcal{L}(u,\lambda) = u^\top\Sigma u - \lambda(u^\top u - 1)$. Setting its gradient in u to zero gives $\Sigma u = \lambda u$, so any stationary point of the constrained problem is an eigenvector. At such a point the objective equals $u^\top\Sigma u = \lambda u^\top u = \lambda$, so the best eigenvector is the one with the largest eigenvalue, and that eigenvalue is the variance captured. The generator checks this on a random 2×2 covariance by finding every stationary point of $u^\top\Sigma u$ around the half-circle of directions. It finds two, each lies along an eigenvector to within $0.02°$, and the value of the objective there matches the eigenvalue to 10^{-6}.

22:11 To keep k dimensions, take the top k eigenvectors $u_1,\dots,u_k$. The new representation of $x^{(i)}$ is the vector of its projections onto them:

$$y^{(i)} = \begin{bmatrix} u_1^\top x^{(i)} \\ \vdots \\ u_k^\top x^{(i)} \end{bmatrix} \in \mathbb{R}^k, \qquad x^{(i)} \approx y_1^{(i)}u_1 + \cdots + y_k^{(i)}u_k \in \mathbb{R}^n. \tag{15.2}$$

The second formula decompresses: it maps the k numbers back to a point in $\mathbb{R}^n$ on the k-dimensional subspace. For the heights, $y^{(i)}$ is a single cleaned-up estimate of each child's height, and mapping it back puts every child exactly on the line. Because Σ is symmetric, its eigenvectors can always be chosen orthogonal, so $u_1,\dots,u_k$ is an orthonormal basis of the subspace. 47:11 A student asks whether the test set needs its own eigenvectors. It does not. The u_j are computed once, on the training set, and the same ones are used to map every future example.

15.4. Pre-processing and the choice of k

28:07 A student asks why the pre-processing matters, and Figure 15.2 shows both cases. If the data are not centred, $\frac{1}{m}\sum_i x^{(i)}x^{(i)\top}$ is a second moment about the origin, not a covariance. Its top eigenvector points from the origin towards the cloud, not along it. In the generator it lies within $2°$ of the direction of the data's mean. If the features are not scaled, a feature measured in small units has numerically large variance and dominates. Ng's example is centimetres against kilometres. The heights show the effect in a milder form. Centimetre values are 2.54 times the inch values, so the unstandardised principal direction lies at $21.4°$, close to $\arctan(1/2.54) = 21.5°$. That is the line $x_2 = x_1/2.54$, which is a statement about units, not about the data. After standardising, the principal direction is at exactly $45°$.

58:57 To choose k, look at the eigenvalues. Since the eigenvalue λ_j is the variance along u_j, the fraction

$$\frac{\sum_{j=1}^{k}\lambda_j}{\sum_{j=1}^{n}\lambda_j}$$

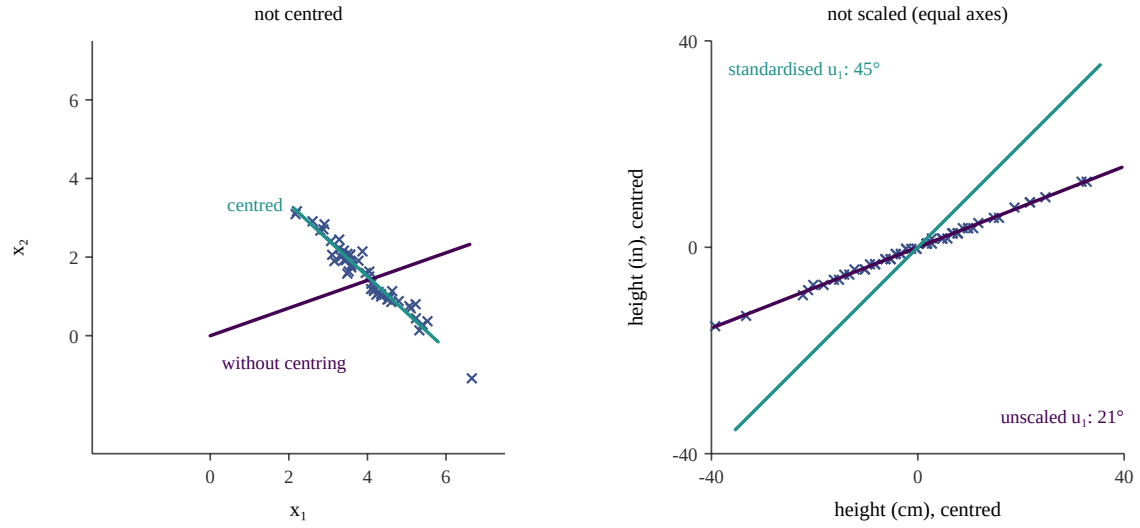


Figure 15.2.: What PCA returns without the pre-processing. Left: a cloud away from the origin; the top eigenvector of the uncentred second-moment matrix points at the cloud (purple), while PCA on centred data follows the cloud's own axis (teal). Right: heights in centimetres against inches, centred but not scaled, on equal axes. The unscaled first direction (purple) follows the centimetre axis at 21°, while after standardising it is 45° (teal).

is the share of the total variance kept by the first k directions. Practitioners report choosing k to "retain 90%" (or 95, 98, or 99%) of the variance. Ng adds that compressing very high-dimensional data tenfold while keeping 95% of its variance is common. The same quantity measures reconstruction quality. The mean squared error of the decompression in Equation 15.2 equals the sum of the discarded eigenvalues, $\sum_{j>k} \lambda_j$, so "variance retained" is one minus the relative reconstruction error. The generator asserts this equality for every k from 0 to 10 on a ten-dimensional data set.

15.5. When to use PCA, and when not to

31:54 The use Ng endorses most is **visualisation**: projecting data from n dimensions down to 2 or 3 so that it can be plotted. His example is from Krishna Shenoy's neuroscience lab at Stanford. Monkeys were trained to tap a square on a touch screen when it appeared, in exchange for juice, while about 50 electrodes recorded activity in their brains. A 50-dimensional time series cannot be looked at directly. A former CS229 student in the lab projected it with PCA and could watch the brain's state move through two or three dimensions: waiting, preparing once the target appeared, holding, and then moving off at the go signal. 36:15 The second endorsed use is **compression for efficiency**. High-dimensional data usually lie close to a much smaller subspace. Ng calls it "just a fact of life". Feeding a learning algorithm a 1,000-dimensional $y^{(i)}$ in place of a 10,000-dimensional $x^{(i)}$ saves memory, bandwidth and compute.

38:23 He calls two other uses questionable. The first is **reducing overfitting** by shrinking the number of features before, say, logistic regression. In his experience this works about half the time. The half that works gets published and the half that fails does not, so the literature makes it look more reliable than it is. He uses regularisation for this instead. 40:16 The second is measuring **similarity in a reduced space**. In early face recognition, two 100×100 images

were compared by Euclidean distance after PCA rather than in raw pixel space, in the hope that the projection would remove noise. Sometimes it helps a little, but not reliably. 42:11 This was the era of eigenfaces. Researchers ran PCA on face images and plotted each eigenvector as a picture. Then they read meaning into individual eigenvectors: one is the lighting direction, another is a beard. Ng's judgement is that such readings were "at least as much a product of the researcher's imagination" as of the data. The reason is that individual eigenvectors are unstable, which the next section quantifies.

44:35 His rule of thumb follows: *before putting PCA into a system, try the system on the original* $x^{(i)}$. Class project proposals often include a PCA step by default. Unless you need the speed or the memory, the step adds a component to debug without a reason. Asked whether there is a minimum dimension below which PCA is pointless, he says no. Even going from two dimensions to one is justified if halving memory really matters.

15.6. Why individual eigenvectors should not be interpreted

54:49 A student asks why k-means centroids are treated as interpretable and principal components are not. Ng's answer is that individual eigenvectors are numerically unstable, while the *subspace* spanned by the first k of them is stable. His picture is three-dimensional data lying close to the plane of the whiteboard. u_1 and u_2 are two directions in that plane, and u_3 points out of the board. A small change to the data can spin u_1 and u_2 around inside the plane, while the plane itself stays put. He says that changing one training example "in the fifth decimal place" can change the eigenvectors significantly.

That claim is true only under a condition the lecture leaves implicit. How far an eigenvector moves depends on the size of the perturbation *relative to the gap* between its eigenvalue and the next one. The generator tests it with data whose top two eigenvalues differ by an exact relative gap. The data are a regular twelve-sided polygon in a plane, stretched slightly along one axis, plus two points off the plane. Changing one coordinate of one point by 10^{-5} moves u_1 by 0.0009° when the gap is 10^{-1}, 0.08° when it is 10^{-3}, 9.6° when it is 10^{-5}, and 59° when it is 10^{-7}. The plane spanned by u_1, u_2 moves by less than 10^{-6} degrees in every case. So a fifth-decimal change matters only when the eigenvalues agree to about five decimals.

The more realistic version is the one Ng gives in answer to a follow-up question 57:56: a lab mate collects a slightly different data set. Then sampling noise, not rounding, is the perturbation. Figure 15.3 draws pairs of 200-point samples from the same three-dimensional distribution and compares their principal directions. When $\lambda_2/\lambda_1 = 0.5$, the two samples' u_1 differ by a median of 5°. At 0.9 they differ by 36°, and at 0.98 and above they are essentially unrelated, around 45–50°. Throughout, the two samples' top-two subspaces differ by under 1°. High-dimensional data typically have long runs of similar eigenvalues, so this is the usual case. The subspace can be reported; the individual axes are an arbitrary choice of basis within it.

Two more questions close the PCA half. 56:21 Is there a probabilistic version? Yes: probabilistic PCA is factor analysis with the noise covariance constrained to $\sigma^2 I$, the spherical restriction of the last lecture, in place of a general diagonal Ψ. 56:33 Can PCA overfit? Ng's answer is that it can when k is close to m, for instance a 1,000-dimensional subspace estimated from 1,000 examples. Then the subspace fits the sample, not the distribution.

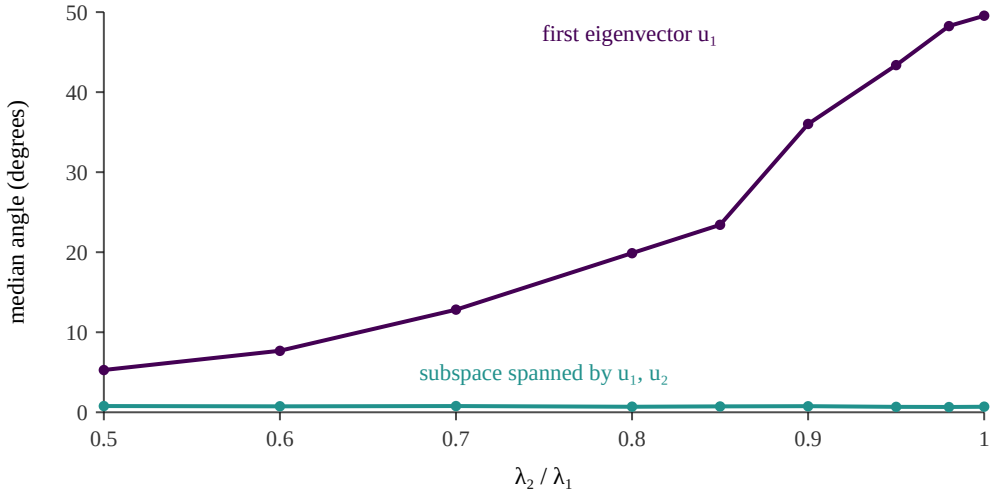


Figure 15.3.: Median angle, over 200 pairs of samples of 200 points from the same three-dimensional distribution, between the two samples' first eigenvectors (purple) and between their two-dimensional principal subspaces (teal), as the second eigenvalue approaches the first. The third eigenvalue is 0.01 throughout.

15.7. A map of the unsupervised algorithms

49:23 A student asks when to use PCA and when to use factor analysis. Ng answers by sorting the four unsupervised algorithms of the course along two axes: whether they model a density $p(x)$, and whether they assume the data lie near a subspace or in clusters ("clumps").

Table 15.1.: The four unsupervised algorithms of lectures 13–15, after Ng's board.

	Models $p(x)$	No probabilistic model
Data near a subspace	factor analysis	PCA
Data in clusters	mixture of Gaussians	k-means

The left column is where EM lives. EM applies whenever there is a model $p(x, z)$ with z unobserved, whether z is discrete (the mixture) or Gaussian (factor analysis). The density models are what you want for anomaly detection. The right column is for compression and for human understanding: PCA to see high-dimensional data, k-means to split customers into market segments or genes into groups. Ng adds that the mixture of Gaussians often works well on data that are not really clustered, so the table is a guide, not a rule.

15.8. Independent components analysis: the cocktail party

1:02:36 PCA looks for directions of high variance. ICA looks for directions that are *statistically independent*. The motivating problem is the cocktail party. Several people talk at once, and several microphones placed around the room each record a different mixture of their voices. Ng plays a demonstration with two microphones. Each one picks up a voice counting to ten in English mixed with another voice in Spanish. ICA, given only the two recordings and told nothing about voices, returns one track per speaker. A second example mixes the counting voice

with music from a radio. Because this one was recorded in a real room, Ng notes, the separation is not perfect, but most of the music is gone.

1:07:32 The model is linear. Let $s^{(i)} \in \mathbb{R}^n$ hold the n sources at time i, where $s_j^{(i)}$ is the air-pressure deviation produced by speaker j. Microphone j records

$$x_j^{(i)} = \sum_{k=1}^{n} A_{jk}\, s_k^{(i)}, \qquad \text{that is,} \qquad x^{(i)} = As^{(i)}, \tag{15.3}$$

with an unknown, square *mixing matrix* A. Here $x^{(i)}$ is the recording at time i, and the recordings must be time-aligned across microphones. The goal is the *unmixing matrix* $W = A^{-1}$, so that $s^{(i)} = Wx^{(i)}$. Writing $w_j^\top$ for the rows of W, source j is $s_j^{(i)} = w_j^\top x^{(i)}$. Asked whether there must be as many microphones as speakers, Ng says yes for now. The general case is deferred.

Two caveats are worth stating. First, Equation 15.3 is an idealisation. In a real room each voice reaches each microphone after a delay and with echoes, so a microphone records a sum of filtered copies of each source, not a single weighted sum. This is part of why the real-room demonstration separates imperfectly. Second, Ng remarks that "sound can only add", and that some versions of ICA put sign constraints on A and W for that reason. That needs restating. The sources are pressure *deviations*, which are positive and negative. Sound waves superpose with signs, and cancellation is how noise-cancelling headphones work. What is plausibly non-negative is the *gain* from a speaker to a microphone, so a sign constraint, where one is used, belongs on the entries of A, not on the signals.

1:14:12 Why should recovery be possible at all? Ng's picture (Figure 15.4) uses two sources that each emit an independent value uniform on $[-1, 1]$ at every instant, so the pairs (s_1, s_2) fill a square. The mixing matrix shears the square into a parallelogram, and that is all the microphones see. The edges of the parallelogram reveal the directions of the original axes, and an unmixing matrix that turns the parallelogram back into an axis-aligned square recovers the sources.

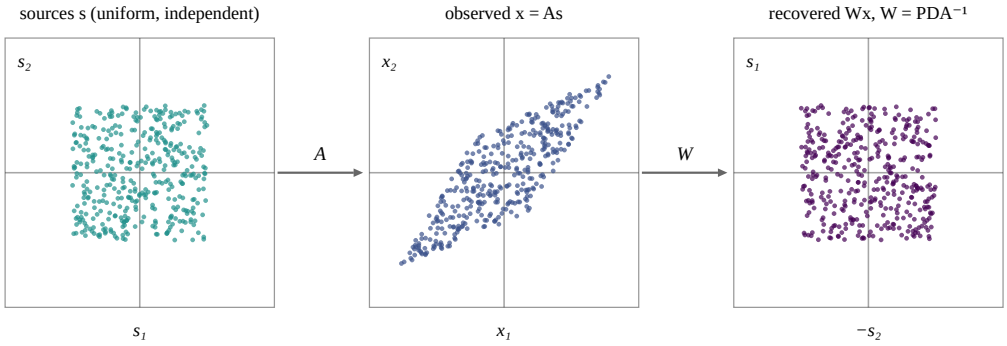


Figure 15.4.: The ICA problem with two uniform sources. Independent sources fill a square (left). Mixing by A gives the parallelogram the microphones observe (centre). Unmixing by $W = PDA^{-1}$, where P swaps the two rows and D flips one sign, gives a square again (right): the sources are recovered, but as $(-s_2, s_1)$.

1:15:44 Recovery has unavoidable ambiguities. The first is *order*. Nothing in the data says which source is the person and which is the radio, so the sources come back in an arbitrary order. The second is *sign*. A square flipped horizontally or vertically is the same square, so a source can come back negated. For audio that is harmless, because $-s_j$ sounds identical to s_j. The official notes add a third, *scale*. Multiplying a source by α and the corresponding column of A by $1/\alpha$

leaves x unchanged, so the sources are determined only up to scale, and for audio that is only a volume. All three combine into $W = PDA^{-1}$ with P a permutation and D diagonal, as in the right panel of Figure 15.4. The generator checks that every such W, including swaps, sign flips and rescalings by 2 and $-\frac{1}{2}$, returns exactly the sources, permuted and rescaled.

The picture also shows what ICA depends on. A uniform square has corners, which mark the original axes. Lecture 16 shows that Gaussian sources, whose density is round, have no such corners, and that this ambiguity cannot be resolved. It then derives the algorithm that finds W by maximum likelihood.

> **In the official notes**
>
> PCA, with the variance derivation and the eigenvector solution, is cs229-notes10. The ICA model, its permutation, scaling and sign ambiguities, and the reason Gaussian sources cannot be separated are in cs229-notes11, section 1. The density of a linear transformation and the maximum-likelihood algorithm for W are in sections 2 and 3.

Part VI.

Reinforcement Learning

16. Independent Component Analysis and RL

Lecture 16

Lecture video: Lecture 16: Independent Component Analysis and RL · 1:18:10

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

The lecture has two halves that share nothing but a date. The first finishes independent components analysis. Chapter 15 set up the cocktail-party model $x = As$ and showed that the sources can be recovered only up to order, sign and scale. This chapter adds the reason ICA needs non-Gaussian sources, writes down the likelihood of the unmixing matrix W, and derives the stochastic gradient rule that maximises it. The second half opens the last part of the course, reinforcement learning. It introduces the Markov decision process and the 11-state gridworld that lectures 16 and 17 use as their running example, and states the goal: a policy that maximises the expected discounted sum of rewards.

16.1. Gaussian sources cannot be unmixed

3:07 Chapter 15's picture of why ICA works used sources uniform on $[-1, 1]$. Their joint distribution fills a square, the mixing matrix shears it into a parallelogram, and the corners of the parallelogram give away the directions of the original axes. Ng now asks what happens if the sources are Gaussian instead. Two independent standard Gaussians have a joint density with circular contours. A mixing matrix stretches the circle into an ellipse, and an unmixing matrix that maps the ellipse back to a circle is easy to find. But it is not unique, because every rotation of a circle is the same circle. If W turns the data back into independent unit Gaussians, so does RW for any rotation R, and nothing in the data can choose between them.

The official notes make this precise. If $s \sim \mathcal{N}(0, I)$ then $x = As$ is Gaussian with covariance $AA^\top$, and for any orthogonal R the mixing matrix AR gives $x' = ARs$ with covariance $ARR^\top A^\top = AA^\top$. The two models produce the same distribution of recordings. The generator checks the same fact from the other side: with a Gaussian source prior, the log-likelihood of the 1,500-example dataset used below is unchanged, to rounding error, when the true unmixing matrix is rotated by 10°, 30°, 45° or 60°. Under the logistic prior introduced below it drops by 0.030 per example at 45°, so that likelihood does single out the right rotation.

6:44 Ng cites the theorem behind this informally: the Gaussian is the only distribution of independent components that is rotationally symmetric, and so ICA is possible exactly when the data are non-Gaussian. The careful version, due to Comon, is slightly weaker in its demand: the sources are identifiable up to order and scale as long as *at most one* of them is Gaussian. One Gaussian source has no other Gaussian to be confused with, and two or more can be rotated into one another.

16.2. The density of a linear transformation

12:08 To fit W by maximum likelihood we need p_x, the density of the recordings, given p_s, the density of the sources, and the relation $s = Wx$. The tempting answer is $p_x(x) = p_s(Wx)$. Since W is invertible, every x corresponds to exactly one s, so it seems the probability of seeing x is the probability of the matching s. Ng points out that this is correct for probability mass functions and wrong for densities.

14:31 His one-dimensional example shows why. Let s be uniform on $[0, 1]$, so $p_s(s) = 1$ there, and let $x = 2s$, so $A = 2$ and $W = \frac{1}{2}$. Then x is uniform on $[0, 2]$, and a density spread over twice the width must be half as tall: $p_x(x) = \frac{1}{2}$ on $[0, 2]$. The naive formula gives $p_s(x/2) = 1$ on $[0, 2]$, which integrates to 2 (Figure 16.1, left). The missing factor is the determinant:

$$p_x(x) = p_s(Wx)\,|W|, \tag{16.1}$$

where $|W|$ is the absolute value of the determinant of W. In one dimension it is just $|W| = \frac{1}{2}$, and the generator confirms numerically that the corrected density integrates to 1 and the naive one to 2. The intuition carries to n dimensions. A maps the unit cube of source space to a parallelepiped of volume $|A|$, so probability spread over that volume has its density divided by $|A|$, that is, multiplied by $|W| = 1/|A|$.

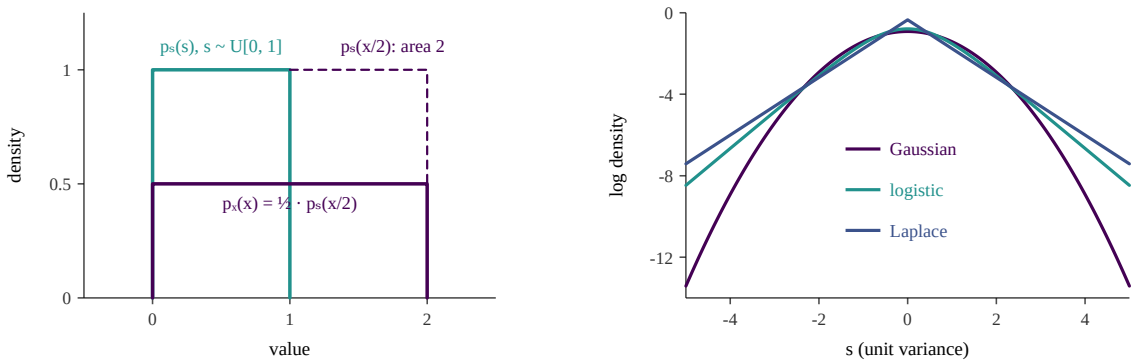


Figure 16.1.: Left: the density of $s \sim U[0, 1]$, the density of $x = 2s$, and the naive $p_s(x/2)$, which has area 2. Right: log densities of the three candidate source priors, all scaled to unit variance. The Gaussian falls off as a parabola, the logistic and Laplace densities linearly.

16.3. A prior for the sources: the sigmoid as a CDF

18:22 The last modelling choice is p_s, the density of a single speaker's signal. It must not be Gaussian. Ng specifies it through its cumulative distribution function $F(s) = P(S \le s)$, which rises smoothly from 0 to 1, and recovers the density as $p_s = F'$. The question then becomes which familiar function rises smoothly from 0 to 1, and the sigmoid is the obvious answer. So the model takes $F = g$, with $g(s) = 1/(1 + e^{-s})$, and

$$p_s(s) = g'(s) = g(s)\,(1 - g(s)).$$

This is the logistic density. Ng's argument for it is its tails. A Gaussian density falls like $e^{-s^2/2}$; the logistic falls like $e^{-|s|}$, so large values are far more likely, and that, he says, describes recorded speech and many natural signals better. The right panel of Figure 16.1 shows the difference on a log scale. At equal variance, the logistic density at four standard deviations is 9.6 times the Gaussian's, and its excess kurtosis is 1.2 where the Gaussian's is 0. Ng adds that a two-sided exponential, the Laplace density, works as well, and he has used it himself. Its tails are heavier still, with excess kurtosis 3. The sigmoid, he recalls, was simply the choice that early ICA researchers needed and tried, in Bell and Sejnowski's information-maximisation algorithm, and it has stayed the default because it works.

28:24 A student asks why the joint density of two logistic sources is not also rotationally symmetric. Ng sketches the contours of a Laplace pair as diamonds and guesses that the logistic looks "a bit like that too", inviting the class to plot it. The plot is Figure 16.2, and the guess is half right. Close to the origin the logistic density is nearly Gaussian, because $\log g'(s) \approx -\log 4 - s^2/4$ for small s, so the inner contours are almost circles: along the diagonal the innermost one reaches 0.98 of its distance along an axis. Farther out the linear tails take over and the contours become diamonds, with that ratio falling to 0.86 at the outermost level drawn. A Laplace pair has diamonds at every level, with ratio $1/\sqrt{2} \approx 0.707$, and a Gaussian pair has circles. What matters for ICA is only that the logistic contours are not circles. Any departure from rotational symmetry gives the likelihood a preferred orientation.

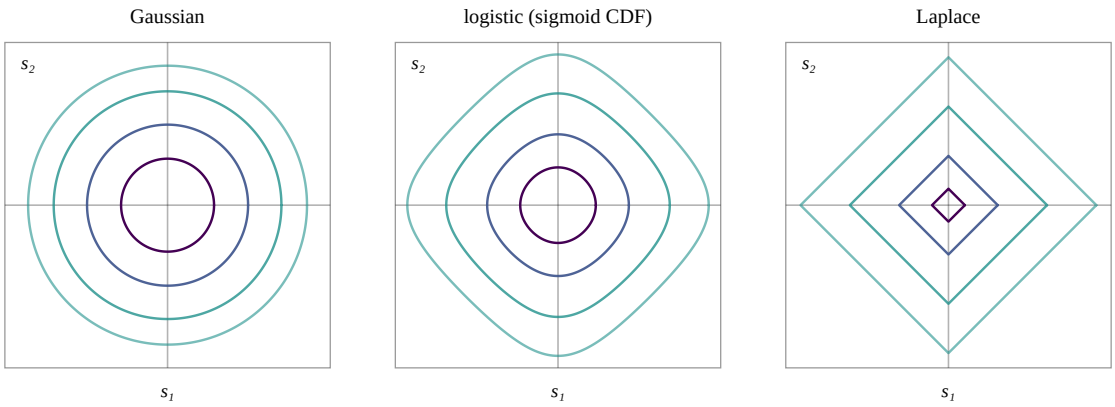


Figure 16.2.: Contours of the joint density of two independent unit-variance sources under three priors, at the same four drops in log density from the peak. Gaussian contours are circles. Logistic contours are nearly round near the centre and become diamonds outward. Laplace contours are diamonds at every level.

16.4. Maximum likelihood for the unmixing matrix

21:39 With these pieces the likelihood is short. ICA's central assumption is that the speakers are independent, so the joint source density is a product, $p(s) = \prod_{j=1}^{n} p_s(s_j)$. Combining that with Equation 16.1 and $s_j = w_j^\top x$, where $w_j^\top$ is the j-th row of W, gives

$$p(x) = \left(\prod_{j=1}^{n} p_s\left(w_j^\top x\right) \right) |W|.$$

For a training set $x^{(1)}, \ldots, x^{(m)}$, one microphone reading per instant, the log-likelihood of W is

$$\ell(W) = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \log g'\left(w_j^\top x^{(i)}\right) + \log|W| \right). \tag{16.2}$$

26:02 Ng writes the derivative without deriving it. The official notes do the algebra, which needs the identity $\nabla_W |W| = |W|(W^{-1})^\top$. For one example it gives the stochastic gradient ascent rule

$$W := W + \alpha \left(\begin{bmatrix} 1 - 2g(w_1^\top x^{(i)}) \\ \vdots \\ 1 - 2g(w_n^\top x^{(i)}) \end{bmatrix} x^{(i)\top} + (W^\top)^{-1} \right). \tag{16.3}$$

The first term comes from the prior: $\frac{d}{ds} \log g'(s) = 1 - 2g(s)$. The second comes from the determinant and keeps W from collapsing to zero, which would otherwise make every $w_j^\top x$ zero and every source density maximal. A student catches a slip on the board here: the bracket should hold $1 - 2g$, with g the sigmoid. The generator compares the analytic gradient with central finite differences of Equation 16.2, and they agree to 2×10^{-10}.

27:12 The whole algorithm, then, is to initialise W, run gradient ascent on Equation 16.2, and read each source off as $s_j^{(i)} = w_j^\top x^{(i)}$. Playing $s_1^{(1)}, \dots, s_1^{(m)}$ through a speaker plays back speaker 1. Figure 16.3 runs it on 1,500 readings of two independent Laplace sources mixed by a fixed A. Batch gradient ascent from $W = I$ raises the average log-likelihood from -3.736 to -3.109 and levels off. The product WA it ends with is $\left[\begin{smallmatrix} 1.310 & 0.051 \\ -0.035 & 1.411 \end{smallmatrix}\right]$, a scaled identity up to cross-talk of under 4%, and each recovered signal correlates with its source at 0.999 or above. The scale is not 1, because maximum likelihood rescales each source to suit the logistic prior. That is the scale ambiguity from chapter 15, resolved arbitrarily. The data here are generated exactly by the ICA model. Ng's real-room recording separated less cleanly, for the reasons Section 15.8 gives.

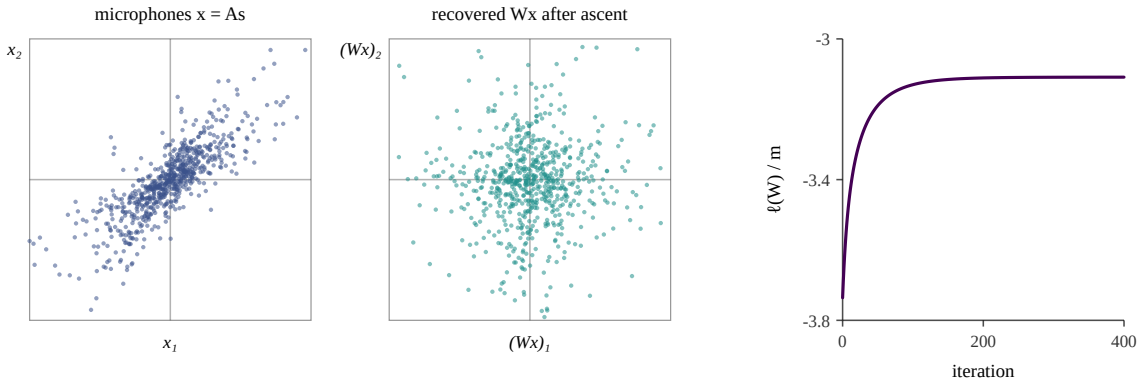


Figure 16.3.: ICA on two mixed Laplace sources. Left: the microphone readings, whose arms run along the columns of A. Centre: the same points multiplied by the W that gradient ascent finds; the arms now lie on the axes, so each coordinate is one source. Right: the average log-likelihood over 400 iterations of batch gradient ascent from $W = I$.

> **In the official notes**
>
> The density of a linear transformation and the full derivation of Equation 16.3 are in cs229-notes11, sections 2 and 3. The notes also remark that the data should have zero mean, which the sigmoid prior assumes, and that the training examples, being successive

moments of audio, are not independent. The notes say this does not hurt the result, and that visiting the examples in a random order can speed up stochastic gradient ascent.

16.5. Where ICA is used, and where it breaks

30:03 Asked for the closest nonlinear extension of ICA, Ng says there is no good answer yet and points to his own group's work on stacking it. The large unsupervised network that learned to detect faces and cats in YouTube frames (Le et al., 2012) was trained layer by layer with an objective the paper calls reconstruction topographic ICA, a close relative of sparse coding. Ng's view is that hierarchical versions of ICA and sparse coding lost attention when supervised deep learning took off, and deserve more of it than they get.

33:46 The standard application is cleaning electroencephalography (EEG) data. Electrodes on the scalp each record a weighted sum of the electrical activity of many brain processes at once: the heartbeat, eye blinks, everything else. That is a cocktail party in the head, and ICA unmixes it the same way. An analyst identifies the components that are heartbeat or blink artefacts, removes them, and remixes the rest (Jung et al., 2000). The clean signal is then the input to whatever comes next, such as a classifier that tries to tell from EEG whether a person is thinking of a person or an object. Run on small patches of natural images, ICA finds components that are localised, oriented edges (Bell and Sejnowski, 1997). The same result from sparse coding (Olshausen and Field, 1996) is one of the reasons neuroscientists ask whether the visual cortex learns its edge detectors this way.

40:41 A student had asked whether the number of microphones must equal the number of speakers. Ng's answer is that more microphones than speakers is fine: run ICA, perhaps slightly modified, and some of the recovered sources come out silent. Fewer microphones than speakers is a research problem. With one microphone and two speakers, A is 1×2 and cannot be inverted, so separation needs knowledge beyond independence. A low voice and a high voice can sometimes be pulled apart, two similar voices hardly at all.

The "slightly modified" matters more than it sounds. Run Equation 16.2 unchanged with ten microphones and five speakers, and W is 10×10 while the recordings fill only a five-dimensional subspace. Any row w_j orthogonal to that subspace outputs exactly zero for every example, so it is one of the silent sources. Scaling that row by c leaves its $\log g'(0)$ terms unchanged and multiplies $|W|$ by c. The log-likelihood therefore grows by $m \log c$ without bound. The generator confirms the exact $m \log c$ gain for three microphones and two sources. So the silent sources are where the plain algorithm diverges, not a harmless by-product. The usual modification is to reduce the recordings to as many dimensions as there are sources, with PCA, before running ICA.

Ng's other suggestion, from a student's follow-up, is to parameterise the model by the rectangular mixing matrix A, 10×5 here, instead of W. That avoids the problem but loses the convenient formula Equation 16.1, which needs a square W. How many sources to assume, he says, is like choosing k in k-means: try several and look for components that come out near-silent.

16.6. Why reinforcement learning

49:52 The fourth and last part of the course is reinforcement learning. Ng motivates it with the autonomous helicopter from lecture 1. At every moment the helicopter is in some *state*, its position, orientation and velocity, and must choose how to move the control sticks. Supervised

learning would need, for each state, the one correct stick position, and nobody can write that down. What a designer *can* write is a *reward function*: high when the helicopter flies the intended trajectory, very negative when it crashes. Ng compares it to training a dog. You cannot tell the dog what to do, but you can say "good dog" and "bad dog", and it learns to do more of what earns the first. Games fit the same mould. Nobody knows the one best chess move in a given position, but a program can play and be rewarded with $+1$ for a win, -1 for a loss and 0 for a draw.

53:41 Such sparse feedback creates what Ng calls the *credit assignment problem*. A chess program checkmated on move 50 receives -1, but the losing blunder may have been on move 20, with the remaining 30 moves only playing out a lost position. A self-driving car that crashes usually made the decisive mistake seconds earlier, not at the instant of impact. Given only the final reward, the algorithm has to work out which of its earlier actions to do more of and which to do less of. Every algorithm in the rest of the course solves this problem, at least indirectly, and the value functions of lecture 17 are the main way they do it.

16.7. Markov decision processes

56:29 Reinforcement learning problems are modelled as Markov decision processes. An MDP is a five-tuple $(S, A, \{P_{sa}\}, \gamma, R)$:

- S, the set of states: chess positions, or the helicopter's possible positions, orientations and velocities;
- A, the set of actions: legal moves, or stick positions;
- P_{sa}, the state transition probabilities: for each state s and action a, a distribution over the next state;
- $\gamma \in [0, 1)$, the discount factor, explained in Section 16.8;
- $R : S \to \mathbb{R}$, the reward function.

58:29 The running example is small enough to draw. Ng takes it from Russell and Norvig's textbook (Figure 16.4). A robot moves on a 4×3 grid with one cell blocked by a pillar, so there are 11 states, written (x, y) from the bottom left. The actions are the four compass directions. Real robots' wheels slip, so the model makes moves noisy. A command goes in the intended direction with probability 0.8 and veers to each side with probability 0.1. From $(3, 1)$, a command to go North reaches $(3, 2)$ with probability 0.8, $(2, 1)$ with 0.1 and $(4, 1)$ with 0.1. A move into a wall or the pillar leaves the robot where it is. Ng stresses that this is a caricature of real robot dynamics, chosen so that the whole MDP fits on the board.

1:02:41 The task is specified entirely through the reward. Ng puts $R = +1$ at $(4, 3)$, the goal, and $R = -1$ at $(4, 2)$, a cell to avoid. To make the robot hurry, every other state carries a small negative reward, $R(s) = -0.02$, which Ng describes as charging the robot for the electricity it spends loitering. Lecture 17 adds that the ± 1 cells are *absorbing*: once the robot reaches one, it collects that reward and the world ends.

16.8. Payoff, discounting and policies

1:05:15 The robot wakes in some state s_0 and chooses an action a_0. The world moves it to $s_1 \sim P_{s_0 a_0}$, it chooses a_1, moves to $s_2 \sim P_{s_1 a_1}$, and so on. Its *total payoff* is the discounted sum of the rewards along the way:

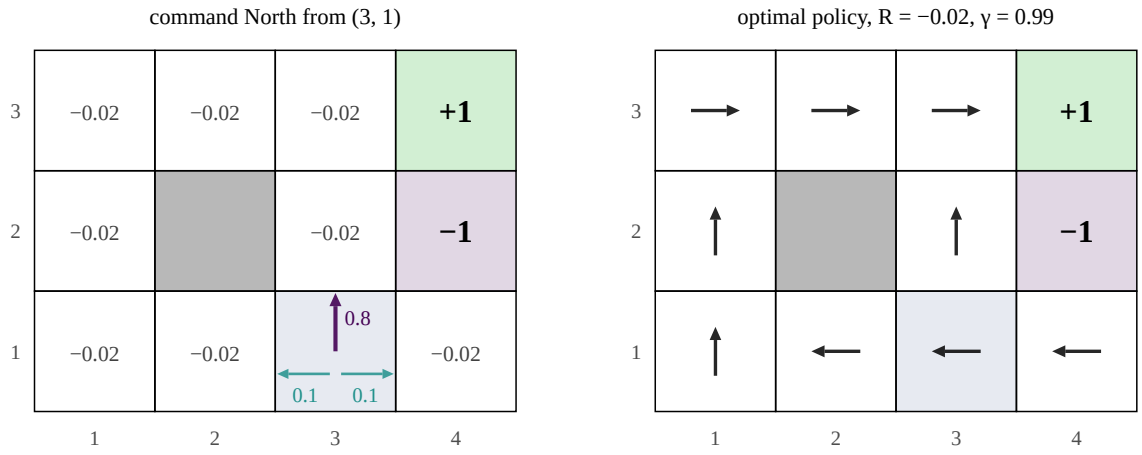


Figure 16.4.: Left: the gridworld's rewards and its transition model. The pillar is grey, the absorbing ± 1 cells are shaded, and every other cell costs 0.02 per visit. From $(3,1)$ a North command succeeds with probability 0.8 and slips sideways with probability 0.1 each way. Right: the optimal policy for $R = -0.02$ and $\gamma = 0.99$. From $(3,1)$ it goes West, the long way round.

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots . \tag{16.4}$$

The goal is to choose actions that maximise the *expected* total payoff, since the transitions are random. With γ around 0.99, a reward one step from now counts 0.99 as much as one now, and one a hundred steps away about 0.37 as much.

1:08:00 Ng gives two reasons for γ, and says plainly that the second is the real one. The first is the time value of money. A dollar today is worth more than a dollar next year, because it can earn interest, and by the same logic a payment is better postponed than made now. So γ makes the robot hurry towards positive rewards and put off negative ones. The second is that with $\gamma < 1$ the algorithms provably converge, and with $\gamma = 1$ many of them do not, or are much harder to analyse. A student supplies the mechanism. Without discounting, the sum Equation 16.4 can grow or shrink without bound. With it, $|\text{payoff}| \le R_{\max}/(1-\gamma)$. For a robot that loiters for ever at -0.02 per step with $\gamma = 0.99$, that is -2, a finite number an algorithm can reason about.

1:11:28 What a reinforcement learning algorithm produces is a *policy*, $\pi : S \to A$, a rule for which action to take in each state. Executing π means taking $a = \pi(s)$ in whatever state the robot is in. Ng shows the optimal policy for the gridworld (Figure 16.4, right), computed offline. Lecture 17 derives value iteration, the algorithm that computes it. The generator runs it and reproduces, to two decimals, the values Ng later reads off the board.

1:14:44 The interesting cell is $(3,1)$. North is the direct route: two steps to $(3,3)$, then one East to the goal. But at $(3,2)$ a 0.1 chance of slipping East lands the robot on -1. West goes all the way round by $(1,1)$ and the top row, which costs more steps and more discounting, but never passes the -1. Ng's point is that the answer is not obvious by eye, and the optimal policy takes the long, safe route. The generator measures how fine the distinction is. At $(3,1)$, the expected value of the next state is 0.736 for West against 0.674 for North, and the runner-up is not North but South, at 0.691: bumping into the wall and staying put is better than going North. The choice holds only while the step reward lies between 0 and -0.0368. Below that, loitering costs enough that the risky short route wins. Elsewhere the margins are smaller still. At $(3,2)$, North beats West by 0.0005, and only because the step reward is below -0.0196; at $(4,1)$, West beats

South by 0.003. An MDP solver makes these distinctions exactly. A person eyeballing the grid would not.

1:16:13 Two last questions point ahead. Chess fits the framework if a "state" is the board when it is your turn: you move, the opponent moves, and the opponent's unknown reply becomes the randomness in P_{sa}. And the probabilities 0.8, 0.1, 0.1 need not be given. For a real robot they are usually estimated from data, which is how lecture 17 ends. The number of policies is finite but large. Lecture 17 counts it as 4^{11}, though only the nine non-absorbing states need an action, so $4^9 = 262{,}144$. Enumerating them is hopeless for any realistic MDP, and the next lecture's value and policy iteration avoid enumeration altogether.

> **In the official notes**
>
> Markov decision processes, the five-tuple, the total payoff and policies are in cs229-notes12, section 1. The notes allow rewards that depend on the action, $R(s, a)$, and mention finite-horizon MDPs. The lecture keeps to $R(s)$ and infinite horizons.

17. MDPs and Value/Policy Iteration

Lecture 17

Lecture video: Lecture 17: MDPs and Value/Policy Iteration · 1:19:14

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Lecture 16 defined a Markov decision process and claimed an optimal policy for the 11-state gridworld without saying where it came from. This lecture supplies the method. It defines three objects: V^{π}, the value of following a fixed policy; V^{*}, the best value any policy can achieve; and π^{*}, the policy that achieves it. Each satisfies a version of Bellman's equation, and the two standard algorithms, value iteration and policy iteration, are two ways of solving those equations. The lecture then drops the assumption that the transition probabilities are known. It estimates them from the robot's own experience, and this raises the question of how much the robot should explore.

17.1. The value of a policy

6:46 For a policy π, the *value function* $V^{\pi} : S \to \mathbb{R}$ gives the expected total payoff of starting in s and following π for ever:

$$V^{\pi}(s) = \mathbb{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \,\middle|\, s_0 = s,\ \pi\right].$$

8:40 Ng's example is deliberately poor. From some states it drives straight for the -1. The left half of Figure 17.1 uses a concrete policy of that kind, East from every state. On the top row it reaches the $+1$ and is worth about 0.6 to 0.76. Along the middle and bottom rows it runs into the -1, or stalls against the east wall paying -0.02 a step, and is worth about -0.3 to -1.1. The ± 1 cells are *absorbing*: once the robot reaches one, it collects that reward and the world ends.

10:57 The equation that governs V^{π} comes from splitting the payoff after its first term. Factor one γ out of everything after $R(s_0)$:

$$R(s_0) + \gamma\big(R(s_1) + \gamma R(s_2) + \cdots\big).$$

The bracket is the payoff of a robot that starts in s_1 and follows π. Its expectation is $V^{\pi}(s_1)$, and s_1 is drawn from $P_{s\pi(s)}$, because in state s the policy takes action $\pi(s)$. Writing s' for the next state gives *Bellman's equation* for V^{π}:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s')\, V^{\pi}(s'). \tag{17.1}$$

The first term is the *immediate reward*, which the robot gets just for waking up in s. The second is the discounted expected future reward.

19:42 For a fixed π, Equation 17.1 is *linear* in the unknowns $V^\pi(s)$. For $s = (3,1)$, with π going North, it reads

$$V^\pi(3,1) = R(3,1) + \gamma\big[0.8\,V^\pi(3,2) + 0.1\,V^\pi(2,1) + 0.1\,V^\pi(4,1)\big],$$

and one such equation for each of the 11 states gives 11 linear equations in 11 unknowns. A linear solver handles that instantly. The generator solves the system for the East policy and checks the result against 5,000 sweeps of repeated substitution into Equation 17.1. The two agree to 10^{-9}.

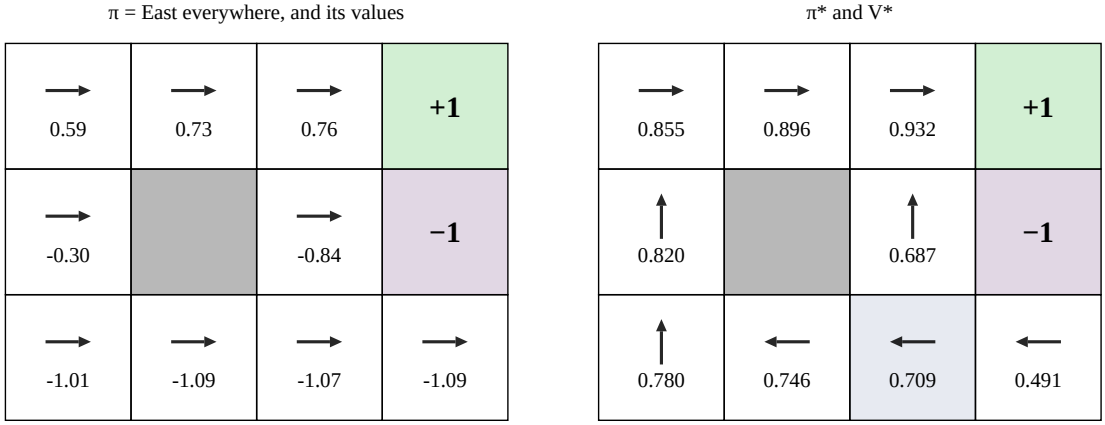


Figure 17.1.: Left: the policy "always go East" and its value V^π, solved from the 11 Bellman equations. Right: the optimal policy and the optimal value function V^* for $R = -0.02$, $\gamma = 0.99$. Cell $(3,1)$, whose action the lecture works out by hand, is highlighted.

17.2. The optimal value function and the optimal policy

24:47 The *optimal value function* is the best value any policy achieves from each state:

$$V^*(s) = \max_\pi V^\pi(s).$$

Ng flags a terminology trap. Both V^π and V^* are called "value functions", but one belongs to a particular policy, good or bad, and the other to the best of them. The maximum is over a combinatorially large set, 4^9 policies even for this small grid, so the definition does not suggest how to compute V^*. Its Bellman equation does.

27:12 Run the same argument as for Equation 17.1, but let the robot choose its first action well. It collects $R(s)$ for waking up in s. Whatever action a it takes, it then lands in $s' \sim P_{sa}$ and can collect at most $V^*(s')$ from there. The best first action is the one with the largest expected continuation, so

$$V^*(s) = R(s) + \max_{a \in A}\ \gamma \sum_{s' \in S} P_{sa}(s')\,V^*(s'). \tag{17.2}$$

29:44 If V^* were known, the optimal action would be the one that attains that maximum. Ng asks the room what to do in state s, and the answer is already in Equation 17.2:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s')\, V^*(s'). \tag{17.3}$$

γ is dropped because a positive constant does not move an argmax. 32:50 Ng states, without proof, that this policy is optimal from every state at once: $V^*(s) = V^{\pi^*}(s) \geq V^{\pi}(s)$ for every π and every s. One policy is best everywhere, so no state has to trade off against another. The generator checks both halves. The value of the greedy policy, solved from its own linear system, equals V^* to 10^{-9}, and V^* is at least V^{π}, state by state, for 3,000 random policies. The plan is therefore to find V^* and then read off π^* with Equation 17.3.

17.3. Value iteration

35:22 Value iteration treats Equation 17.2 as an update rule:

1. Initialise $V(s) := 0$ for every s.
2. Repeat until convergence: for every s, $V(s) := R(s) + \max_a \gamma \sum_{s'} P_{sa}(s')\, V(s')$.

38:07 There are two ways to carry out step 2, and Ng compares them to gradient descent. A *synchronous* update computes the right-hand side for all 11 states from the old values, then overwrites all 11 at once. In vector form this is $V := B(V)$, where B is called the *Bellman backup operator*. An *asynchronous* update sweeps through the states one at a time and overwrites each value as soon as it is computed, so later states in the sweep already use the new values. Both converge to V^*.

Ng's default is the synchronous form, because it is a matrix operation and vectorises. He says "asynchronous" at this point in the recording, but the reason he gives, vectorisation, and the procedure he describes, compute all eleven and then update, are the synchronous one. The asynchronous form has the opposite advantage: it needs fewer sweeps. On the gridworld with $\gamma = 0.99$, the max-norm error falls below 10^{-6} after 21 asynchronous sweeps against 29 synchronous ones (Figure 17.2).

40:32 A student asks how to code the absorbing states. One way is to give them transition probabilities of zero to every state, so that their value is just R. Another is to add a twelfth, zero-reward "sink" state that they lead to and that loops on itself. Ng calls the second cleaner and the first more common. Both give the same values.

43:07 Why does it converge? The problem set shows that B is a *contraction* in the max norm: $\|B(V) - B(V')\|_\infty \leq \gamma \|V - V'\|_\infty$. Since $V^* = B(V^*)$, each synchronous sweep shrinks the error by at least a factor γ, so $\|V_k - V^*\|_\infty \leq \gamma^k \|V_0 - V^*\|_\infty$. Ng turns this into a rule of thumb: at $\gamma = 0.99$, value iteration needs a few hundred iterations, and at $\gamma = 0.9$ a few dozen.

The bound is right but, for this MDP, very loose. Taken literally, it needs 1,375 sweeps per unit of initial error to reach 10^{-6} at $\gamma = 0.99$, and 132 at $\gamma = 0.9$. The gridworld gets there in 29 sweeps at $\gamma = 0.99$ and 21 at $\gamma = 0.9$. The generator asserts that every synchronous sweep is within the bound, so nothing here contradicts the theorem. The reason is the absorbing states. After k sweeps the remaining error is roughly the discounted value of what happens beyond step k. Under π^*, starting from $(1, 1)$, the robot is still wandering after 10 steps with probability 0.04, and after 20 steps with probability essentially zero, so there is little left to discount. The factor γ only governs the rate when the process can run on for a long time without ending. The

more useful fact for practice is that the *policy* converges before the values. With $\gamma = 0.99$, the greedy policy Equation 17.3 of the synchronous estimate is already exactly π^* after 11 sweeps, while the values are still off by as much as 0.08.

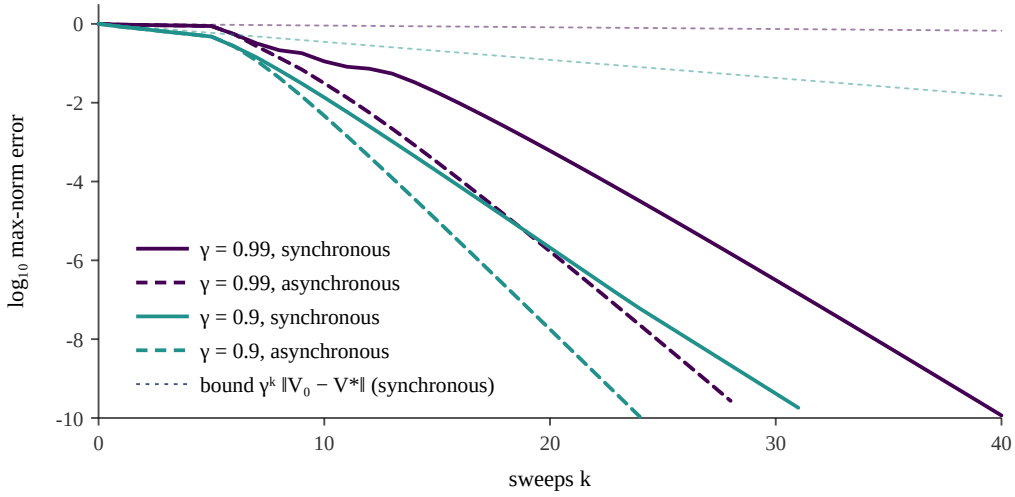


Figure 17.2.: Value iteration on the gridworld: base-10 logarithm of the max-norm error against the number of sweeps, synchronous and asynchronous, at $\gamma = 0.99$ and $\gamma = 0.9$. Dotted lines are the guarantee $\gamma^k \|V_0 - V^*\|_\infty$, which every synchronous curve stays below, by a wide margin here because the robot is absorbed within a few steps.

17.4. Reading the policy off V^*

45:15 Ng writes the converged V^* on the board and uses it to justify the West arrow at $(3,1)$ that lecture 16 asserted. Going West reaches $(2,1)$ with probability 0.8, where $V^* \approx 0.75$. It slips North to $(3,2)$, worth 0.69, with probability 0.1, and South into the wall, staying at $(3,1)$, worth 0.71, with probability 0.1. The expected value is 0.740. Going North gives $0.8 \times 0.69 + 0.1 \times 0.75 + 0.1 \times 0.49 = 0.676$, so West wins.

The generator's V^* (Figure 17.1, right) matches the board values to two decimals: 0.746, 0.687, 0.709 and 0.491. Computed from the unrounded values, the two expectations are 0.736 and 0.674. Ng adds that a full check compares all four actions, and the full check is instructive. South, at 0.691, beats North: bumping into the wall and staying put is the second-best action, because North risks the slip into -1. East is last at 0.532.

17.5. Policy iteration

50:18 Policy iteration keeps a policy, not a value function, at the centre:

1. Initialise π randomly.
2. Repeat until π stops changing:
   a. $V := V^\pi$, by solving the linear system Equation 17.1;
   b. for every s, $\pi(s) := \arg\max_a \sum_{s'} P_{sa}(s')\, V(s')$.

Step (b) pretends that the current V is optimal and acts greedily on it. Value iteration waits until V has converged to extract a policy. Policy iteration extracts a new one on every round. 54:44 The trade-off, as Ng puts it, is the linear solve in step (a). For 11 states, or a few hundred, it is trivial, and policy iteration is fast. For 10,000 states or a million, solving the system on every round is expensive, and he would use value iteration. He also notes a more academic difference. Value iteration approaches V^* and never exactly reaches it, as gradient descent approaches the minimum of a least-squares cost. Policy iteration stops after a finite number of rounds, at exactly π^* and exactly V^*, the way the normal equations jump straight to the answer. It must stop, because each round improves the policy, by the policy improvement theorem, and there are only finitely many policies to improve through.

The generator checks this on the gridworld. Started from 200 random policies, policy iteration needs between 2 and 8 evaluations, 5 on average, and always ends at the π^* of Figure 17.1. From "always East" it needs 3. The values never decrease from one round to the next, in any state. In practice, Ng says, he sees value iteration used more, because real problems are large. For small ones policy iteration may converge a little faster.

> **In the official notes**
>
> Both algorithms, the synchronous and asynchronous variants, and the statement that policy iteration terminates are in cs229-notes12, section 2. The contraction argument for value iteration is a homework problem in the course.

17.6. Learning the model

58:41 So far P_{sa} has been given. For a real robot, a helicopter or a chess opponent, it usually is not, and has to be estimated from experience. Run some policy for a while, count what happens, and use the maximum-likelihood estimate

$$\hat{P}_{sa}(s') = \frac{\#\{\text{times action } a \text{ was taken in } s \text{ and led to } s'\}}{\#\{\text{times action } a \text{ was taken in } s\}}. \tag{17.4}$$

When the denominator is zero, because a has never been tried in s, the common heuristic is a uniform guess, $\hat{P}_{sa}(s') = 1/|S|$.

1:01:41 Laplace smoothing would also avoid the $0/0$: add 1 to every numerator and $|S|$ to every denominator. Ng says it is allowed but uncommon, because, unlike naive Bayes, MDP solvers are not badly hurt by zero estimates. The generator suggests a stronger statement: on the gridworld, smoothing makes things worse. It runs a uniformly random policy for a given number of moves, builds $\hat{P}$ with and without smoothing, solves for the policy, and scores it by its true loss, the largest $V^*(s) - V^{\hat{\pi}}(s)$ over states, averaged over 40 runs. The plain counts lose 0.078 after 1,000 moves and 0.010 after 10,000. With smoothing the losses are 0.263 and 0.023. The reason is visible in the formula. Smoothing gives every action a small probability of moving the robot to *every* state, including straight to the $+1$ or -1 from across the grid. With 11 states and only a few visits per state-action pair, that invented mass is large, and it is exactly the mass the optimal policy is most sensitive to. Exact recovery of π^* is rare either way, 14 runs in 40 even after 10,000 moves without smoothing, because of the 0.0005 margin at $(3, 2)$ noted in lecture 16. Measuring the loss, not whether the policy is exactly π^*, is the fair test.

1:03:01 Put together, the practical loop is:

1. Take actions according to some policy π to gain experience in the MDP.

2. Update the estimates of P_{sa} from the counts, and of R if it is unknown.
3. Solve Bellman's equations for the estimated MDP by value iteration, warm-started from the previous V if convenient, to get a new V.
4. Update π to be greedy with respect to V, and repeat.

1:06:03 The reward is usually part of the problem specification, but not always. For a trading application the reward is a day's return, and for a car it might be a penalty for every pothole. Both are random functions of the state, so the same loop estimates the expected reward of each state by averaging what it observes.

17.7. Exploration versus exploitation

1:07:33 The loop above is greedy: it always acts on its current best estimate. Ng shows the trouble with a different MDP. A robot starts with a +1 nearby and a +10 farther away. If its first runs happen to reach the +1, the estimated model says that going there works, and the greedy policy keeps going there. Every trip ends at the +1, so the robot's experience stays in that half of the state space, and the +10 is never found. Acting on what is known is *exploitation*. Trying actions that look worse to learn something new is *exploration*. The tension between them is not academic. Ng's example is online advertising. Showing only the ads a user is predicted to click maximises short-term revenue. Showing occasional others is how the platform discovers interests nobody knew the user had.

1:13:28 The standard fix is ε*-greedy* exploration. On each step, with probability $1-\varepsilon$ take the action $\pi(s)$, and with probability ε, for instance 0.1, take a random one. Ng points out that the name is backwards, since ε is the probability of acting *randomly*, not greedily, but it is the name everyone uses. With $\varepsilon > 0$ every action keeps being tried in every state the robot can reach, so the estimated model converges to the true one and the planned policy to π^*, though finding a remote reward by chance can take a long time.

Figure 17.3 makes Ng's example concrete in one dimension, in a toy of our own: an eight-cell corridor with the +10 five steps to the left of the start and the +1 two steps to the right. Moves slip 20% of the time, each step costs 0.02, $\gamma = 0.95$, and the robot learns both P_{sa} and R as above. Going left is worth about 7.65 and going right 0.86, so the right answer is not close. Over 200 independent learners, the fraction whose policy heads for the +10 after 150 episodes is 0.62 for the greedy learner, 0.68 with $\varepsilon = 0.1$ and 0.78 with $\varepsilon = 0.2$. Two details are worth noticing. The greedy learner is not completely stuck: its curve keeps creeping up, presumably because slipping moves sometimes carry it left without being asked. And none of the curves has reached 1, which is Ng's caveat about how long a remote reward can take to find.

1:16:17 Asked whether ε should stay fixed, Ng says there are many heuristics, and a reasonable one is to start large and shrink it. Another is *Boltzmann exploration*. Instead of a uniformly random action with probability ε, choose action a with probability proportional to $\exp(Q(s,a)/\tau)$, where $Q(s,a) = \sum_{s'} P_{sa}(s')V(s')$ is the estimated value of taking a and τ is a temperature. Actions that look nearly as good as the best are tried often, and actions that look terrible almost never. A student asks about rewarding the robot for visiting states it has never seen. That is *intrinsic motivation*, a research line Ng suggests looking up. And asked how often to re-plan, he says as often as the computation allows. It does no harm. With a physical robot or helicopter, where each flight is a session, re-planning between flights is the natural rhythm.

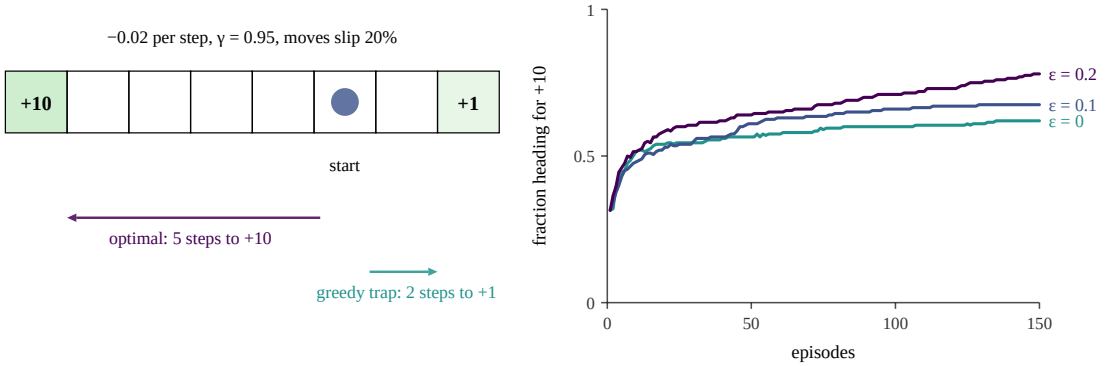


Figure 17.3.: A toy version of the lecture's exploration example, not from the lecture. Left: the corridor, with the start between a distant +10 and a nearby +1. Right: over 200 learners that estimate the model from experience and re-plan after each episode, the fraction whose current policy heads for the +10, for = 0 (greedy), 0.1 and 0.2.

In the official notes

Estimating P_{sa} and R from experience, with the $1/|S|$ rule for untried actions and the warm-started value iteration loop, is in cs229-notes12, section 3. The notes do not treat exploration. Lecture 18 extends the framework to continuous states, covered in section 4 of the same notes.

18. Continuous State MDPs and Model Simulation

Lecture 18

Lecture video: Lecture 18: Continuous State MDP and Model Simulation · 1:20:15

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Value iteration as lecture 17 left it keeps one number per state, so it needs a state space it can list. A car, a helicopter or a pole balanced on a cart has a state made of real numbers. This lecture gives two ways to cope. The first is to chop the state space into cells and pretend it is discrete. The second is to treat the value function as something to *fit*: sample states, compute what value iteration says their values should be, and run linear regression on features of the state to fill in everything between. That second method, *fitted value iteration*, needs a simulator it can query at any state, and Ng spends the middle of the lecture on where such a simulator comes from. His preferred answer is to learn one from a pilot's flight logs.

18.1. States that are vectors

3:43 Ng starts from three robots and asks, for each, what the state has to contain. A car seen from above needs its position (x, y) and its heading θ. That is enough at walking pace, but at any real speed the next position depends on how fast the car is moving, so the state also carries the velocities $\dot{x}, \dot{y}, \dot{\theta}$: six numbers. Whether to add the steering-wheel angle or the wear on the tyres is a modelling choice that depends on what you are building. A helicopter moves in three dimensions, so it has position (x, y, z), an orientation given by roll, pitch and yaw (ϕ, θ, ψ), and the six corresponding velocities: twelve numbers. The third robot is the classic toy of early reinforcement learning, the *inverted pendulum*: a cart on a rail with a pole hinged freely on top, which the controller must keep upright by pushing the cart left and right. Its state is $(x, \theta, \dot{x}, \dot{\theta})$, four numbers.

10:06 In all three, $S = \mathbb{R}^n$ with n equal to 4, 6 or 12. Angles wrap around rather than run along the real line, which a mathematician would object to; for the algorithms in this lecture the distinction does not matter.

18.2. Discretization, and what it costs

10:55 The direct approach lays a grid over the state space, gives every cell a name, and treats the robot as being "in" a cell whenever its true state falls inside it. The continuous problem becomes a discrete MDP that value iteration can solve, and the policy for a continuous state is

the policy of its cell. Ng is clear that this is often fine. For a low-dimensional problem it is the quickest thing to get working.

12:31 It has two weaknesses, and he makes the first one with the example the course opened with. Suppose you predicted house prices by cutting the size axis into five intervals and fitting a constant on each. You would get a staircase, a poor representation of a price that varies smoothly with size. Discretizing the state space does exactly this to the value function: V^* is approximated by a constant on every cell. The left of Figure 18.1 shows the staircase on a one-dimensional MDP of our own. The state s is a point on a line, the actions move it left, right or not at all by 0.25 with Gaussian slip of standard deviation 0.15, the reward is $-s^2$, and $\gamma = 0.9$. With six cells, the discretized value function misses the true V^* by up to 14.8. Doubling the cells halves the error, to 7.0, then 3.4 and 1.8 with 48 cells. The dashed curve is the method the rest of the lecture builds, which fits five numbers and lands within 1.8.

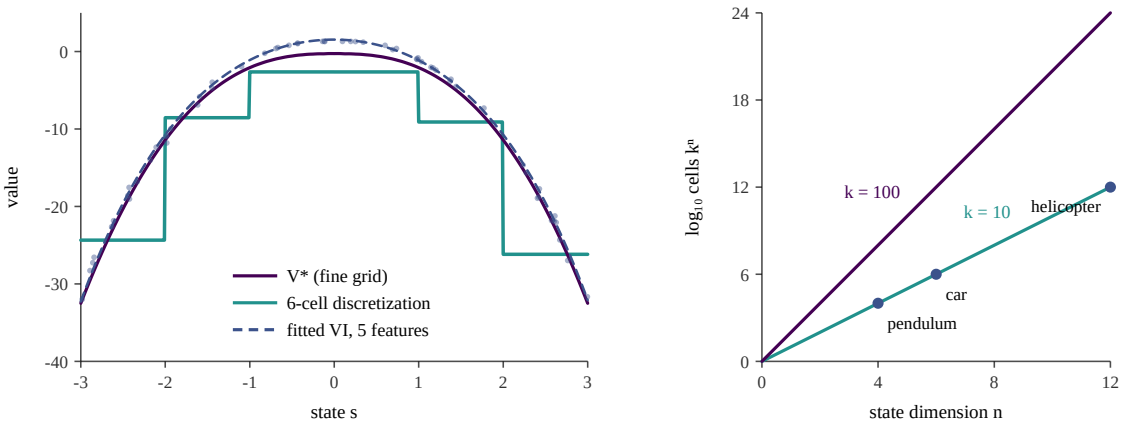


Figure 18.1.: Left: the optimal value function of a one-dimensional MDP (purple), the staircase a six-cell discretization gives (teal), and fitted value iteration with features $1, s, \dots, s^4$ (dashed). Right: the number of cells k^n needed to discretize each of n state variables into k values, on a log scale, with the lecture's three robots marked.

14:35 The second weakness is the one Bellman named the *curse of dimensionality*. Discretizing each of n state variables into k values gives k^n cells. At a coarse $k = 10$ the pendulum needs 10^4 cells and the car 10^6. The helicopter needs 10^{12}, and the official notes' example of a 10-dimensional state at 100 values each needs 10^{20} (right of Figure 18.1). Ng adds that the curse is not only about continuous states. A factory with 100 machines, each in one of k conditions, has k^{100} discrete states, and fitted value iteration can be the better algorithm there too.

17:25 His working rules for when to discretize are worth having as stated. In two or three dimensions, just do it. In four to six, think about it, and it will often still work. The pendulum is fine, and friends of his made a six-dimensional bicycle model work "if you put some work into it". The usual trick in this range is to discretize unevenly, spending more values on the variables the policy is most sensitive to, because every value saved is saved in the exponent. At seven or eight dimensions he becomes nervous. Above that he would very seriously look for an algorithm that does not discretize. The official notes draw the line a little lower: they promise reliable results in one or two dimensions, four with care, and six only with luck.

18.3. Getting a model

23:20 Fitted value iteration needs a *model*, or *simulator*: a black box that takes a state s and an

action a and returns a next state s' drawn from P_{sa}. For simplicity the action space is taken to be discrete. Ng points out that this is usually a mild assumption, because robots tend to have far fewer controls than state variables. The car has a six-dimensional state but two controls, steering and braking. The helicopter has twelve state variables and four controls, a stick under each hand that moves in two directions. The pendulum has four and one.

26:13 One way to get a simulator is physics. For the pendulum, Newtonian mechanics gives $\dot{s}$ as a function of the state, the push and the masses and length of the parts. Ng shows the equation only as decoration. Stepping it forward as $s' = s + \Delta t\, \dot{s}$ with $\Delta t = 0.1$ s simulates the system at 10 Hz. In practice nobody derives these by hand. You download an open-source physics engine and give it the dimensions. He once built a four-legged walking robot's controller this way.

29:15 The way he uses more often is to *learn* the model from data. Take the helicopter into a field and have a human pilot fly it around while GPS, accelerometers and a compass log the state ten times a second, along with the stick positions. One flight gives a trajectory $s_0, a_0, s_1, a_1, \ldots, s_T$; repeat it m times. Predicting s_{t+1} from (s_t, a_t) is now a supervised learning problem. Its linear version is

$$s_{t+1} \approx A s_t + B a_t, \qquad \min_{A,B} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - (A s_t^{(i)} + B a_t^{(i)}) \right\|^2, \tag{18.1}$$

with $A \in \mathbb{R}^{n \times n}$ square and $B \in \mathbb{R}^{n \times d}$ tall and thin when the action has $d < n$ components. Ng's verdict from experience is that this is "actually not a terrible model" for a helicopter flying slowly and roughly level, and a poor one for fast, aggressive manoeuvres. Nonlinear features $\phi(s)$ and $\phi'(a)$ can replace the raw state and action when the linear model is not enough.

Equation 18.1 looks like a new kind of regression because A and B are matrices. Ng's remark that it is the same as running ordinary linear regression n times, once per state variable, is right: the squared norm is a sum over the n components, and each component involves only one row of $[A\ B]$. The notes add that the least-squares fit is also the maximum-likelihood one. That holds even if the noise in different state variables is correlated, which is the realistic case, because every equation shares the same regressors. The generator confirms all three routes on a simulated two-dimensional system with correlated noise. The joint fit, the n separate regressions and the Gaussian maximum-likelihood fit with the true covariance agree to 10^{-15}.

35:37 A student asks whether m short flights differ from one long one. For the algorithm, Ng says, they do not. In practice the helicopter burns fuel and gets lighter, and the wind changes, so several flights average over conditions a single one would not cover.

37:41 Having fitted A and B, there are two simulators to choose from. The *deterministic* one sets $s_{t+1} = A s_t + B a_t$. The *stochastic* one adds $\varepsilon_t \sim \mathcal{N}(0, \Sigma)$, whose covariance can be estimated from the residuals, and draws a fresh ε_t on every step. This whole approach of learning a model and then planning in it is *model-based* reinforcement learning. The alternative, *model-free* RL, runs the learning algorithm on the real system directly. Ng's view in 2018 is that model-based methods were taking off faster in robotics. A physical robot cannot afford to be crashed thousands of times while it learns, while a video game or a board game is its own perfect simulator and can be.

41:45 He ends this part with a lesson from many robots. A policy optimised against a deterministic simulator tends to be brittle. It exploits details of the simulator that the real machine does not share, and that is how a five-legged robot can walk beautifully in a video and fall over in the lab. A simulator is always a little wrong: it misses a wind gust, or banks a degree too far in a turn. If the policy has to work in a slightly noisy simulator, it is much more likely to work on the real robot. His advice is to always add noise. The exact distribution matters much less than

remembering to add it. (He mentions in passing that his own PhD thesis used reinforcement learning to fly helicopters, and that he lived through the crashes.)

18.4. Fitted value iteration

46:13 With a simulator in hand, the value function is approximated the way a price is approximated in linear regression:

$$V(s) = \theta^\top \phi(s), \tag{18.2}$$

where $\phi(s)$ is a vector of features of the state chosen by hand. Ng's advice on choosing them is to remember what a value is: the expected payoff from that state, so features should say how well the robot is doing. For the pendulum, a pole leaning far to the right is bad *unless* the cart is moving fast to the right to catch it. So the product of the pole angle and the cart velocity is a natural feature, alongside x, $\dot{x}$ and squares. A model-based method also makes overfitting less of a worry than in ordinary supervised learning. The training data come from the simulator and cost nothing, so a generous set of features, some of which turn out useless, can be fitted from as many samples as the computer can afford.

50:14 The algorithm starts from value iteration's update in continuous form,

$$V(s) := R(s) + \gamma \max_a \mathbb{E}_{s' \sim P_{sa}}\big[V(s')\big], \tag{18.3}$$

and replaces the two things it cannot do. It cannot visit every state, so it visits a sample. It cannot store an arbitrary V, so it fits Equation 18.2 to the targets that Equation 18.3 produces at the sampled states:

1. Sample states $s^{(1)}, \dots, s^{(m)}$ at random, and set $\theta := 0$.
2. Repeat:
   a. For each $i = 1, \dots, m$ and each action a: draw $s'_1, \dots, s'_k \sim P_{s^{(i)}a}$ from the simulator and set $q(a) = \frac{1}{k}\sum_{j=1}^{k}\big[R(s^{(i)}) + \gamma V(s'_j)\big]$.
   b. Set $y^{(i)} = \max_a q(a)$.
   c. Set $\theta := \arg\min_\theta \frac{1}{2}\sum_{i=1}^{m}\big(\theta^\top \phi(s^{(i)}) - y^{(i)}\big)^2$.

56:46 Each $q(a)$ is a sample-average estimate of the expectation in Equation 18.3, so $y^{(i)}$ is an estimate of the whole right-hand side at $s^{(i)}$. 58:04 Discrete value iteration would now set $V(s^{(i)}) := y^{(i)}$, but a linear function cannot have its values set one point at a time. The best it can do is come as close as possible to all the targets at once, which is a least-squares fit. Step (c) is linear regression with $s^{(i)}$ in the role of x and $y^{(i)}$ in the role of the price. A student asks where $V(s'_j)$ in step (a) comes from. 1:08:35 It is $\theta^\top \phi(s'_j)$ with the θ from the previous iteration, exactly as discrete value iteration uses the previous sweep's values.

1:00:51 Two questions clarify the bookkeeping. The m here is the number of sampled states and has nothing to do with the m flights used to learn the model. Ng's picture is one team flying the helicopter a thousand times and publishing a model, and another team running fitted value iteration on it. 1:03:06 He then asks the class how the algorithm simplifies with a deterministic simulator. The answer is that $k = 1$ suffices, since all k samples would be the same next state. The generator checks this mechanically: with the noise switched off, $k = 1$ and $k = 5$ give the same θ to within 10^{-14}. On how large m should be, 1:05:38 he reasons as for any regression.

With 50 features, 500 samples might do. But samples are free, so there is no harm in 5,000 or 100,000 beyond running time. 1:09:07 Finally, nothing requires the regression to be linear. Locally weighted regression works, and replacing it with a neural network is, he says only half joking, how you get *deep* reinforcement learning.

On the one-dimensional MDP of Figure 18.1, with 200 sampled states, $k = 10$ and five polynomial features, fitted value iteration lands within 1.8 of V^* everywhere. That is as good as the 48-cell discretization, using five numbers instead of 48. With only three features, $1, s, s^2$, the error is 7.1. The true V^* is not exactly quadratic, and a fit can only be as good as its features.

What the lecture does not say, and the notes do in one sentence, is that fitted value iteration "cannot be proved to always converge". Discrete value iteration converges because its update is a contraction in the max norm (Section 17.3). A least-squares fit is not a max-norm operation, and composing the two can expand distances instead of shrinking them. Tsitsiklis and Van Roy give the smallest example, reproduced in Figure 18.2. There are two states with no actions and no rewards; state 1 moves to state 2, which stays put, so $V^* = 0$. There is one feature, $\phi = 1$ in state 1 and $\phi = 2$ in state 2. The targets are $y = \gamma \cdot 2\theta$ for both states, and the least-squares fit of $\theta'\phi$ to them is

$$\theta' = \frac{1 \cdot 2\gamma\theta + 2 \cdot 2\gamma\theta}{1^2 + 2^2} = \frac{6\gamma}{5}\,\theta,$$

so θ grows geometrically whenever $\gamma > 5/6$. This happens although the features represent V^* exactly at $\theta = 0$. The cause is that the fit weights both states equally while the dynamics send everything to state 2. The algorithm as written starts at $\theta := 0$, and in this example it would sit there for ever. Any nonzero start, or any noise in the targets, sets the divergence going.

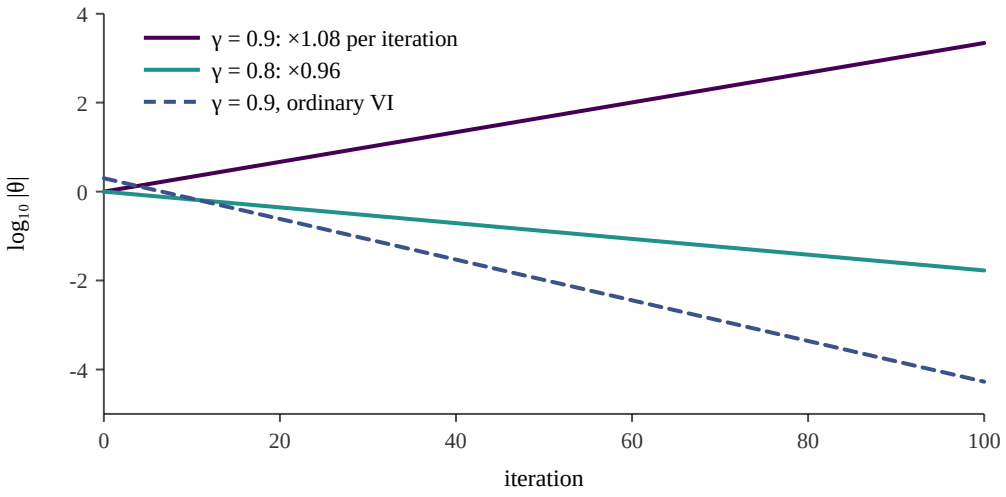


Figure 18.2.: Fitted value iteration on Tsitsiklis and Van Roy's two-state chain, started at $\theta = 1$. At $\gamma = 0.9$ the parameter grows by a factor $6\gamma/5 = 1.08$ per iteration and passes 1,000 after 90; at $\gamma = 0.8$ it shrinks. Ordinary value iteration on the same two states converges at $\gamma = 0.9$.

None of this makes the algorithm unusable. The notes say it usually converges or nearly converges in practice. At the start of lecture 19 Ng answers a student who asked whether it really works: it does, on a real helicopter at low speed. The counterexample is a reason to watch θ across iterations rather than trust it.

> **In the official notes**
>
> Discretization, simulators, learning a linear model and fitted value iteration are in cs229-notes12, section 4, which also writes the update with an integral over s' and states the convergence caveat.

18.5. From a value function to an action

1:10:37 Fitted value iteration returns θ, and π^* is only implicit in it:

$$\pi(s) = \arg\max_a \mathbb{E}_{s' \sim P_{sa}}\big[V(s')\big].$$

On a helicopter this has to be computed ten times a second. The obvious method is to estimate the expectation from k simulator samples, as in training. Ng is reluctant to do that. He does not like the idea of a self-driving car with a random number generator in its control loop, where an unlucky draw could pick a bad action. 1:13:35 The common fix uses the form most simulators have, $s_{t+1} = f(s_t, a_t) + \varepsilon_t$. Train with the noise, so that the policy is robust, but at deployment set $\varepsilon_t = 0$ and $k = 1$:

$$\pi(s) = \arg\max_a V\big(f(s, a)\big). \tag{18.4}$$

Each tenth of a second the controller simulates one step for every action, evaluates V at the predicted next states, and takes the best. Ng admits this is not rigorously justified. There is one case where it is exact. If V is quadratic and the noise is additive with a distribution that does not depend on the action, then

$$\mathbb{E}\big[V(f + \varepsilon)\big] = V(f) + \tfrac{1}{2}\operatorname{tr}(H\Sigma)$$

for zero-mean noise with covariance Σ, where H is the (constant) Hessian of V. The second term is the same for every action, so dropping the noise cannot change the argmax. The generator confirms that the quadratic fit's two read-outs agree at every one of 601 states. For other features the noise term depends on $f(s, a)$, so the two can disagree, though in this toy the quartic fit still never did. This is a first glimpse of a fact lecture 19 makes central: for quadratic values and linear-Gaussian dynamics, the noise does not change the optimal action at all.

1:18:26 Asked about continuous actions, Ng says a car's steering and braking can be discretized without much loss, but a helicopter's controls usually are not. When f is smooth, the maximisation in Equation 18.4 can be done by a fast numerical optimiser in real time, which is the territory of *model predictive control*. 1:19:24 And to a student asking how the controller knows where it has got to after acting: it measures. A tenth of a second later a new GPS reading and new sensors give the new state, wind and model errors included, and the argmax is taken again from there.

19. Finite-Horizon MDPs and the Linear Quadratic Regulator

Lecture 19

Lecture video: Lecture 19: Reward Model and Linear Dynamical System · 1:21:07

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Fitted value iteration (Section 18.4) handles continuous states by approximating the value function. This lecture finds the one important case where no approximation is needed: if the dynamics are linear and the reward quadratic, the value function is *exactly* quadratic, the optimal policy is *exactly* linear, and both come out of a short matrix recursion however many state variables there are. Getting there takes two small generalizations of the MDP first, rewards that depend on the action and a finite horizon, because the algorithm, the *linear quadratic regulator* (LQR), is most naturally a backward pass over a fixed number of time steps. Ng's summary is that LQR applies to a narrow set of problems, but when it applies it is the algorithm to use.

19.1. Rewards that depend on the action

4:16 So far the reward has been a function of the state alone. Letting it depend on the action too, $R : S \times A \to \mathbb{R}$, makes the payoff

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \cdots,$$

and it lets a problem say that actions have costs. A robot in a maze can be charged for moving and nothing for standing still. A helicopter can be charged for yanking the sticks hard, which makes it jerk around. A route planner can charge each road segment its own cost, and a factory can price sending a maintenance crew differently from doing nothing. 5:55 Bellman's equation changes in one visible way:

$$V^*(s) = \max_a \Big[R(s, a) + \gamma \sum_{s'} P_{sa}(s') V^*(s') \Big], \tag{19.1}$$

with the max now outside the immediate reward, since that reward depends on the action being chosen. A student checks that the max covers the whole bracket, and it does. Value iteration uses Equation 19.1 as its update unchanged, and the optimal policy is the argmax of the same bracket. The reward depends on the current state and action, not on the next state. Ng notes that other formulations exist and this is one convenient one.

19.2. Finite horizons and policies that watch the clock

11:00 The second generalization replaces the discount factor with a *horizon* T. The MDP runs for T steps and then stops, and the payoff is the finite sum

$$R(s_0, a_0) + R(s_1, a_1) + \cdots + R(s_T, a_T),$$

usually without discounting, though discounting is still allowed. Ng's example is a model helicopter with thirty minutes of fuel: the aim is to collect as much reward as possible before the clock runs out.

13:11 A finite horizon has a consequence that is easy to miss. The best action can depend on the time. Put a robot between a small $+1$ nearby and a large $+10$ farther off. With plenty of time it should go for the $+10$. With two or three steps left it should grab the $+1$, because the $+10$ is out of reach. So the optimal policy is written $\pi_t^*(s)$, and is called *non-stationary*. 15:46 Ng points out what was implicit until now: in the infinite-horizon discounted setting of lectures 16 and 17 the optimal policy is *stationary*, the same at every step, because the future always looks the same.

16:31 Once the policy may change over time, so may the world. A non-stationary MDP has $P_{sa}^{(t)}$ and $R^{(t)}(s, a)$ indexed by time. Ng gives three examples. A long-haul airliner can burn a third of its take-off weight in fuel, so it handles differently on landing. A drive to San Francisco at 5 pm runs into rush hour, and into mud if the forecast says rain. And in a factory running round the clock, labour costs and availability change with the shift.

19.3. Solving it backwards

21:11 Define $V_t^*(s)$ as the best expected payoff from state s at time t to the end. With nothing left after the last step, the base case is just the best immediate reward,

$$V_T^*(s) = \max_a R(s, a),$$

23:27 and each earlier step looks one step ahead to the next:

$$V_t^*(s) = \max_a \Big[R(s, a) + \sum_{s'} P_{sa}(s') V_{t+1}^*(s') \Big], \qquad \pi_t^*(s) = \arg\max_a \Big[\cdots \Big]. \tag{19.2}$$

This is no longer value iteration in the sense of repeating an update until it converges. It is *dynamic programming*: compute V_T^*, then V_{T-1}^* from it, then V_{T-2}^*, down to V_0^*, once each, and read π_t^* off at every step. Time-varying $P^{(t)}$ and $R^{(t)}$ slot into Equation 19.2 at no extra cost.

28:31 A student asks whether letting $T \to \infty$ recovers ordinary value iteration. The two are closely related, but undiscounted sums can grow without bound as T grows. That is one reason discounting exists. If $|R| \le R_{\max}$, a discounted value is at most $R_{\max}/(1-\gamma)$ in size, and a finite-horizon one at most $(T+1)R_{\max}$.

30:30 Ng sketches the $+1/+10$ corridor with deterministic moves. With two steps left, the values read like "10, 10, 10, 1, 1, 1, 0, 0, 0": cells within reach of the $+10$, cells only within reach of the $+1$, and cells that can reach neither. He suggests building the little simulator to watch the policy change as the clock runs down. Figure 19.1 does that for an eleven-cell corridor of our own, with the $+10$ in cell 0, the $+1$ in cell 8, and a state-action reward of -0.1 for each

move, so that it uses both of this lecture's generalizations. The generator checks the dynamic programme against brute force over every action sequence up to seven steps. Cell 5 shows the non-stationarity in full. With one or two steps left it stays put, since nothing is reachable and moving costs. With three or four it heads right for the +1. With five or more it heads left for the +10.

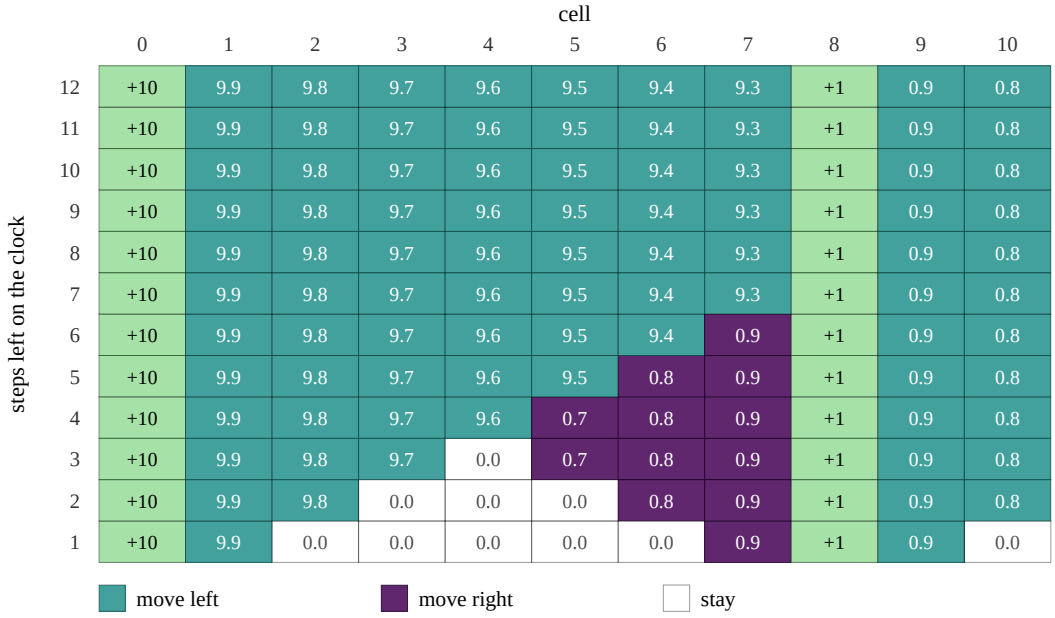


Figure 19.1.: The optimal action in each cell of a corridor, for each number of steps left, from one at the bottom to twelve at the top, with the optimal value in each square. Cell 0 holds +10, cell 8 holds +1, and each move costs 0.1. Where moving now and moving later tie, as they often do when waiting is free and nothing is discounted, the figure shows the move.

The figure's caption hides one more lesson. Without discounting and with waiting free, many cells have *several* optimal actions. Leaving for the +10 now or one step later gives the same total if there is time either way. Discounting is what makes sooner strictly better.

19.4. The LQR model

32:04 LQR is a finite-horizon MDP with continuous states $S = \mathbb{R}^n$, continuous actions $A = \mathbb{R}^d$, and two specific assumptions. The first is that the dynamics are linear with Gaussian noise:

$$s_{t+1} = As_t + Ba_t + w_t, \qquad w_t \sim \mathcal{N}(0, \Sigma_w), \tag{19.3}$$

with $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times d}$. 34:27 Ng apologises for the notation. A is now both the set of actions and a matrix, because LQR comes from control theory in electrical and mechanical engineering and reinforcement learning from computer science, and when the two literatures met, their letters clashed. The same thing happens to V below. 36:30 The second assumption is a quadratic reward,

$$R(s, a) = -(s^\top U s + a^\top V a), \qquad U \succeq 0,\ V \succeq 0, \tag{19.4}$$

with $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{d\times d}$ positive semi-definite, so both terms are costs. For a helicopter that should hover at the origin, $U = I$ and $V = I$ give $R = -\|s\|^2 - \|a\|^2$. That penalises drifting from the hover point and penalises large control inputs. Time-varying A_t, B_t, U_t, V_t are allowed but not needed for the derivation.

40:43 Where do A and B come from? The first way was lecture 18's (Equation 18.1): fly the helicopter, log m trajectories, and fit A and B by least squares. Ng says again that this gives a good model for slow flight.

19.5. Linearizing a nonlinear model

43:02 The second way starts from a nonlinear simulator $s_{t+1} = f(s_t, a_t)$, from a physics engine or "a friend with a physics degree". Ng tried the inverted pendulum's equations himself and found them complicated. To make it linear, he draws the one-dimensional case. 45:31 Plot $s_{t+1} = f(s_t)$ against s_t, pick a point $\bar{s}_t$, and replace f with its tangent line there:

$$s_{t+1} \approx f(\bar{s}_t) + f'(\bar{s}_t)\,(s_t - \bar{s}_t).$$

49:05 The tangent is good near $\bar{s}_t$ and poor far away, so $\bar{s}_t$ should be where the system spends its time: the hover point for a helicopter, the upright pole at rest for the pendulum. 50:37 With an action as well, linearize around a typical pair $(\bar{s}_t, \bar{a}_t)$ using the Jacobians:

$$s_{t+1} \approx f(\bar{s}_t, \bar{a}_t) + \nabla_s f(\bar{s}_t, \bar{a}_t)\,(s_t - \bar{s}_t) + \nabla_a f(\bar{s}_t, \bar{a}_t)\,(a_t - \bar{a}_t). \tag{19.5}$$

53:20 This is affine rather than linear, because of the constant terms. Ng's fix is to add an intercept to the state, a component fixed at 1. Then Equation 19.5 takes the form $s_{t+1} = As_t + Ba_t$ after "some algebraic munging". When the linearization point is the origin and the system rests there, $f(0, 0) = 0$, as for the upright pendulum, the constants vanish and no intercept is needed.

Figure 19.2 does this for a pendulum on a motorised pivot, with the angle θ measured from upright and a torque as the action. The only nonlinear term is gravity, $(g/l)\sin\theta$, and linearizing at $\bar{\theta} = 0$ replaces it with $(g/l)\theta$. The generator checks the hand-derived A and B against finite differences. The line is 1% too strong at 0.25 rad, 4% at 0.5 rad, 19% at 1 rad and 50% at 1.5 rad. The right panel runs the LQR controller designed on the linear model on the true pendulum. From 0.3 rad the real and predicted trajectories coincide. From 1.4 rad they come apart by up to 0.11 rad, but the controller still brings the pole up. Feedback forgives a model that is wrong in the right direction. Because the measured state is fed back every step, the controller corrects its own model errors as they appear. In this toy, with unlimited torque, it recovers even from 3 rad. A real motor's torque limit would end that long before.

19.6. The value function is exactly quadratic

56:05 Now run the backward recursion Equation 19.2 on the LQR model. With continuous states, the sum over s' becomes an expectation over $s_{t+1} \sim \mathcal{N}(As_t + Ba_t, \Sigma_w)$. 57:07 The base case is immediate:

$$V_T^*(s_T) = \max_{a_T} [-s_T^\top U s_T - a_T^\top V a_T] = -s_T^\top U s_T,$$

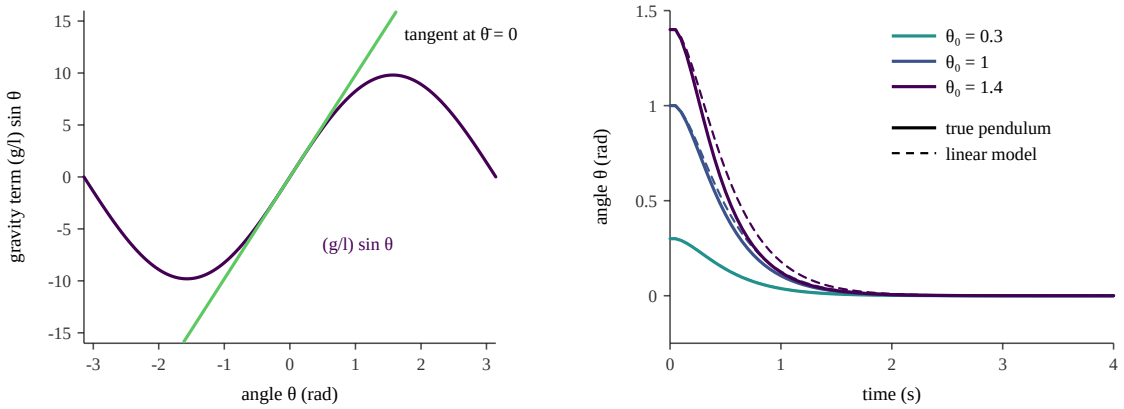


Figure 19.2.: Left: the pendulum's gravity term $(g/l)\sin\theta$ and its tangent at the upright position. Right: the pole angle under the LQR controller designed on the linear model, on the true pendulum (solid) and as the linear model predicts it (dashed), from three starting angles.

because $a^\top V a \geq 0$, so the best last action is $a_T = 0$. A student notices that $s^\top U s$ is also non-negative. It is, but it cannot be chosen: only a_T is maximised over.

59:09 The inductive step carries the whole method. Suppose

$$V^*_{t+1}(s) = s^\top \Phi_{t+1} s + \Psi_{t+1}$$

for some matrix $\Phi_{t+1} \in \mathbb{R}^{n\times n}$ and scalar Ψ_{t+1}. This holds at $t+1 = T$ with $\Phi_T = -U$ and $\Psi_T = 0$. 1:04:23 Then

$$V^*_t(s_t) = \max_{a_t}\Big[-s_t^\top U s_t - a_t^\top V a_t + \mathbb{E}_{s_{t+1}\sim\mathcal{N}(As_t+Ba_t,\,\Sigma_w)}\big[s_{t+1}^\top \Phi_{t+1} s_{t+1}\big] + \Psi_{t+1}\Big].$$

The expectation of a quadratic under a Gaussian is a quadratic of the mean plus a constant, $\mathbb{E}[s^\top \Phi s] = \mu^\top \Phi \mu + \operatorname{tr}(\Sigma_w \Phi)$. So the bracket is a quadratic function of a_t, and since $\Phi_{t+1} \preceq 0$ and $V \succeq 0$ it is concave. 1:06:57 Setting its derivative to zero gives

$$a^*_t = -(B^\top \Phi_{t+1} B - V)^{-1} B^\top \Phi_{t+1} A\, s_t = L_t\, s_t. \tag{19.6}$$

1:08:03 Ng stops on this. The optimal action is a *linear* function of the state. He stresses that this is not a function-approximation claim. Nobody fitted a line to the optimal action. Of all functions of the state, linear or not, the best one turns out to be linear. Substituting a^*_t back into the bracket gives another quadratic in s_t, which completes the induction. 1:09:45 The coefficients obey the *discrete Riccati equations*:

$$\begin{aligned} \Phi_t &= A^\top\Big(\Phi_{t+1} - \Phi_{t+1} B (B^\top \Phi_{t+1} B - V)^{-1} B^\top \Phi_{t+1}\Big) A - U, \\ \Psi_t &= \Psi_{t+1} + \operatorname{tr}(\Sigma_w \Phi_{t+1}). \end{aligned} \tag{19.7}$$

1:11:04 The whole algorithm is to set $\Phi_T = -U$ and $\Psi_T = 0$, then apply Equation 19.7 and Equation 19.6 for $t = T-1$ down to 0, and finally act with $\pi^*_t(s) = L_t s$. 1:14:08 What impresses Ng is that nothing in it is approximate. The approximation happens once, when a real system is

forced into the form of a linear model with a quadratic cost, by regression or by linearization. After that, the value function really is quadratic and the policy really is linear, and the recursion computes both exactly.

The generator tests each claim by a second route on a cart with position and velocity at 10 Hz and a 60-step horizon. At 20 random times and states, a numerical maximisation of the Bellman bracket lands on $L_t s$ and on $s^\top \Phi_t s + \Psi_t$ to within 10^{-6}. None of 400 perturbed policies beats $L_t s$, and half of them add nonlinear terms of the state and time.

> ⚠ Two signs the 2018 notes get wrong
>
> cs229-notes13 prints the optimal action without the leading minus sign in Equation 19.6. It also prints $\Psi_t = -\operatorname{tr}(\Sigma_t \Phi_{t+1}) + \Psi_{t+1}$, and the board in the lecture shows the same minus sign. Both are wrong, and the notes' own footnote, $\mathbb{E}[w^\top \Phi w] = \operatorname{tr}(\Sigma\Phi)$, gives the plus sign. The noise should make the value *worse*, and since $\Phi_{t+1} \preceq 0$, only $+\operatorname{tr}$ does that. Monte Carlo settles it. From $s_0 = (1, 0)$ with $\Sigma_w = 0.25I$, 4,000 rollouts of the cart under L_t average -168.8 ± 1.1. Equation 19.7 predicts -170.4, and the printed sign predicts $+152.3$, a positive value for a problem whose rewards are all negative. The action's sign matters even more. Without the minus, the controller pushes the cart away from the origin, and the payoff of a single noiseless run is -1.4×10^{14}. The notes also write B_t where $B_t^\top$ is meant, and V_t and W_t for the same matrix.

> ℹ In the official notes
>
> Finite-horizon MDPs, LQR and linearization are cs229-notes13, sections 1, 2 and 3.1. Section 3.2 goes on to *differential dynamic programming*, which re-linearizes around a trajectory instead of a single point. Section 4 covers LQG, LQR when the state is observed only through noise, with the Kalman filter. The lecture covers neither.

19.7. The noise does not change the policy

1:15:22 Ng ends with a property he calls fun, and warns against generalizing. Look at what each quantity depends on. L_t in Equation 19.6 needs Φ_{t+1} but not Ψ_{t+1}, which makes sense: adding a constant to every future value cannot change which action is best. Φ_t in Equation 19.7 needs only Φ_{t+1}. 1:17:36 The noise covariance Σ_w appears in exactly one place, the update for Ψ. Delete the Ψ line from the code and the optimal policy is computed perfectly without Σ_w ever being used. **The optimal LQR policy does not depend on the noise.** The value does: a helicopter in a gale does worse than one in still air. But the best response to any state is the same in both.

Figure 19.3 shows both effects on the cart. The gains computed with $\Sigma_w = 0$ and with $\Sigma_w = 0.25I$ are identical to the last bit, while Ψ_0 goes from 0 to -161.4. The figure also shows the finite horizon at work. For the first half of the horizon the gain is flat at $L \approx (-2.76, -2.51)$, a stationary policy in all but name. Over the last 25 or so steps it relaxes to zero, because there is less and less future in which a correction would pay off. That is the clock-watching of Figure 19.1 in continuous form.

1:19:00 Ng draws two lessons. The narrow one is that, when applying LQR, there is no need to work hard at estimating Σ_w. The broader one is about practice. For most MDPs the noise *does* change the policy, and the form of the noise matters. But in his experience of building robots, remembering to add noise matters more than getting its size exactly right. Ten per cent

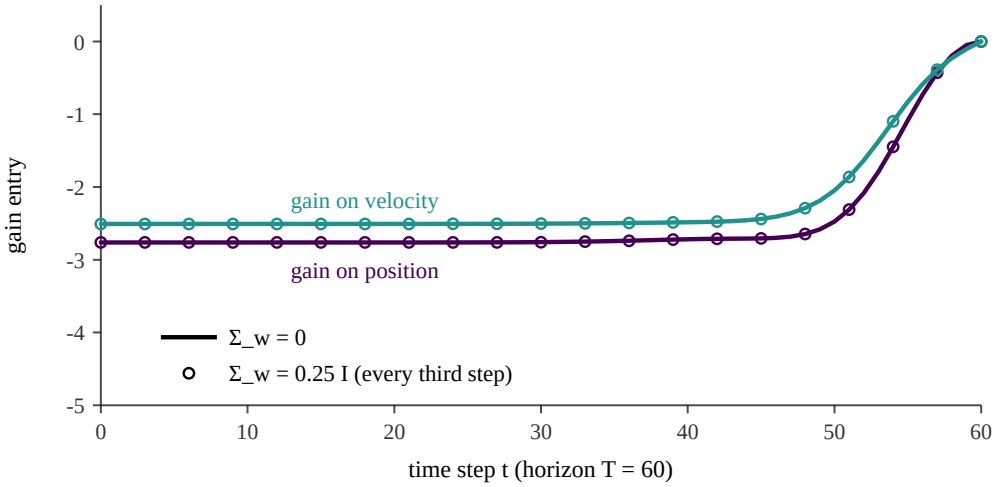


Figure 19.3.: The two entries of the LQR gain L_t for the cart, on position and on velocity, against the time step for a 60-step horizon. Lines are for $\Sigma_w = 0$; circles, drawn every third step, are for $\Sigma_w = 0.25I$.

more or less rarely matters; a factor of two often does. That is the same advice as lecture 18's (Section 18.3), now with one case where it can be proved.

20. RL Debugging and Direct Policy Search

Lecture 20

Lecture video: Lecture 20: RL Debugging and Diagnostics · 1:12:43

Based on the lecture by Andrew Ng — CS229, Stanford University, Autumn 2018.

Every reinforcement-learning method in the course so far has been indirect: estimate the value function, then read the policy off it with an argmax. The last lecture introduces the other family. *Direct policy search* writes the policy down as a parametrised function, the way logistic regression writes down a hypothesis, and tunes its parameters by gradient ascent on the expected payoff, never computing a value function at all. The algorithm Ng derives, REINFORCE, gets an unbiased estimate of that gradient from a single run of the system. Unbiased is not the same as useful: the estimate is so noisy that REINFORCE is notoriously slow, and part of that noise comes from terms that average to zero and can be dropped. Before any of that, Ng finishes a promise from lecture 12 and restates the helicopter diagnostics in the notation the course has since introduced.

20.1. Debugging the helicopter, in RL notation

4:05 Lecture 12 walked through the diagnostics for Ng's autonomous helicopter before the course had any reinforcement learning to describe them with (Section 12.5). The setup is now familiar. Build a simulator, a learned model $s_{t+1} = As_t + Ba_t$ or something richer, as in Section 18.3. Choose a reward function, such as the negative squared distance from where the helicopter should be. Run a reinforcement-learning algorithm on the finite-horizon problem to get a policy π_{RL} that maximises $\mathbb{E}[R(s_0) + \cdots + R(s_T)]$ in the simulator. The first time, the controller flies much worse than the human pilot, and there are three things to try: improve the simulator, change the reward, or improve the RL algorithm. Ng is concrete about the last one. It means changing the discretization of a continuous state, or replacing the linear function approximator in fitted value iteration with a richer one such as a neural network.

The diagnostics are those of lecture 12 with the sign turned round, since the algorithm now maximises a payoff rather than minimising a cost. If π_{RL} flies well in simulation and badly in reality, the simulator is at fault. Otherwise, 8:28 record the human pilot's flight and score it with the same reward. If the human's payoff is higher, the algorithm failed to maximise the sum of rewards, because here is a policy that does better. If the algorithm's payoff is higher and it still flies worse, maximising this reward is not the same as flying well, and the reward is what needs work.

13:12 A student asks whether the tests have an order. Ng checks the simulator first and turns to the other two only once it seems adequate. He also qualifies the whole procedure: the diagnostics look like a rigid decision rule, but in practice the answers are graded. The simulator is "kind of good", the human beats the algorithm by a little, and the team looks at all three pieces

of evidence together and decides where the next month goes. What he insists on is that the bottleneck moves as the project progresses, so the diagnostics are re-run after each round of work rather than once, and teams that prioritise this way move faster than teams that go by gut feel.

20.2. Searching for the policy directly

17:01 Ng shows one more example of the indirect route first. Zico Kolter's quadruped crossing rough terrain relied on an approximate value function that scored, for a given height map of the terrain, how good each position of the robot was; the robot moved towards higher value. That is value iteration's way of thinking: learn V^*, or an approximation to it, then act greedily with $\pi^*(s) = \arg\max_a \mathbb{E}_{s' \sim P_{sa}}[V^*(s')]$ (Section 17.4 and Section 18.5).

18:25 *Direct policy search* goes straight for the policy, with no value function in between. Its first step is the same as the first step of supervised learning. When logistic regression was introduced, Ng recalls, the form $h_\theta(x) = g(\theta^\top x)$ was pulled out of a hat (Section 3.5), and the learning algorithm's job was then to find the best θ within that form. Policy search likewise starts by choosing a class of policies, parametrised by θ, and its job is to find the best member of the class.

The running example is the inverted pendulum, a pole on a hinge atop a cart, with two actions: accelerate left or accelerate right. The state is

$$s = \begin{bmatrix} 1 \\ x \\ \dot{x} \\ \phi \\ \dot{\phi} \end{bmatrix},$$

the cart's position and velocity and the pole's angle and angular velocity, with a 1 for an intercept. The angle is ϕ rather than the usual θ, which is about to be taken by the parameters.

20.3. Stochastic policies

22:51 The policies of lectures 16 to 19 were deterministic, a map from states to actions. Policy search uses a *stochastic policy*, a function $\pi(s, a)$ giving the probability of taking action a in state s:

$$\pi(s, a) \geq 0, \qquad \sum_a \pi(s, a) = 1 \quad \text{for every } s.$$

On every step the agent draws its action from $\pi(s, \cdot)$. 24:38 For the pendulum, Ng's choice is logistic:

$$\pi_\theta(s, \text{right}) = \frac{1}{1 + e^{-\theta^\top s}}, \qquad \pi_\theta(s, \text{left}) = 1 - \pi_\theta(s, \text{right}). \tag{20.1}$$

25:47 To see that this can express sensible behaviour, take $\theta = (0, 0, 0, 1, 0)$. Then $\theta^\top s = \phi$, and the probability of accelerating right is $1/(1 + e^{-\phi})$: the further the pole leans to the right, the

likelier the cart is to accelerate right and get back under it. This ignores where the cart is, and a cart that keeps chasing the pole will run off the end of its track. 27:18 With $\theta = (0, -0.5, 0, 1, 0)$ the position enters with a negative weight, so the further right the cart already is, the less likely it is to accelerate further right. The general problem is to find the five numbers that trade off position, velocity, angle and angular velocity best, and that is what the algorithm will do.

Figure 20.1 plots these policies. One thing the board example hides is scale. With a weight of 1 on an angle measured in radians, a pole tilted half a radian, nearly thirty degrees, still gets accelerated the right way only 62 % of the time: the policy is close to a coin flip over every angle a balancing controller would see. The signs in θ say which way to lean; the magnitude says how decisively, and a weight of 10 turns the same policy into almost a step. The learning algorithm has to find both.

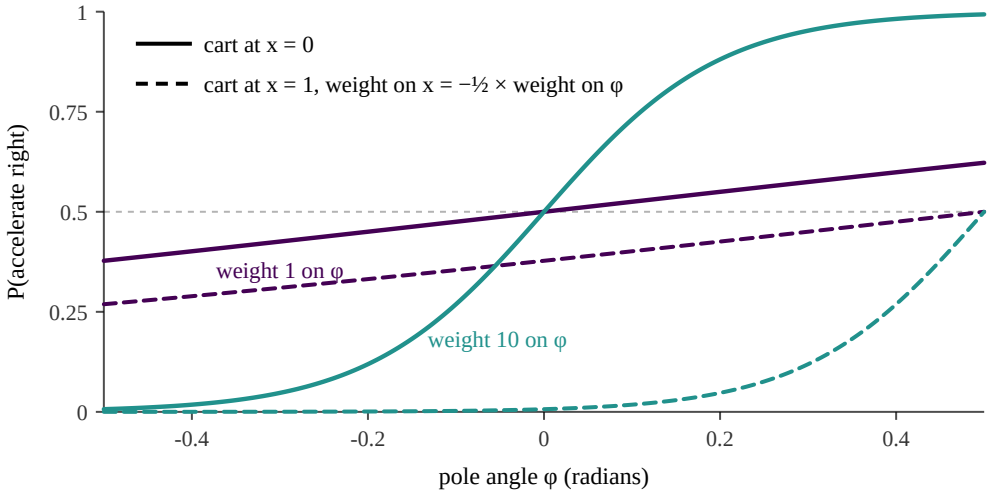


Figure 20.1.: Probability of accelerating right against the pole angle, for the logistic policy Equation 20.1. Purple: Ng's weight of 1 on the angle; teal: the same direction scaled by ten. Solid: the cart at $x = 0$; dashed: the cart at $x = 1$ with a weight of -0.5 times the angle weight on position, which shifts each curve right. Computed in `figures/20-policy-search/make_diagrams.py`.

49:04 Later, answering a question, Ng gives the reason for the switch to stochastic policies. A deterministic policy over two actions is a step function of θ, either left or right, and there is no derivative to follow. Probabilities move smoothly with θ, so the expected payoff becomes differentiable and gradient ascent applies.

20.4. The objective

29:03 The goal is to choose θ to maximise the expected payoff when the policy is run:

$$\max_{\theta} \; \mathbb{E}\left[R(s_0, a_0) + R(s_1, a_1) + \cdots + R(s_T, a_T) \,|\, \pi_\theta\right], \tag{20.2}$$

in the finite-horizon setting of Section 19.2, with rewards that may depend on the action as in Section 19.1. For the pendulum the reward might be -1 whenever the pole falls over and 0 otherwise.

30:15 One difference from value iteration is easy to miss. V^* gives the optimal policy for every starting state at once, so a single π^* is simultaneously best from wherever the system begins. Policy search optimises from one fixed initial state s_0, or one fixed distribution over initial states, and the policy it finds is best for that start. A policy class too small to contain π^* forces a trade-off between states, and the initial distribution decides which states the trade-off favours.

20.5. REINFORCE

32:02 To keep the board manageable, Ng sets $T = 1$: two states and two actions. The expectation in Equation 20.2 is then a sum over every state-action sequence, weighted by its probability:

$$\mathbb{E}[\text{payoff}] = \sum_{s_0,a_0,s_1,a_1} P(s_0)\,\pi_\theta(s_0,a_0)\,P_{s_0a_0}(s_1)\,\pi_\theta(s_1,a_1)\,[R(s_0,a_0)+R(s_1,a_1)]\,. \tag{20.3}$$

The initial distribution $P(s_0)$ and the transition probabilities $P_{s_0a_0}(s_1)$ do not depend on θ; only the two policy factors do. Everything below carries over to longer horizons with longer products and sums.

34:58 The algorithm that climbs this objective is called REINFORCE, after Williams' 1992 paper (Simple statistical gradient-following algorithms for connectionist reinforcement learning). Ng presents its core and notes that the original has a few further refinements, one of which comes up below. Each iteration does three things:

1. Run the system for T steps under the current policy, sampling each action from π_θ, to get a sequence $s_0, a_0, s_1, a_1, \dots, s_T, a_T$.
2. Compute the payoff, the sum of the rewards along it.
3. Update
$$\theta := \theta + \alpha \left[\sum_{t=0}^{T} \frac{\nabla_\theta \pi_\theta(s_t,a_t)}{\pi_\theta(s_t,a_t)}\right] \times \text{payoff}. \tag{20.4}$$

37:48 Each update is random, because it depends on which states the system happened to visit and which actions the policy happened to draw. Ng's picture is the one he drew for stochastic gradient descent on linear regression (Section 2.6): batch gradient descent heads straight for the optimum, while the stochastic version takes a jittery path that heads the right way on average. The claim is that REINFORCE is stochastic gradient *ascent* on the expected payoff: the expected value of the bracketed update is exactly the gradient of Equation 20.3.

20.6. Why the update follows the gradient

40:05 The proof is three lines of calculus. Differentiate Equation 20.3. Inside the sum only the two π_θ factors depend on θ, so the product rule gives two terms:

$$\nabla_\theta \mathbb{E}[\text{payoff}] = \sum_{s_0,a_0,s_1,a_1} P(s_0)\,P_{s_0a_0}(s_1)\,[\nabla_\theta\pi_\theta(s_0,a_0)\,\pi_\theta(s_1,a_1) + \pi_\theta(s_0,a_0)\,\nabla_\theta\pi_\theta(s_1,a_1)] \times \text{payoff}.$$

43:58 Now multiply and divide the first term by $\pi_\theta(s_0,a_0)$ and the second by $\pi_\theta(s_1,a_1)$. Both terms then contain the full product $P(s_0)\,\pi_\theta(s_0,a_0)\,P_{s_0a_0}(s_1)\,\pi_\theta(s_1,a_1)$, which is the probability of the whole sequence, and it factors out:

$$\nabla_\theta \mathbb{E}[\text{payoff}] = \sum_{s_0,a_0,s_1,a_1} P(s_0,a_0,s_1,a_1)\left[\frac{\nabla_\theta \pi_\theta(s_0,a_0)}{\pi_\theta(s_0,a_0)} + \frac{\nabla_\theta \pi_\theta(s_1,a_1)}{\pi_\theta(s_1,a_1)}\right] \times \text{payoff}. \tag{20.5}$$

That is a probability-weighted sum over sequences of exactly the quantity REINFORCE adds to θ in Equation 20.4, so it is the expected value of that update. 48:03 With a small learning rate, many updates average out and the parameters drift along the gradient of the expected payoff. Each ratio $\nabla_\theta \pi_\theta / \pi_\theta$ is $\nabla_\theta \log \pi_\theta$, the *score* of the action, which is the form the update is usually written in today. For the logistic policy Equation 20.1 it is the familiar $(\mathbb{1}\{a = \text{right}\} - \pi_\theta(s, \text{right}))\, s$, the same expression as the logistic-regression gradient (Section 3.6), now weighted by the payoff instead of the label error.

The generator for this chapter checks Equation 20.5 by two routes on a small pole-balancing problem described below. It enumerates every two-step trajectory to compute the expected REINFORCE update exactly, differentiates the expected payoff numerically through a dynamic programme, and finds the two agree to 10^{-8} at twenty random settings of θ.

48:36 A student asks whether the derivation depends on the logistic form. It does not. Nothing in it used anything about π_θ except that it is the probability of the action and that it is differentiable in θ. A softmax over many actions works, and so does a logistic function of elaborate features of the state. 50:48 The same holds for continuous actions, which policy search handles without discretizing. If the action is a real number, the force on the cart or a car's steering angle, a natural policy is $a = \theta^\top s + \varepsilon$ with Gaussian ε; $\pi_\theta(s,a)$ becomes a Gaussian density, and the same update applies with the density in place of the probability.

49:40 Another student asks why not skip all this and train the helicopter by supervised learning on a human pilot's stick inputs, learning the map from state to the pilot's action. Ng thinks that might fly a helicopter slowly without crashing it, but not well. Learning from rewards is what let the controller fly some manoeuvres better than the human pilots did, and imitation gives the controller no signal for doing better than the teacher it copies.

20.7. Why REINFORCE is slow, and what helps

58:14 Ng is blunt about the cost. The expected update is exactly right, but each individual update is very noisy, so REINFORCE needs a small learning rate and a very large number of iterations. Papers that use it routinely report millions of them.

Part of that noise is self-inflicted, and the lecture's form of the update shows where. Equation 20.4 multiplies the score of *every* action by the *whole* payoff, including rewards collected before the action was taken. The action at step t cannot have affected those, and the corresponding terms average to zero. The reason is an identity that holds in every state:

$$\mathbb{E}_{a\sim\pi_\theta(s,\cdot)}\left[\nabla_\theta \log \pi_\theta(s,a)\right] = \sum_a \nabla_\theta \pi_\theta(s,a) = \nabla_\theta \sum_a \pi_\theta(s,a) = \nabla_\theta 1 = 0.$$

A reward from before step t is fixed by the time a_t is drawn, so its product with the score of a_t has mean zero. Those terms contribute nothing to the expected update and add variance to every individual one. Dropping them gives the update usually written for REINFORCE, with each action weighted only by the rewards that follow it:

$$\theta := \theta + \alpha \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(s_t,a_t) \left(\sum_{\tau=t}^{T} R(s_\tau,a_\tau) - b_t\right). \tag{20.6}$$

The same identity licenses b_t, a *baseline*: any quantity that does not depend on the action a_t can be subtracted from the weight without changing the expected update. A good choice is the expected reward-to-go from step t, which turns the weight into a measure of whether the run went better or worse than usual from that point. That is one of the refinements in Williams' original algorithm, whose update is written with a reinforcement baseline from the start.

The generator checks all three versions on a toy that keeps every expectation computable. The pole's tilt is an integer from -2 to 2, and it falls, ending the reward, if it leaves that range. Gravity pushes a tilted pole further over with probability one half; an upright pole wobbles either way with probability a quarter each. Accelerating corrects the tilt by one step with probability 0.6. The reward is 1 for every step the pole is up, over 20 transitions, and the policy is $\pi_\theta(s, \text{right}) = 1/(1 + e^{-(\theta_0+\theta_1 s)})$. At $\theta = 0$, the coin-flip policy, the expected payoff is 5.25 and the best deterministic policy gets 13.50. Averaged over 20,000 runs, all three estimators agree with the numerically differentiated gradient, $\partial J/\partial\theta_1 = 2.34$, within their standard errors. Their variances do not agree. On θ_1, the lecture's version has variance 174, the rewards-that-follow version 51, and with the baseline 19: nine times less noise for the same expected step.

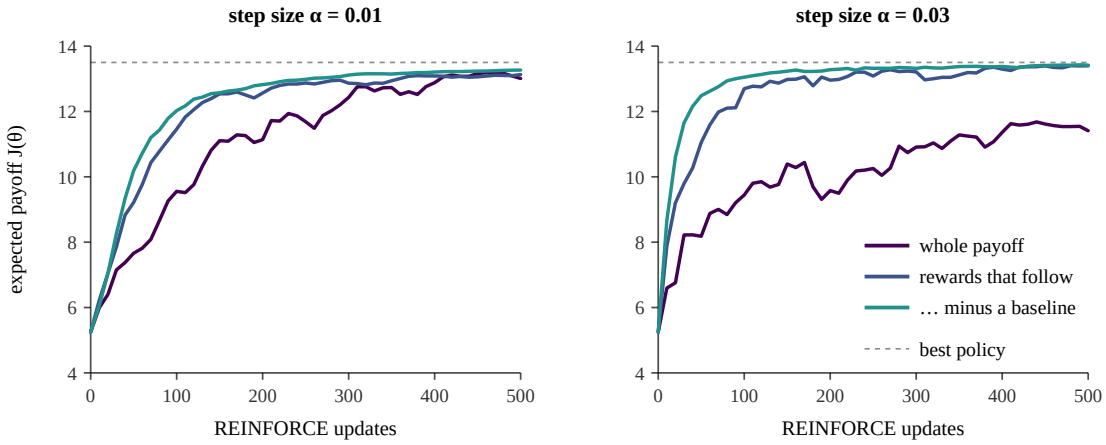


Figure 20.2.: Exact expected payoff of the current policy during REINFORCE on the toy balancing problem, averaged over 20 runs, for the lecture's update Equation 20.4 (whole payoff), Equation 20.6 with $b_t = 0$ (rewards that follow), and Equation 20.6 with b_t the expected reward-to-go. Dashed: the best deterministic policy. Computed in `figures/20-policy-search/make_diagrams.py`.

Figure 20.2 shows what that buys. At step size $\alpha = 0.01$ all three approach the optimum, but the lecture's version needs about 290 updates for its average run to reach 90 % of the best payoff, against 130 and 110 for the other two. At $\alpha = 0.03$ the lower-variance versions get there in about 100 and 40 updates. The lecture's version does not get there at all on average: after 500 updates, 5 of its 20 runs are still below 90 % of the best payoff. Inspecting them shows how it fails. A single large, noisy step throws the intercept θ_0 far from zero, the policy then pushes the same way in almost every state, and with the sigmoid saturated the scores, and so the further updates, are tiny. A larger step size is not available to an estimator this noisy, which is Ng's point about small learning rates, and removing terms that average to zero is the cheapest way to make it available.

20.8. Partial observability

52:24 Ng names two situations in which direct policy search is the better choice. The first is a *partially observable* MDP, or POMDP. Suppose the pendulum has a position sensor and an angle sensor but nothing that measures velocities, and both sensors are noisy. On each step the agent sees not the state but an observation

$$y = \begin{bmatrix} x \\ \phi \end{bmatrix} + \text{noise},$$

two noisy readings of four state variables, and it still has to choose an action.

Value-function methods are stuck here. 55:34 Even with V^* and π^* in hand, the policy $\pi^*(s)$ needs the state, which the agent does not have. There is a large academic literature that extends value-based methods to POMDPs, but in Ng's assessment those algorithms are very complicated and are not widely used in production. Policy search needs no extension. Define the policy on what the agent can see,

$$\pi_\theta(y, \text{right}) = \frac{1}{1 + e^{-\theta^\top y}},$$

and REINFORCE runs unchanged: nothing in Equation 20.5 required the policy's input to be the state, only that the action's probability be a differentiable function of θ. The policy may not be as good as one that sees the state, but the algorithm finds the best one of this form.

57:00 The input can be any features of the history, not just the current reading. A common design is to run a state estimator, such as a Kalman filter, over all the measurements so far, which produces an estimate of the full state vector including the velocities, and feed that estimate to the policy as its features.

20.9. Policy or value function?

58:59 The second consideration is which object is simpler. Ng distinguishes *low-level control* tasks, the ones a skilled human does by feel with a joystick in a fraction of a second: hovering a helicopter, balancing an inverted pendulum, keeping a car in its lane on a straight road. For those a simple function of the state, often close to linear, is a good policy, so a small policy class contains a good answer, and direct policy search is promising. Chess and Go are the opposite: a good move depends on reasoning several steps ahead, no simple function of the board position plays well, and value-function approximation is the more promising route. Driving sits on both sides. Staying in the lane is low-level control; planning to overtake one car before a truck arrives in that lane is multi-step reasoning.

1:02:01 The helicopter went through both phases. Ng's first controllers, including the one in the inverted-flight video at the start of the lecture, came from a direct policy search algorithm, a relative of REINFORCE rather than REINFORCE itself. More complicated manoeuvres called for methods closer to value-function approximation. The two families combine, too: Ng cites AlphaGo, which trained a policy network and a value network and used both in its search.

1:03:20 The course closes with where reinforcement learning was being applied in 2018. Game playing is the visible case, but Ng expects robotics to matter most, and mentions large-machine control in industry, factory optimisation, sequential medical decisions in which each test result decides the next step, and the execution of large stock trades in pieces small enough not to move

the price. The common thread is the one the whole reinforcement-learning unit has been about: a sequence of decisions whose payoff arrives only over time.

> **In the official notes**
>
> The 2018 notes for this part of the course, cs229-notes12, cover MDPs, value and policy iteration and continuous-state MDPs, and end before policy search. The lecture is the source for REINFORCE here; Williams' 1992 paper has the general form, including the baseline.