Lecture notes from the online machine learning tought by Andrew Ng fall 2011

Bjørn Remseth
rmz@rmz.no

Jan. 10 2012
(last revised Feb, 23 2013)

Contents

Introduction

These are my notes for the course in machine learning based on Andrew Ng's lectures, the autumn 2011.

I usually watched the videos while typing notes in LaTeX. I have experimented with various note-taking techniques including free text, mindmaps and handwritten notes, but I've ended up using LaTeX, since it's not too hard, it gives great readability for the math that inevitably pops up in the things I like to take notes about, and it's easy to include various types of graphics. The graphics in this video is exclusively screenshots copied directly out of the videos, and to a large extent, but not completely, the text is based on Ng's narrative. I haven't been very creative, that wasn't my purpose. I did take more screenshots than are actually available in this text. Some of them are indicated in figures stating that a screenshot is missing. I may or may not get back to putting these missing screenshots back in, but for now the are just not there. Deal with it .-)

This document will every now and then be made available on http://dl.dropbox.com/u/187726/machine-learning-notes.pdf. The source code can be cloned on git on https://github.com/la3lma/mlclassnotes.

A word of warning: These are just my notes. They should't be interpreted as anything else. I take notes as an aid for myself. When I take notes I find myself spending more time with the subject at hand, and that alone lets me remember it better. I can also refer to the notes, and since I've written them myself, I usually find them quite useful. I state this clearly since the use of LaTeX will give some typographical cues that may lead the unwary reader to believe that this is a textbook or something more ambitious. It's not. This is a learning tool for me. If anyone else reads this and find it useful, that's nice. I'm happy, for you, but I didn't have that, or you in mind when writing this. That said, if you have any suggestions to make the text or presentation better, please let me know. My email address is la3lma@gmail.com.

1 Linear regression

Normal Equation Noninvertability.:

ocateve: pinv(X'*X) * X' * y is the normal equation $X^T X)^{-1} X' y$ implemented Octaves pseudoinverse function.

If there are linearly depenent features (e.g. feet/meter features, that are linear eqution) If there are too many features. Could cause noninvertability. Deleting features or using *regularization* could do the trick.

If X'X singular then look for redundant features, then delete one of them. Otherwise nuke some features or regularization.

2 Octave:

Prototyping in Octave is very efficient. Only for very large scale implementations do we need lower leve language implementations. Programmer time is incredibly valuable so optimizing that is the first optimization that should always be done.

Prototyping langues: numPy, R, Octave, Matlab, Python. ALl of them slightly clunky than octave.

```
Setting the prompt "PS1('>> ')".    A==B => a matrix of  results of the logical th

Printing:
    disp(sprintf(' 2 decimals: %0.2f', a))

format long -> print with lots of digits
format short -> print with few digits

v = 1:0.1:2   -> all elements from one to two in steps 0.1 (default stepsize 1)

ones(2,3)  -> 2x3 matrix of only ones.
zeros(1,3) -> 1x3 matrix of zeros
rand(3,3) -> 3x3 andom numbers (uniform)
randn(3,3) -> 3x3 random numbers (gaussian distribution)

hist(w) is a  histogram function.  Really nice.

eye(4) -> 4x4 identity matrix (diagonal)

help(eye)

the "size" command returns a 1x2 matrix that is the size of a matrix.

size(A,1) gives the first element (rows), 2 gives column.
length(A) gives the length of the longest dimension.  Usually we use this only for

pwd shows current directory
cd changes directory
```

```
ls lists files on desktop

load('feturesX.dat') loads files with space separated rows and sets a variable fro

the "who" command shows which matrics are available.

whos shows the sizes etc. for the variables.
the clear command removes variable.
saving data.

v=priceY(1:10) the first ten elements of priceY.

save hallo.mat v;

saves the variable v into the file hallo.mat

clear without parameters will remove everything.

save hello.txt  v --ascii

will save as a text (not binary which is the default)

A(3,2) get the A_3^{(2)} element
A(2,:) gets the second row

A([1 3], :) get everything from the first and third rows.

stopped at 8:30
```

3 Logistic regression: Classification

The variable y we want to classify into classes: Spam/not spam. Fraudulent(not raudulent online transactions. Classifying tumors malign/benign.

In all these cases $y \in \{0, 1\}$. The zero being called the *negative class* and the one being called the *positive class*. The assignemtn of the two classes to positive and negative class is somewhat arbitrary, but the negative is often the absence of something and the positive presence.

We'll start with a classification problem with only two possible values, called *two-class* or *binary* classification problem. Late we'll also look at *multiclass* classification problems.

How do we develop a classification problem. One thing could do is to use linear regression

and then threshold the hypothesis at some value e.g. 0.5. We then have a classifier algorithm. In the exmple above this looks reasonable, but adding an outlier to the right will tilt the regression line and that makes the prediction very bad as shown in the example ??.

Applying linear regression to a classification problem usually isn't a good example. Also, $h(x)$ can output values much larger than 1 or smaller than 0, and that seems ind of weird.

Logistic regression is nicer than that, among other things since $0 \leq h_\theta \leq 1$. The term regression in "logistic regression"is the name the algorithm was given for historical reasons, even though it is really a classification algorithm.

The *hypothesis representation* in logistic regression is:

$$h(x) = g(\theta^T x) = \frac{1}{1 + e^{\theta^T x}}$$

g is called the *sigmoid function* or the *logistic function*, and the latter is what gives logistic regression its name. The two names are synonyms and can be used interchangably (is that a word?.

What we need to do is to fit the parameters to the θ, and we'll get an algorithm for this soon enough.

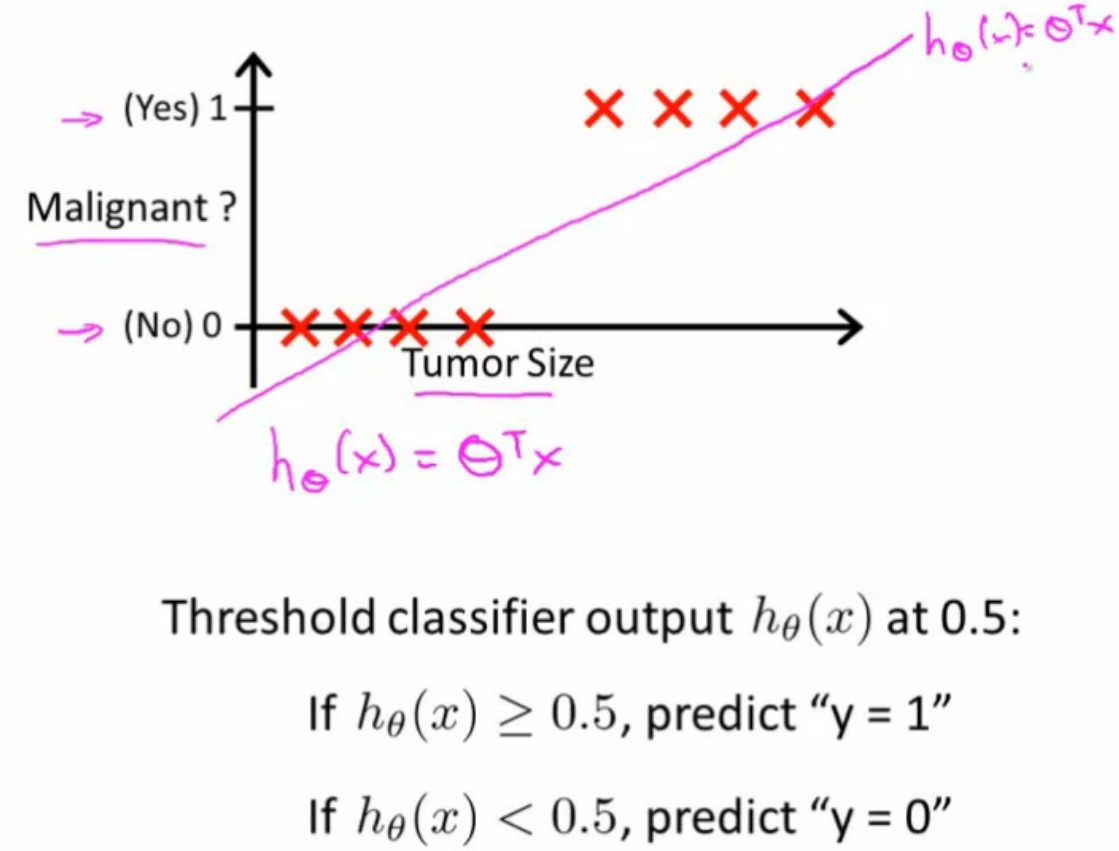


Fig. 3.1 Linear regression classifier

Decision boundary

More Intuition for the hypothesis function for logistic regression (24 secs into the video).

When should we predict one and when should we predict zero? One way is just to select a threshold.

In essence, our choice will flip when $\theta^T \geq 0$.

We can use this fact to better unnerstand how logistic regression makes decisions. Assume $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$. We'll lok at how to fit the parameters later, but let's just assume that we've done so and have $\theta = \{-3, 1, 1\}^T$.

The trick is to find the line that most with the smallest wrong classification (error) divides the classes in the feature plane.

The dividing line is called the *decision boundary*. The decision boundary is the property of the parameters, not the data set. We can use the data to determine

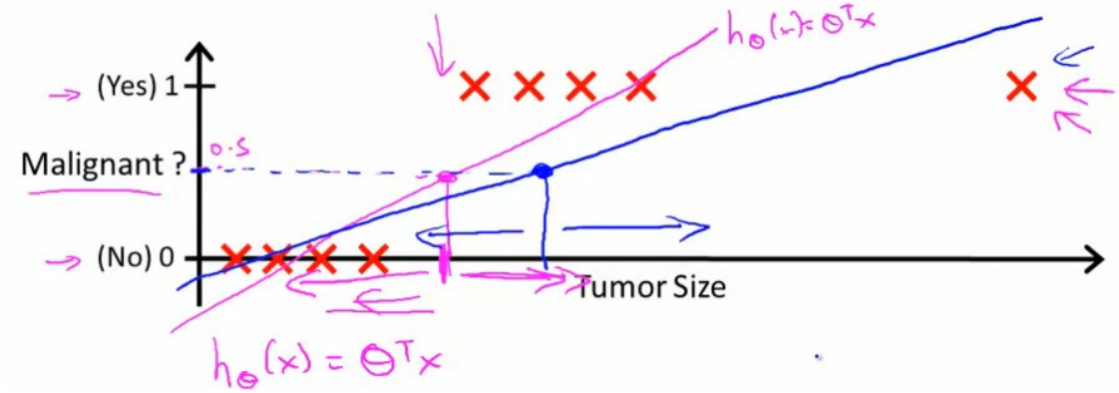


Fig. 3.2 Tilted regression classifier

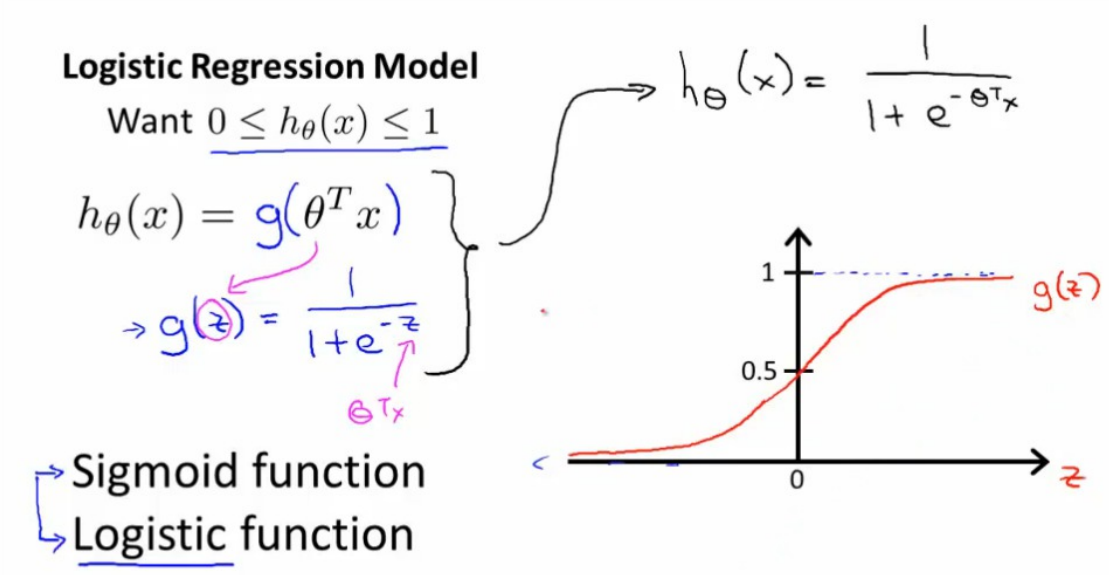


Fig. 3.3 Sigmoid function

the theta, but after that we don't need the training set.

Non-linear boundaries

We can add higher order polynomials to model non-linear features.

The boundary is again property of the hypothesis function.

We can use even higher order polynomials. The boundaries can be then be very

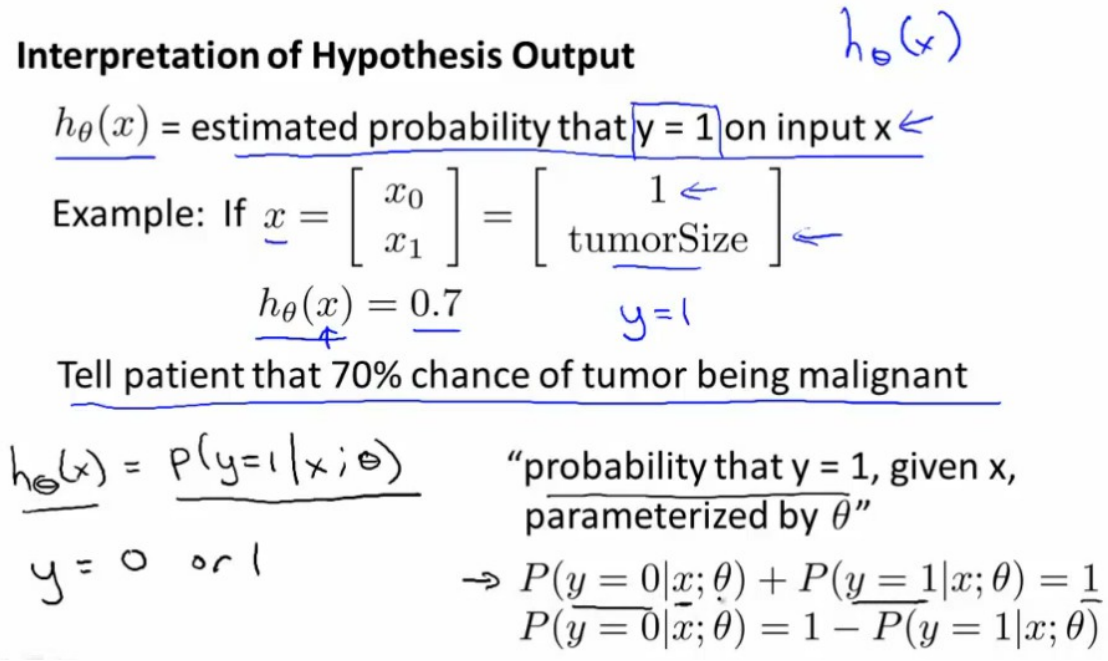


Fig. 3.4 Interpretation of results from logistic regression

complex. Ellipsis, strange shapes. It's a question of

Choosing parametmers to fit the data

The *cost function* defined for the supervised learning problem. In the linear regression case we used the cost function:

$$J(\theta) = \frac{1}{m} \sum_{i_1}^{m} \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

We won't use that now. Instead we'll define it as:

$$J(\theta) = \frac{1}{m} \sum_{i_1}^{m} \text{Cost} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

where

$$\text{Cost} \left(h_\theta(x^{(i)}), y^{(i)} \right) = \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

which can be simplified even more:

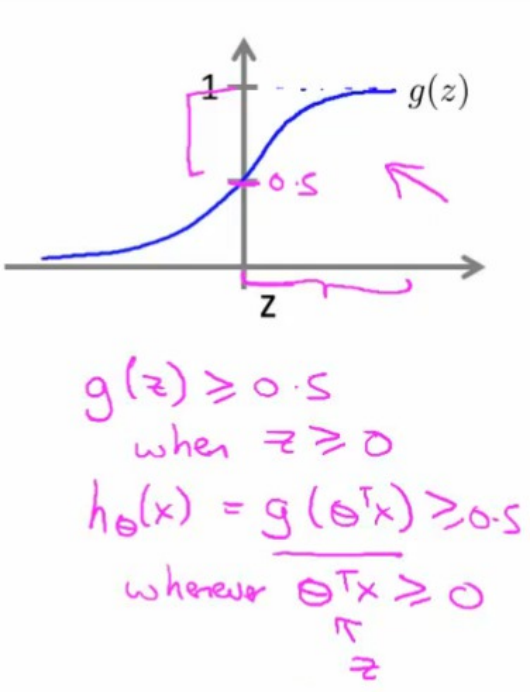


Fig. 3.5 Sigmoid selection criterion

$$\text{Cost}\left(h_\theta(x,y)\right) = \frac{1}{2}\left(h_\theta(x) - y\right)^2$$

It's just the square difference. That worked fine for linear regressjon, but for logistic regression that isn't so good since it's a *non-convex* error function. It has many minima and it's hard to run a gradient descent function on it since it won't find a global minimum

We would like a different, convex, cost function so we can use gradient descent. One such cost function is:

$$\text{Cost}(h_\theta(x), y)) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Convexity analysis is not within the scope of this course, but convex cost functions are nice. Next we'll simplify the notation for the cost function and based on that work out a gradient descent algorithm.

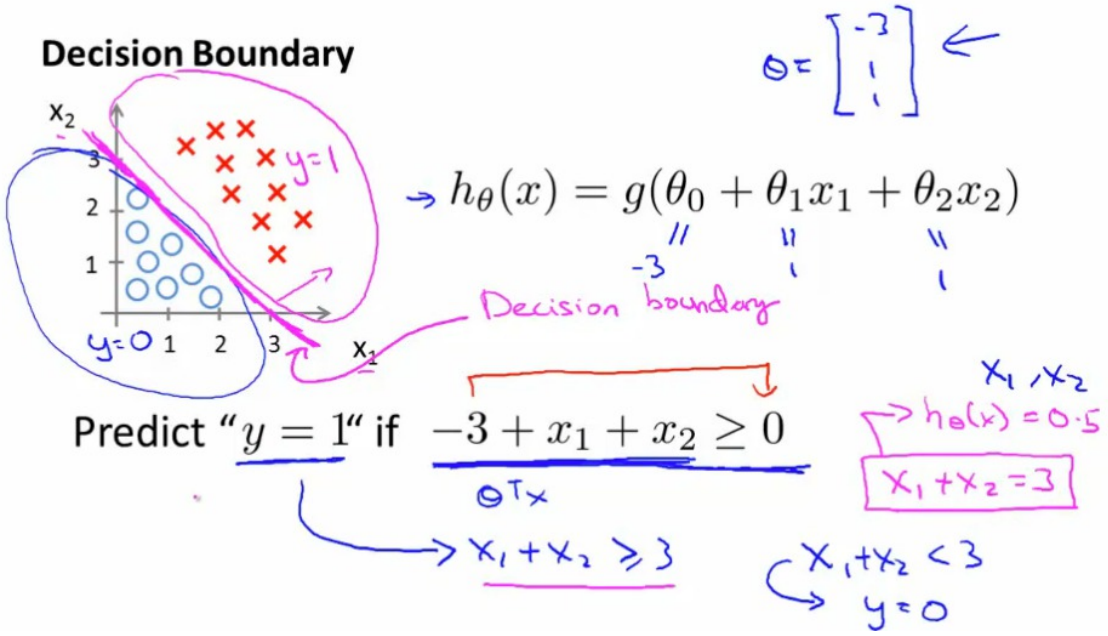


Fig. 3.6 The decision boundary for a linear regression classifier

3.0.1 Simplified cost function

The cost function is

$$\text{Cost}(h_\theta(x), y)) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

y is either zero or one. Because of this we can simplify the cost function. In particular we can compress the two lines as

$$\text{Cost}(h_\theta(x), y)) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$$

You can prove this by plugging in one for y in the equation above, that makes the second part disappear. Putting zero in will let the first term go to zero, and that's the same as the function above.

this gives us the actual cost function:

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \text{Cost}\left(h_\theta(x^{(i)}), y^{(i)}\right) \\ &= \frac{1}{m} \sum_{i=1}^{m} \log(h_\theta(x^{(i)}) + (1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right) \end{aligned}$$

Now why tis particular function? It's derived from *maximum likelyhood estimation* which is nice, it's also *convex*.

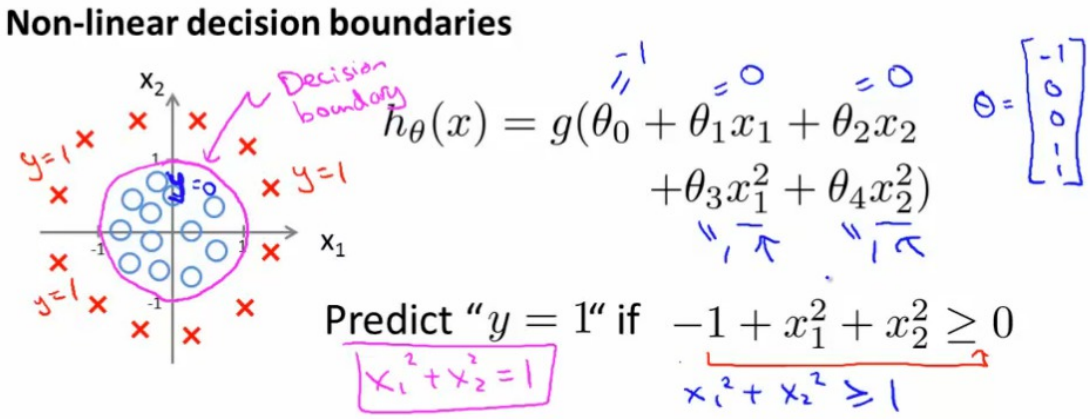


Fig. 3.7 A circular decision boundary

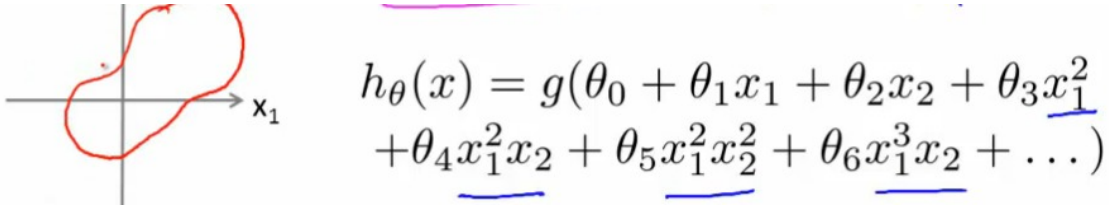


Fig. 3.8 An irregular decision boundary

Given this cost function, what we'll do is:

$$\min_\theta J(\theta)$$

now all we need to do is how to find the thetas. We will use gradient descent. The standard template is to use a step within the gradient descent algorithm:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

In our case this means that:

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) x_j^{(i)} \right)$$

(simultaneously update for all θ_j. This looks identical to lienar regression!

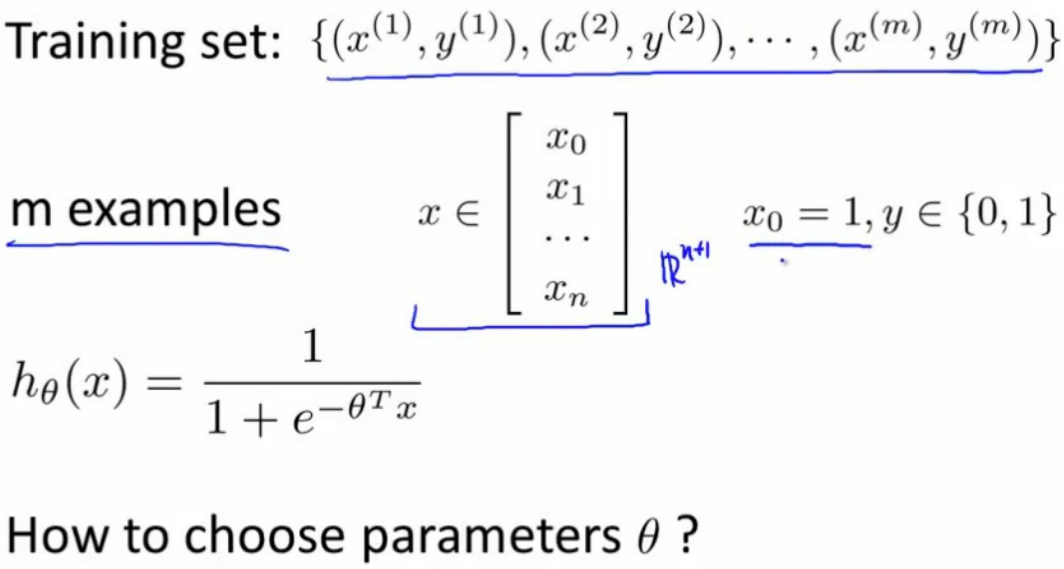


Fig. 3.9 Supervised logistic regression

We can use the same techniques for monitoring to make sure that this regression is progressing nicely. Slow descent of the error J is what we are looking for.

we should try to use vectorized implementations if we can.

feature scaling can help both linear regresson and logistic regression to run faster.

Logistic regression is very powerful and perhaps the most used classification algorithm in the world and now I know how to work with it myself :-)

3.0.2 Advanced optimization

With these techniques we'll get logistic regression to have better performance: Better running times, more features. We have a cost function, but we need to have code that can produce partial derivatives

Strictly speaking we don't need the actual J function, but we'll consider that case anyway since it is very useful for monitoring progress and convergence.

When we've got code for J and the partials, we can use many algorithms to compute a minimum:

- *Gradient descent*
- *Conjugate gradient*
- *BFGS*

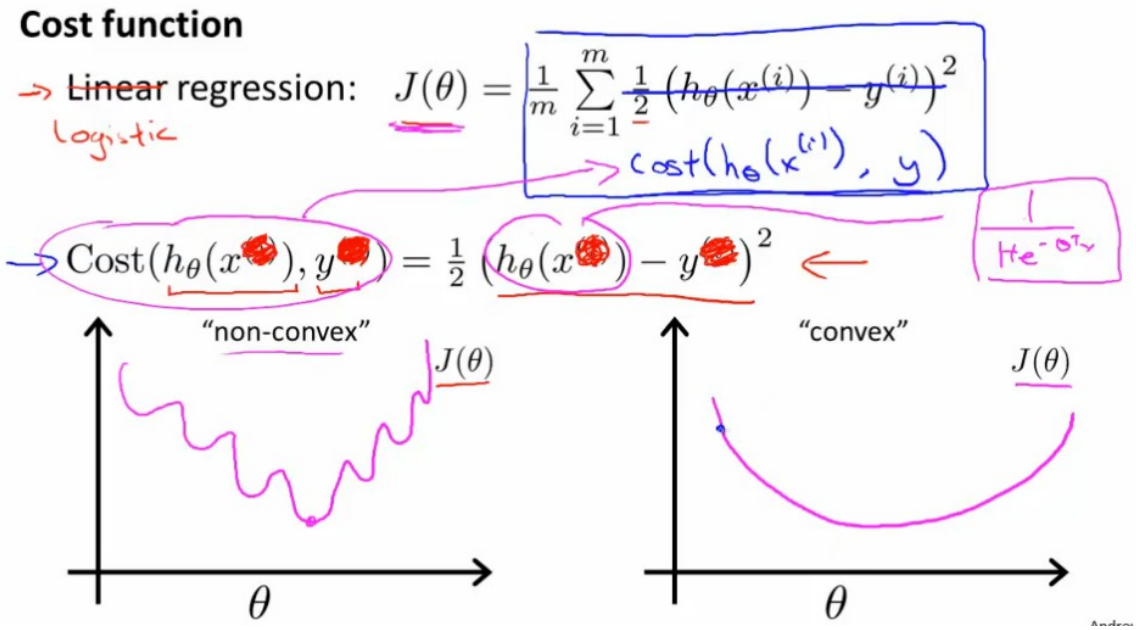


Fig. 3.10 Convex v.s. non-convex cost functions

- *L-BFGS*

The details of these functions are beyond the scope of this voruse.

Don't need to manually pick α.

They have a clever inner loop called a *line search algorithm* that picks an efficient α for us. Often faster than gradient descent, but more complex. In essence, they put a little regulator on the optimization algorithm to keep it within the sweet spot. Can choose a different learning rate for each iteration.

Ng used these algorithms for a long while (over a decade), but only recently did he figure out the details of what they do.

These algorithms are so complex that you probably shouldn't write these algorithms yourself. Use a library instead. Fortunately octave has a very good library implementing some of these algorithms, so just use those libraries and you'll get a decent result.

An example

fminunc is the octave function for "function minimization unconstrained". initial-theta is the initial guess, and options is a set of options. the atsign is a pointer in octave syntax.

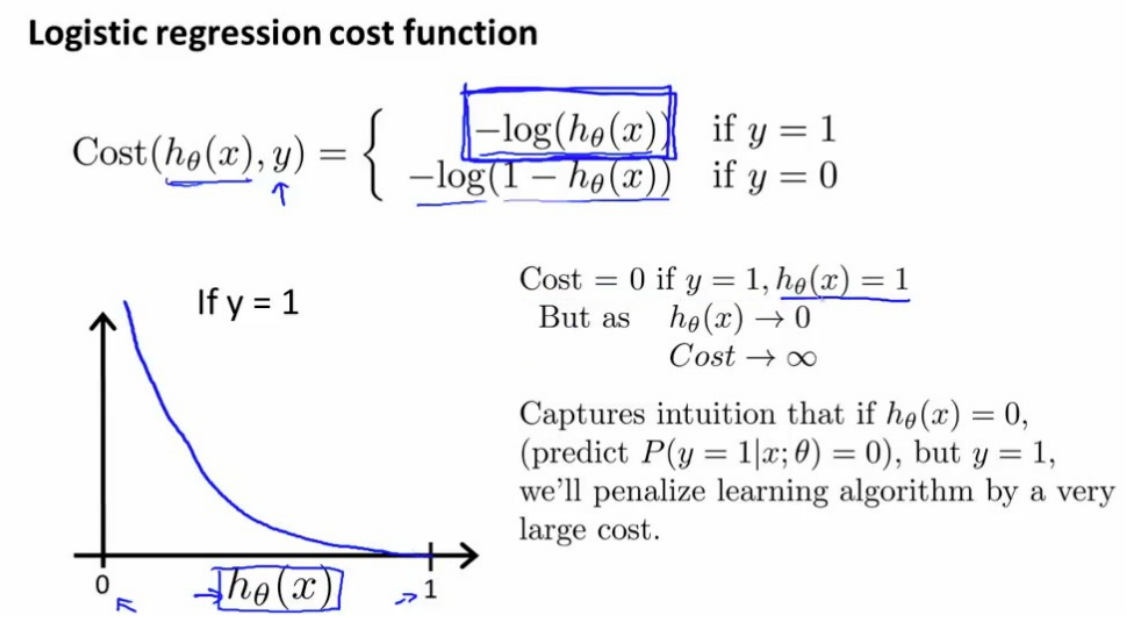


Fig. 3.11 Half the cost function for logistic regression

In logistic regression we use this like this that gives us the cost function for logistic regression.

Using advanced algorithms is a bit more opaque, but for big problems, they are bettrer for big problems.

3.0.3 Logistic regression on multiclass clasification one-vs-all

This is about the *one v.s. all algorithm*. Assume we have a mail classification problem, the classes are work,friends, family hobby. In a medical system we may diagnose patients not ill, cold flu, or weather sunny, cloudy, rain, snow.

One-vs-al classification works like this: Assume three classes. Turn it into three binary classification problems. class one v.s. the rest, class two v.s. the rest and class three v.s. the rest.

for each classifiser we fit a classifier:

$$h_\theta^{(i)}(x) = P(y = i|x; \theta)\, (i \in \{1, 2, 3\}$$

We estimate "what is the probability that x is in class i parameterized by theta"

To make a prediction, we run all the classifiers and select the one with the highest probability:

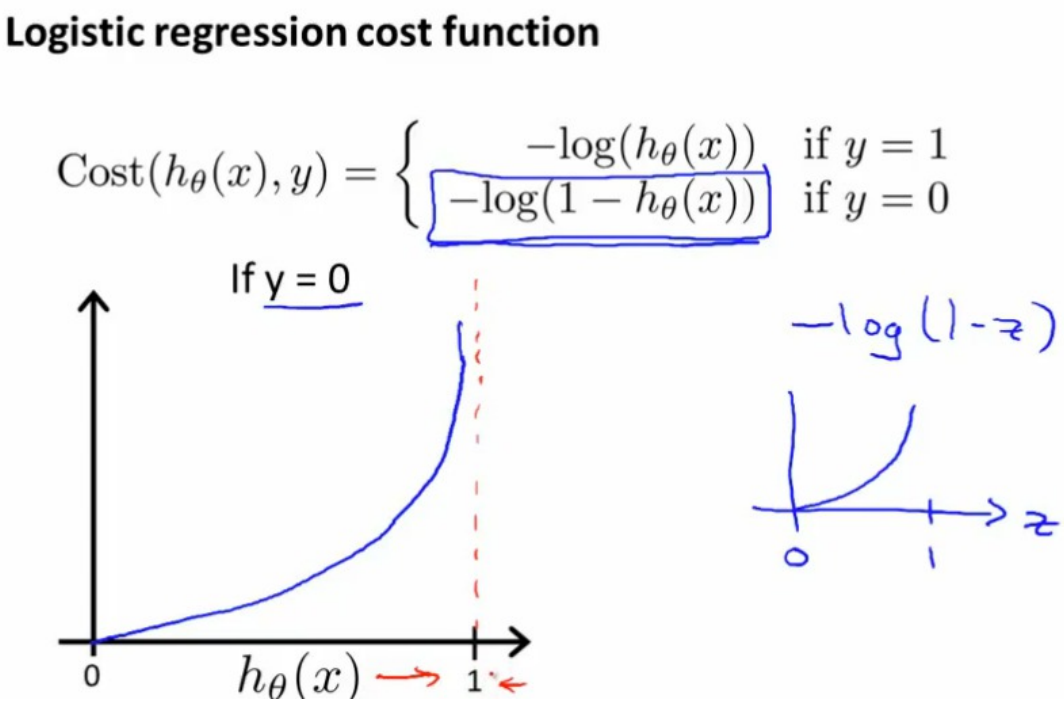


Fig. 3.12 The other half of the cost function for logistic regression

$$\max_i h_\theta^{(i)}(x)$$

"Pick the classifier with the most enthusiasm :-)"

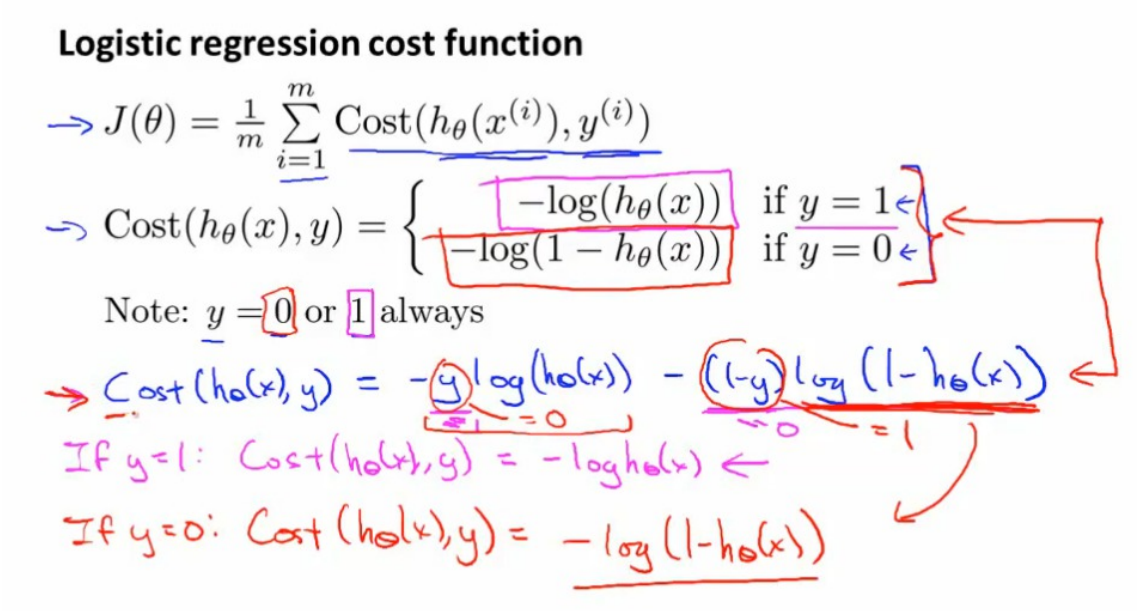


Fig. 3.13 A simplified cost function for logistic regression

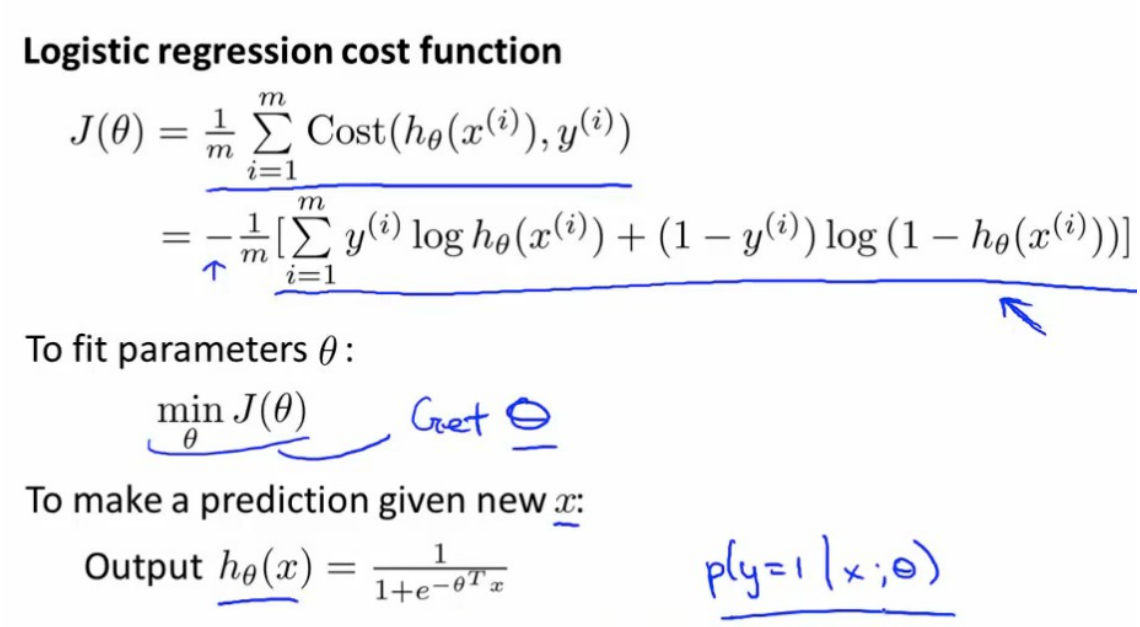


Fig. 3.14 A cust function for logistic regression that is actually useful for something :-)

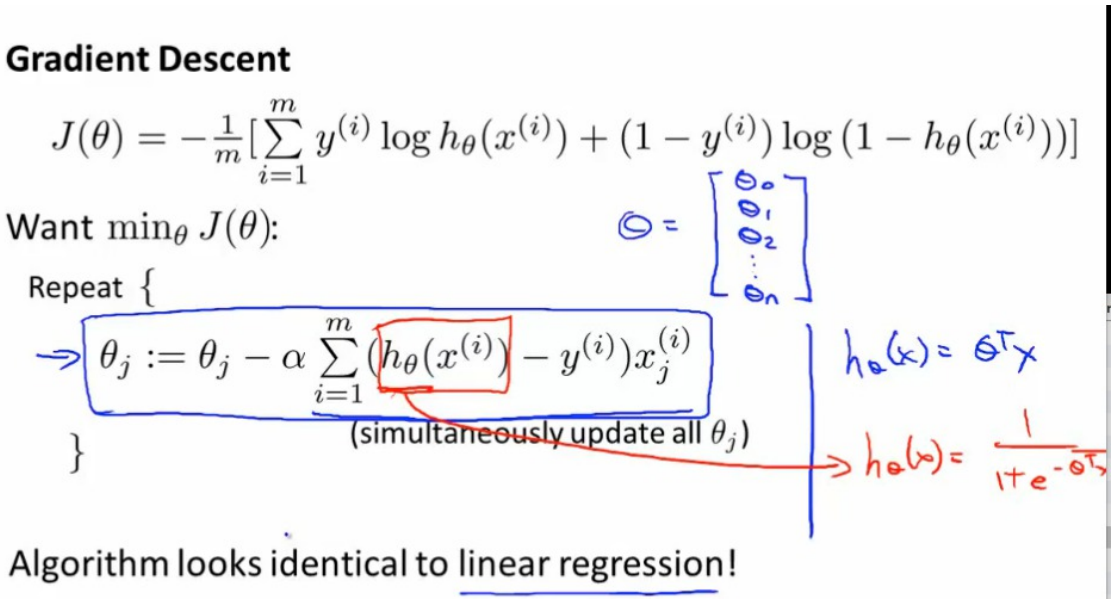


Fig. 3.15 Gradient descent with logistic regression

One iteration of gradient descent simultaneously performs these updates:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}$$
$$\vdots$$
$$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_n^{(i)}$$

We would like a vectorized implementation of the form $\theta := \theta - \alpha\delta$ (for some vector $\delta \in \mathbb{R}^{n+1}$). What should the vectorized implementation be?

- (•) $\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^{m} [(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}]$
- ( ) $\theta := \theta - \alpha \frac{1}{m} (\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})) \cdot x^{(i)}$
- ( ) $\theta := \theta - \alpha \frac{1}{m} x^{(i)} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$
- ( ) All of the above are correct implementations.

Correct

Continue

Fig. 3.16 A vectorized implementation of logistic regression

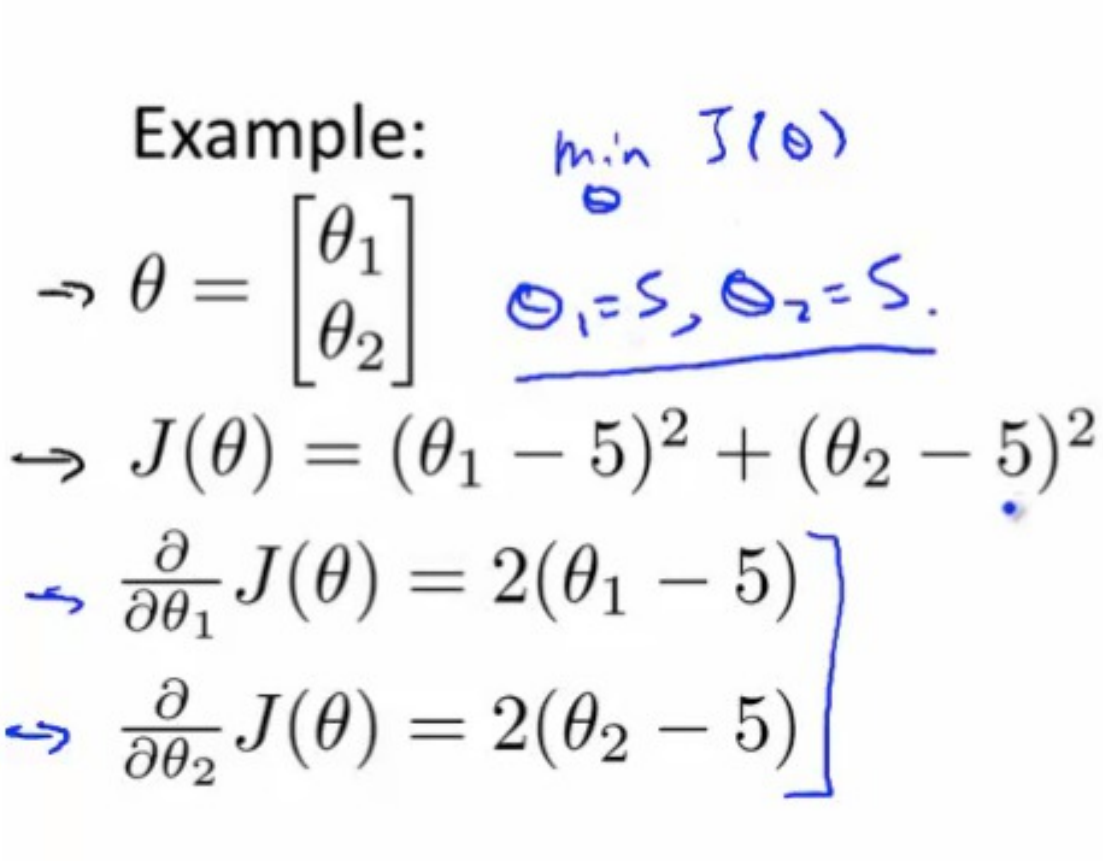


Fig. 3.17 An example function for illustrating how the advanced optimization algorithms can be used

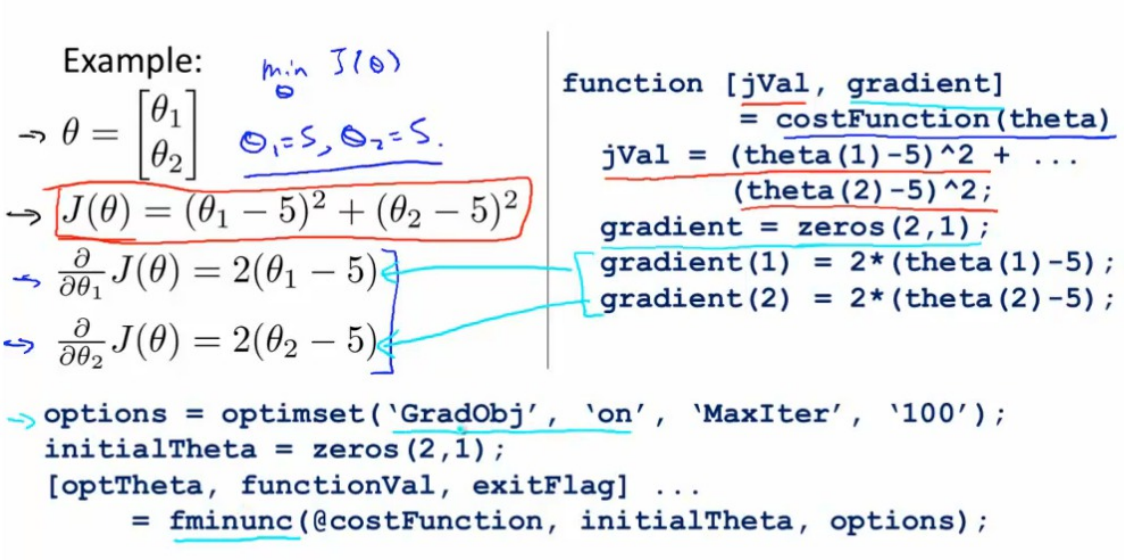


Fig. 3.18 example2

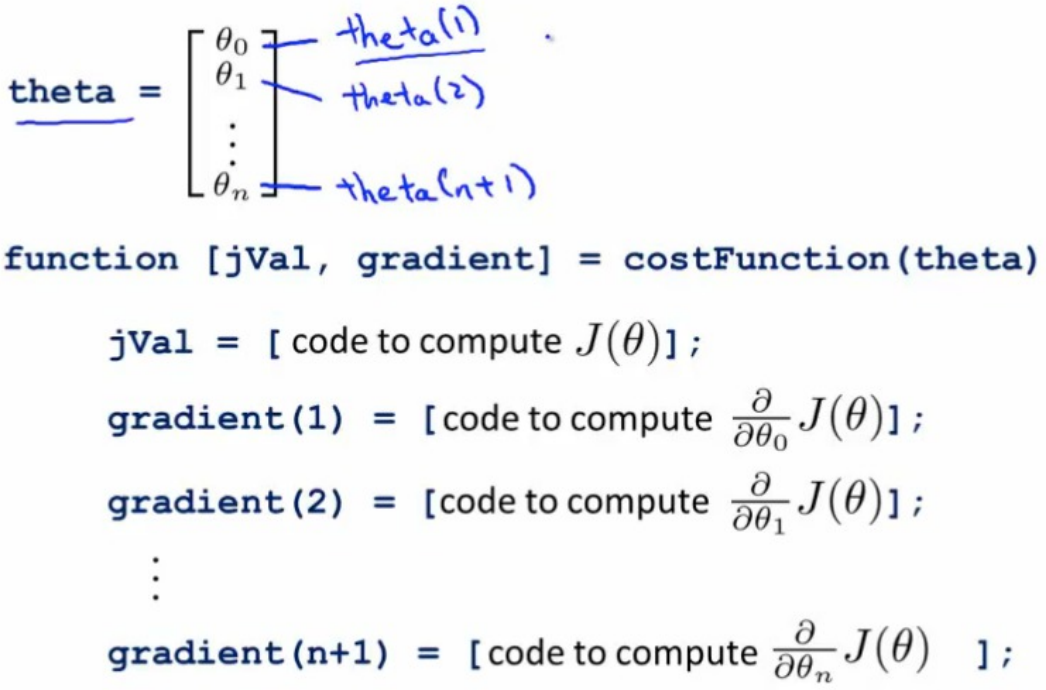


Fig. 3.19 Using the advanced optimization functions

Suppose you want to use an advanced optimization algorithm to minin the cost function for logistic regression with parameters θ_0 and θ_1. You write the following code:

```
function [jVal, gradient] = costFunction(theta)
  jVal = % code to compute J(θ)
  gradient(1) = CODE#1  % derivative for θ0
  gradient(2) = CODE#2  % derivative for θ1
```

What should CODE#1 and CODE#2 above compute?

- ○ CODE#1 and CODE#2 should compute J(θ).
- ○ CODE#1 should be theta(1) and CODE#2 should be theta(2).
- ◉ CODE#1 should compute $\frac{1}{m}\sum_{i=1}^{m}[(h_\theta(x^{(i)})-y^{(i)})\cdot x_0^{(i)}]\ (=\frac{\partial}{\partial\theta_0}J(\theta))$, CODE#2 should compute $\frac{1}{m}\sum_{i=1}^{m}[(h_\theta(x^{(i)})-y^{(i)})\cdot x_1^{(i)}]\ (=\frac{\partial}{\partial\theta_1}J(\theta))$.
- ○ None of the above.

Fig. 3.20 An implementation of logistic regression

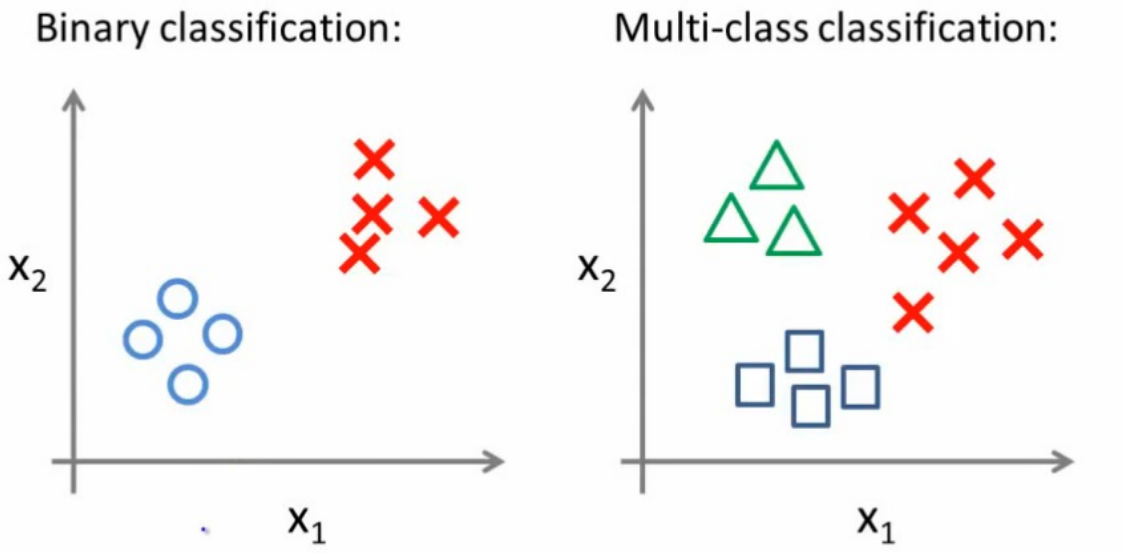


Fig. 3.21 A datset containing entries of many classes

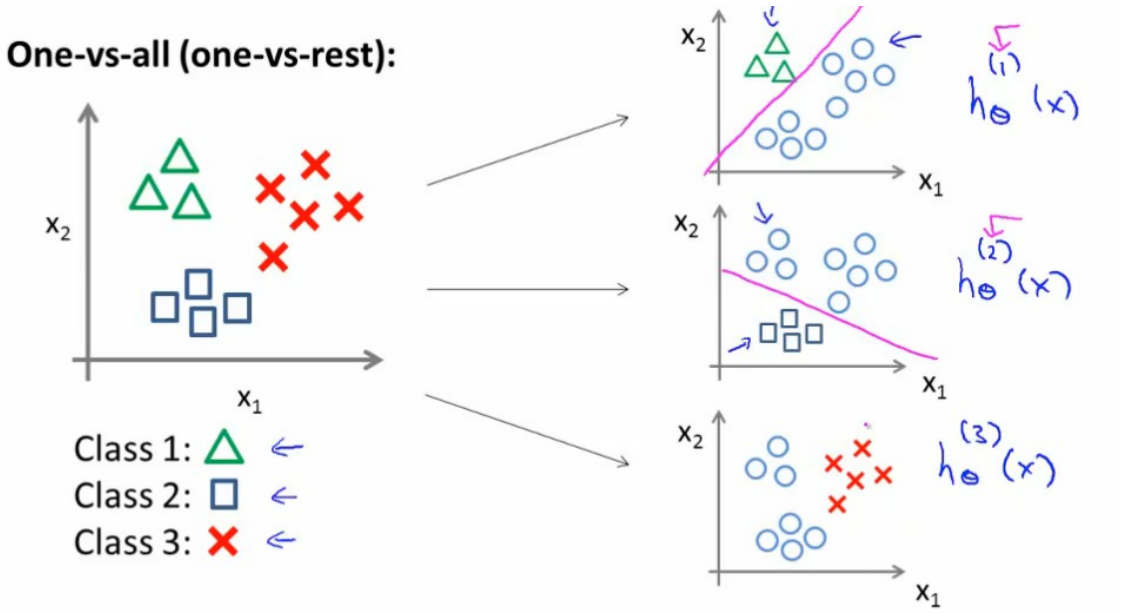


Fig. 3.22 Using the "one v.s. all" classification method

4 Regularization

4.1 The problem of overfitting

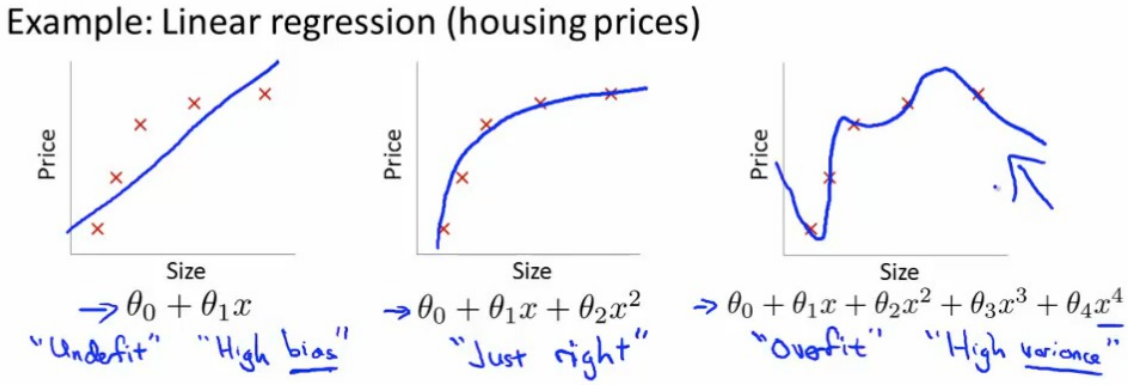


Fig. 4.1 Ovefitting – The nemesis of curve fitters

Linear regression and logistic regression can run into *overfitting* which. One way of ameliorating the problem is *regularization*. We'll now look at overfitting and later the fix.

If we get a poorer fit from linear regression, we get a model that is *underfit* or has *high bias*. The term bias "bias" is historical, an indicates that the model has a strong preconception that the data will be linear, and despite the data it will still fit linear data. We can then add more factors and for instance try using higher order polynomials to match the data. That will reduce the error to the point that will not have any error. However the curve will be wiggely, and that curve will be *overfitted*, or have *high variance*. The term "high variance" is another technical one. The space of possible hypothesis is just too large, to variant, and we don't

have enough data to give a good hypothesis. In the middle there is the *just right* case :-) The problem with overfitting is that it gives bad predictors, even if it gives good fit for the training set. It fails to generalize.

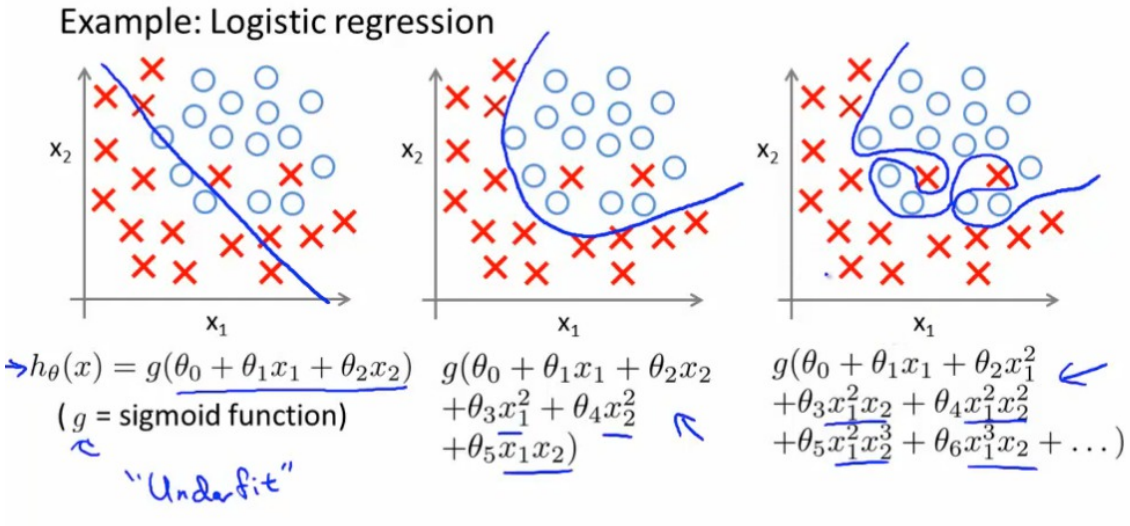


Fig. 4.2 Overfitting a logistic regression classifier

The problem applies both to linear regression and to logistic regression.

4.1.1 Adressing overfitting

Overfitting can be recognized by tools we'll learn more about later. However, there are two mi options for adressing the problem.

1. reduce number of features
   a) Manually select which features to keep.
   b) Model selction algorithm (later in the course)
2. Regularization
   a) Keep all the features, but reduce magnitudes/alues of parameters θ_j
   b) Works well when we have a lot of features, each of which contributes a little bit to predicting y.

4.2 Cost function

Implementing regularization is a good way to learn regularization.

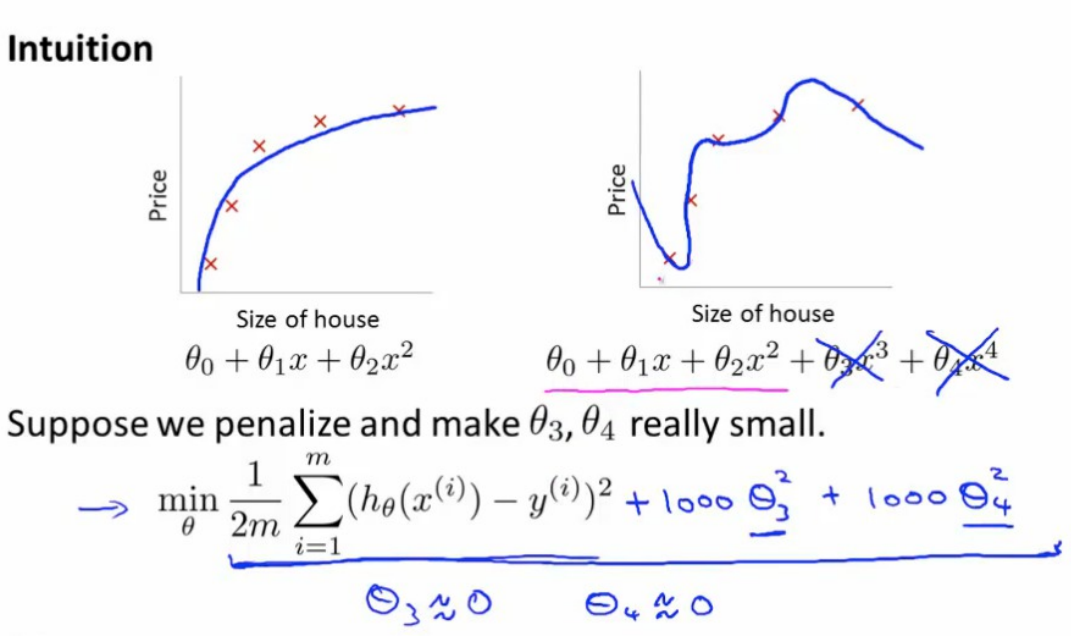


Fig. 4.3 How adding a penalty for higher order terms reduces overfitting

Add penalty for higher order factors, meaning that they are only included in the model if they are really worth it. The idea is to have smal values for parameters in θ. This gives us smoother and simpler hypothesis which are less prone to overfitting.

If we have a bunch of features it is difficult to pick the ones that are relevant. So what we'll do is to modify the cost function to penalize all parameters by adding a regularization term:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda \sum_{i=1}^{n} \theta_j^2\right]$$

It's convention to not penalize θ_0. λ is the *regularization parameter* it regulates the tradeoff between the two objectives of fitting the data well and keeping the parameters smal.

A too high value of λ is equivalent to fitting a straight horizontal line. It will result i severe *underfitting* due to a too strong bias that the function is straight horizontal line.

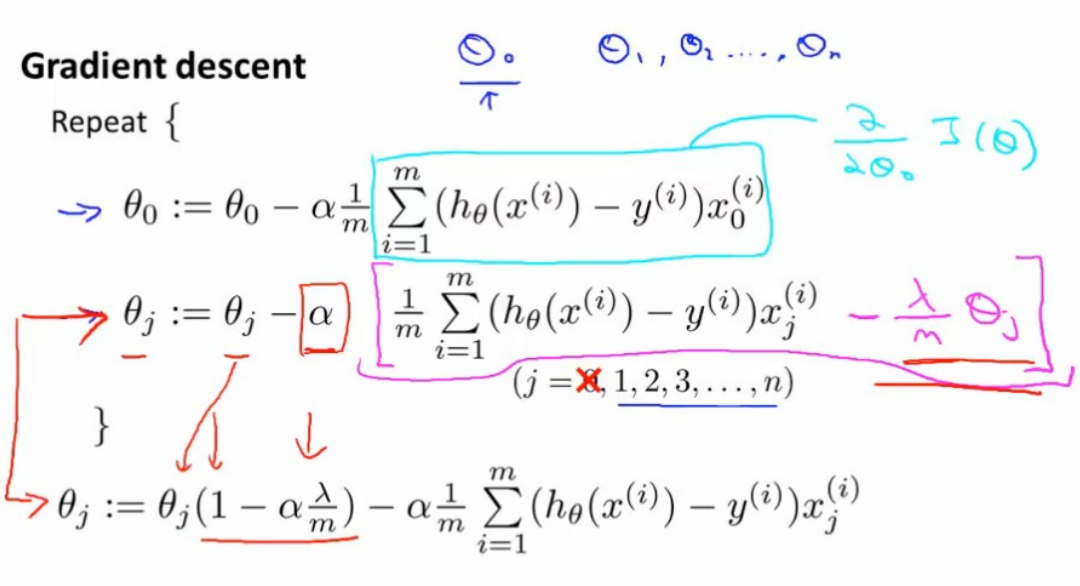


Fig. 4.4 Regularized gradient descent: Gradient descent with penalty for higher order terms.

4.3 Regularized Linear Regression

We found this error function.

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \right] + \lambda \sum_{i=1}^{n} \theta_j^2$$

When using this in gradient descent, we need a new error function and gradient. The gradient is very similar to the original, but it has an extra step:

$$\theta_j := \theta_j - \alpha \left[\sum_{i=1}^{m} \left(h_\theta(x^{(i)}) x_j^{(i)} \right) - \frac{\lambda}{m} \theta_j \right]$$

Weirdly it's must be true that $1 - \alpha \frac{\lambda}{m} \leq 1$
By grouping the θ_j factors together, we get:

$$\theta_j := \theta_j (1 - \alpha \frac{\lambda}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) x_j^{(i)} \right)$$

We also have the normal equation that can be used to find a minimization.

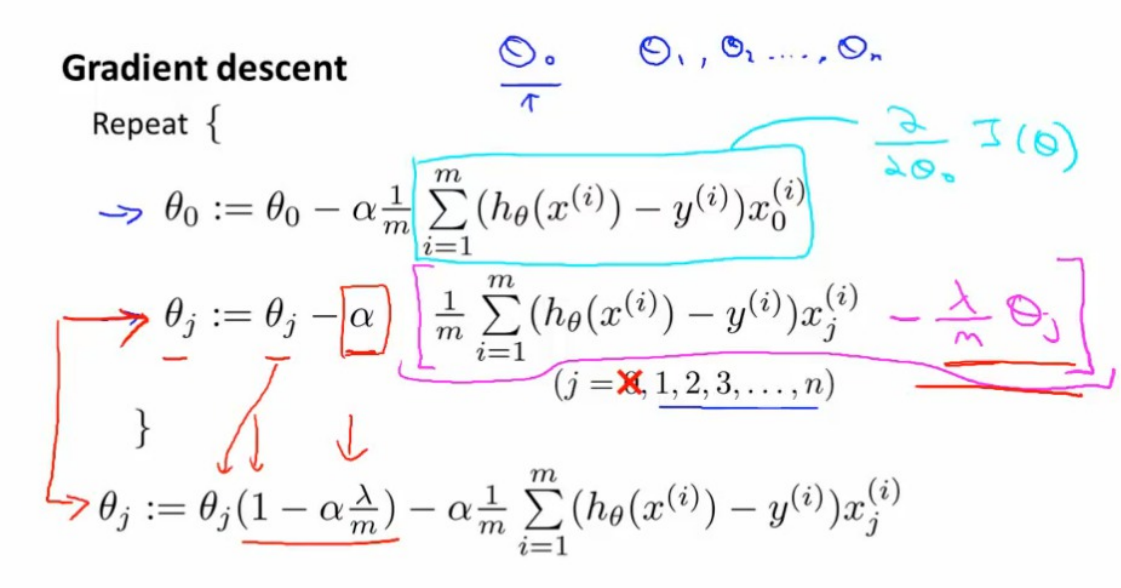


Fig. 4.5 Gradient descent taking regularization into account

Non invertibility

The regular inverse will fail if you have a noninvertible matrix, so the pinv method is necessary.

Fortuntately, regularization takes care of that problem too, so we won't have a degenerate (singular) matrix.

4.4 Regularized Logistic Regression

We have both advanced optimization method and gradient descent, and we'll now learn how to used these for regularized logistic regression.

Logistic regression is prone to overfitting. All we have to do is to change the cost function.

How do we implement this? We treat θ_0 separately, but then do something very similar to what we did for linear regression.

The update actually is cosmetically identical to the one we use in linear regression, but it is actually different since the hypothesis uses the logistic function, not just a linear function of x.

The term in the square bracket is the new partial derivative.

How to implement this using the advanced optimization. First we must define a cost function

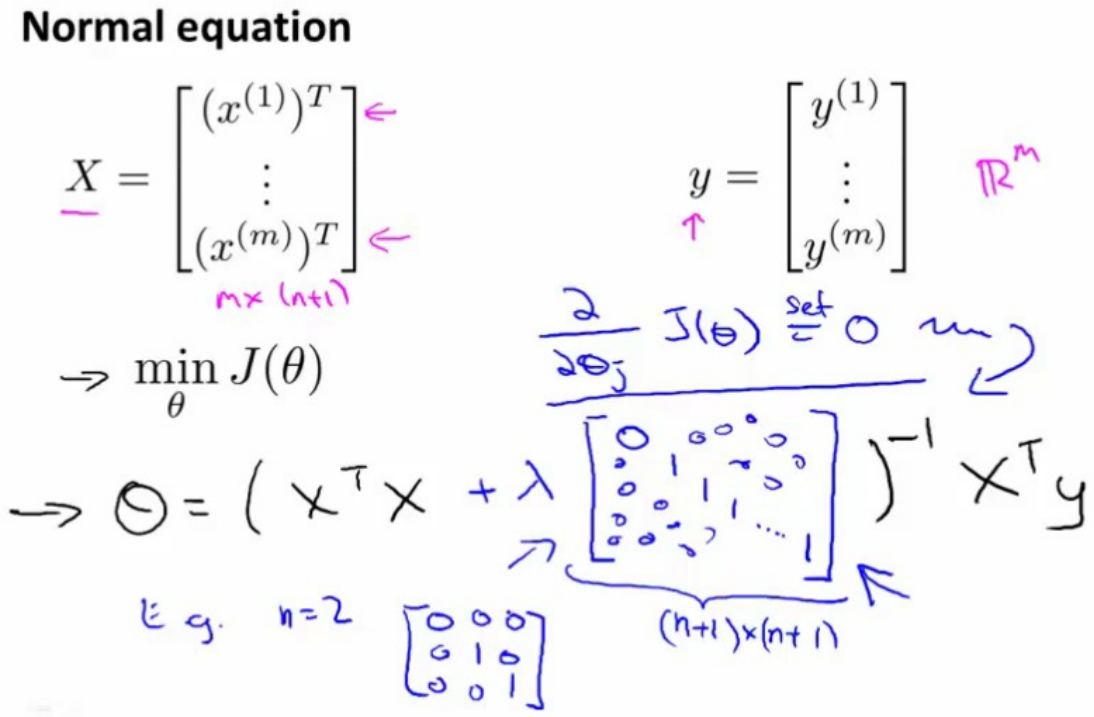


Fig. 4.6 The normal equation for gradient descent with regularization

If you understand the stuff presented so far, you probably know as much machine learning as a lot of engineer working in the silicon valley. Still more to learn though :-) Next up is highly nonlinear classifiers that we can use.

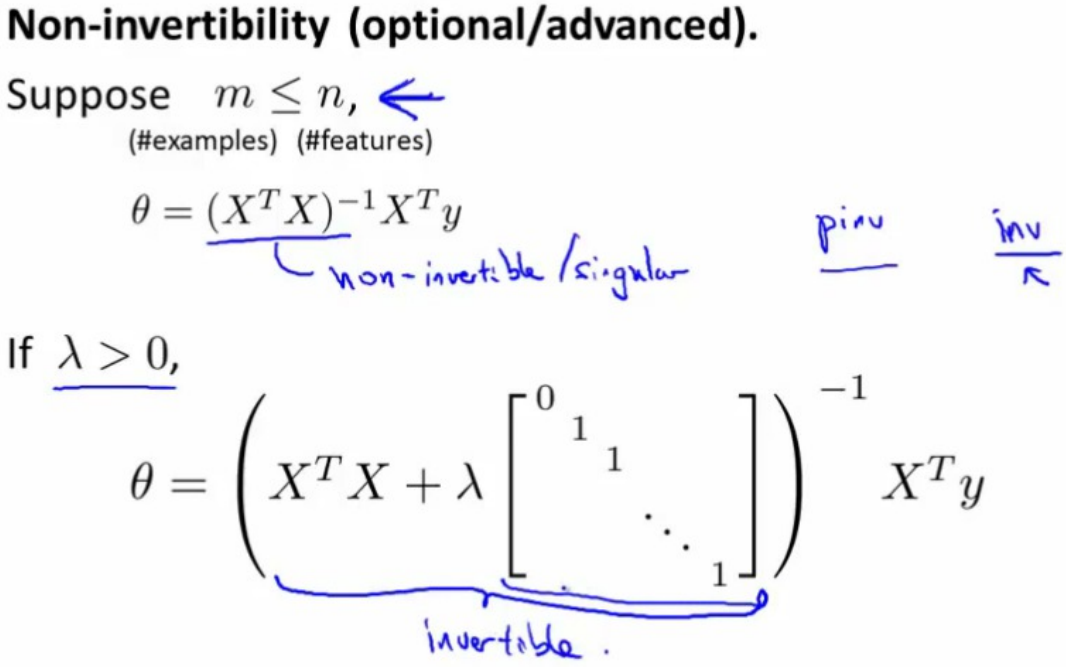


Fig. 4.7 A noninvertible/singular covariance matrix in the normal equation

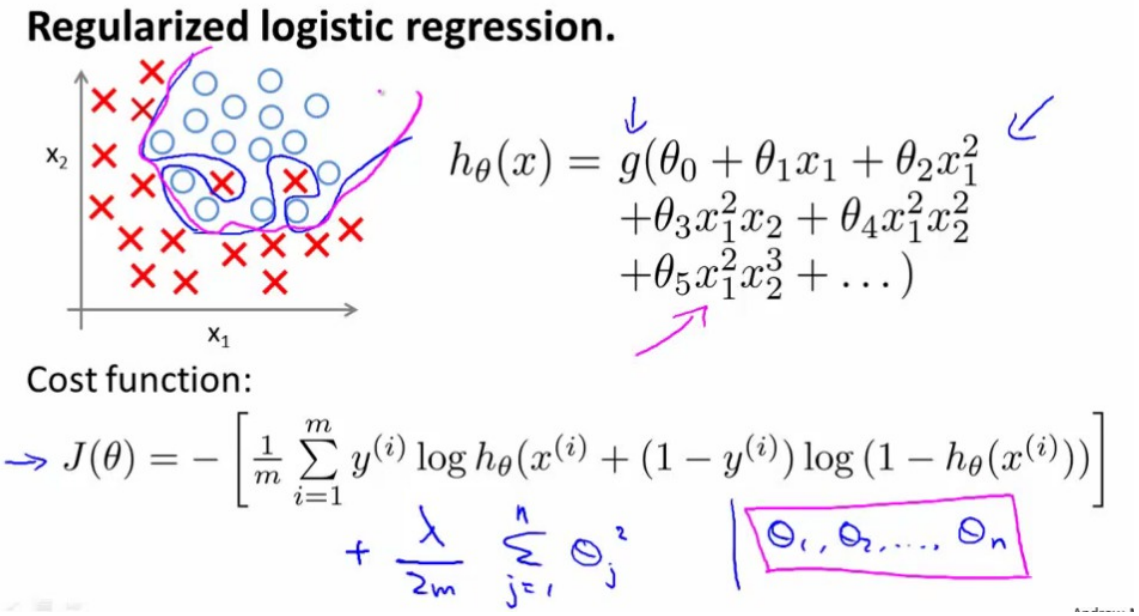


Fig. 4.8 Logistic regression with normalization

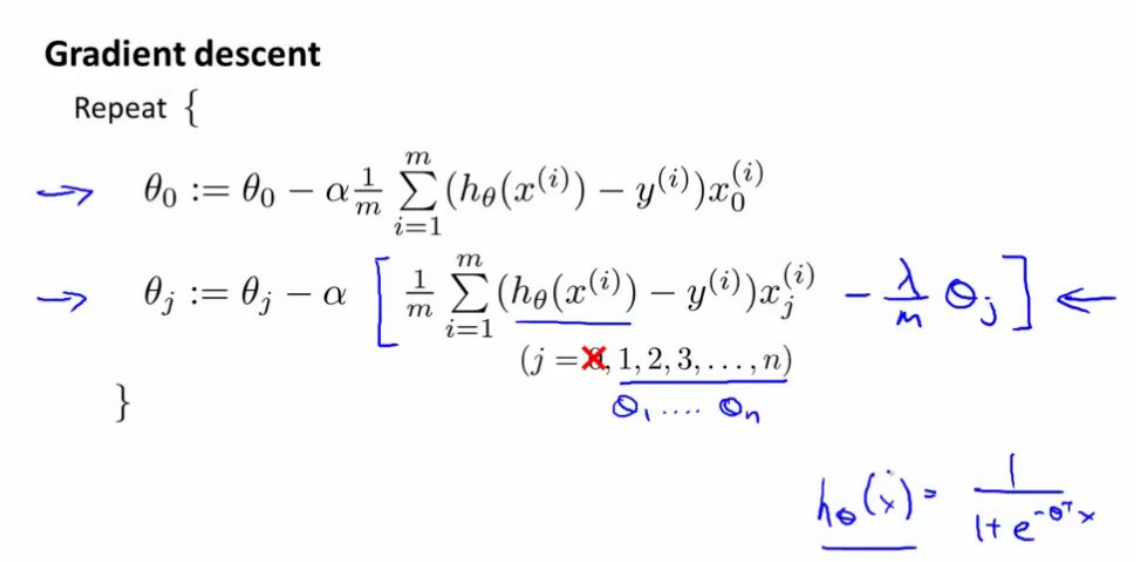


Fig. 4.9 Gradient descent for logistic regression with regularization

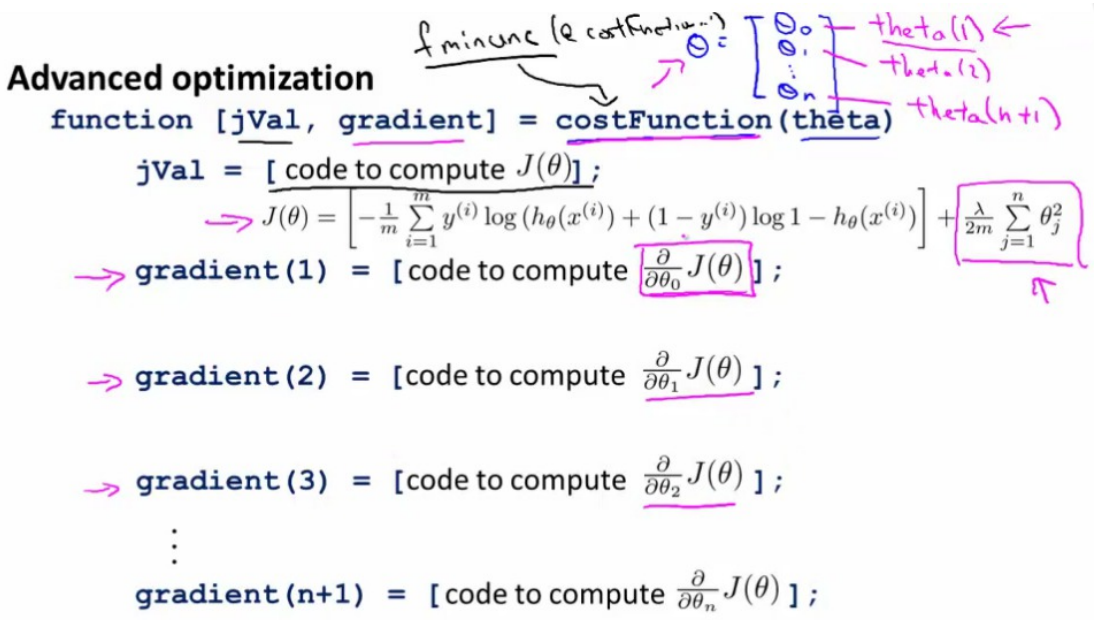


Fig. 4.10 A cost function to be used with the advanced optimization algorithms

5 Neural networks representation

Neural networks are old, were out of favor for a few years, but today it's the state of the art.

Why? If the decision boundary is highly nonlinear the logistic regression will be very difficult. For a quadratic features, the number of features grows quadratically, and that's a lot of features. It's both computationally expensive and it's prone to overfitting. Same thing for even higher orders. High orders of parameters is a pretty normal situation in machine learning.

Computers don't see things s we do, they just se a whole lot of numbers. Building a car detector for images is hard. Training the classifier for cars needs a lot of parameters :-)

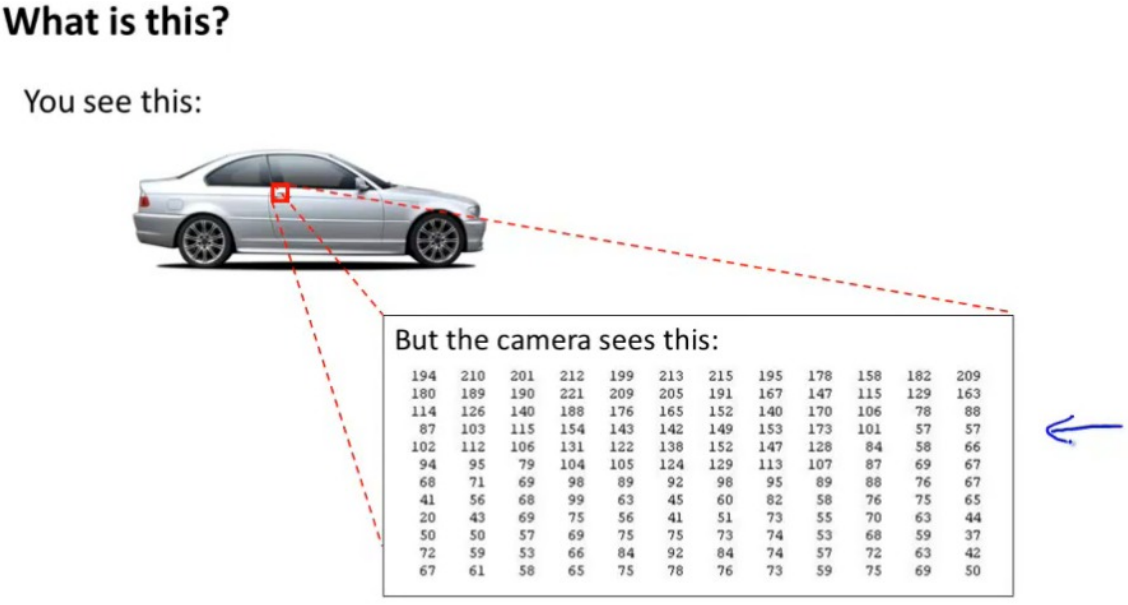


Fig. 5.1 Why computer vision is hard, the computer only sees numbers and has to create a concept of whatever it sees based on these numbers.

Quadratic regression over a a pixel can easily have several tens of million of features that are applicable for logistic regression.

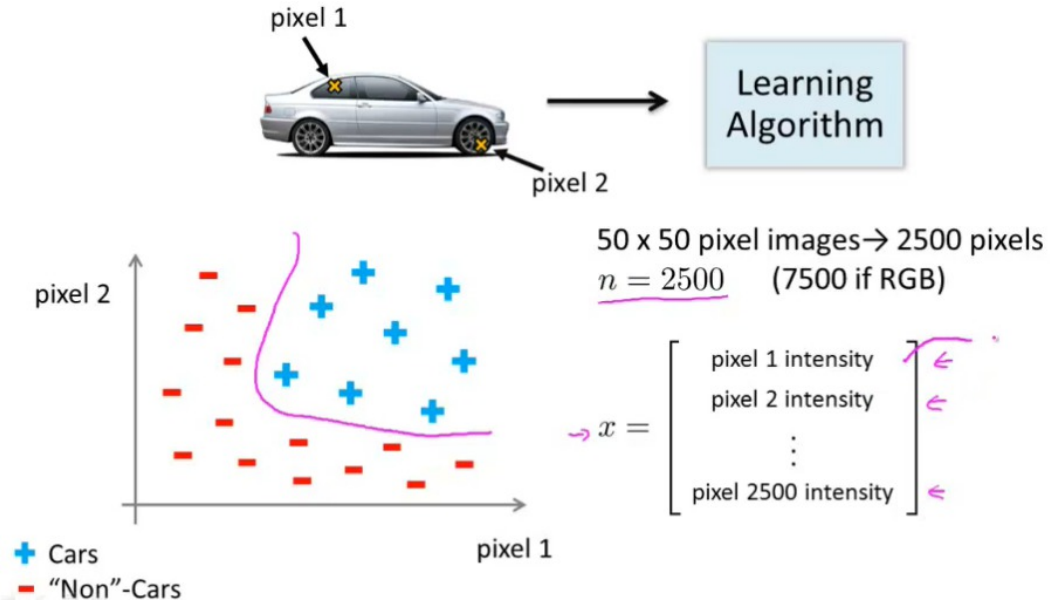


Fig. 5.2 Representing cars

5.1 Neuron and the brain

They are biologically motivated, just to get some idea of what they can do. Origins in algorithsm to mimic the brain. Widely used in 80s and 90s, popularities diminished in late 90s. Recent resurgence: Stae of the art tehniuqe for many applications.

The does a lot of amazing things. There is a hypothesis that the brain uses a single learning algorithm, the *one learning algorithm hypothesis*. Example, if we route visual nerves to the auditory cortex, the animal will learn how to see again. The brain rewires dynamically. *neuro rewiring experiments* indicates that different parts of the brain can process different types of information, so perhaps it is reasonable to assume that all the parts of the brain uses the same learning algorithm, not a raft of task-specific algorithm.(metin and frost 1989).

5.1.1 Model representation

Brains are full of neurons. Neurons have a body and number of input wires *dendrites*. The output-wire is called *axon*. At a simplistic level it gets a bun of input from its input and send an output to the output axon.

The neurons sends signals called *spikes* to dendrites of other neurons. This is the process by which all neural computation happens. This is also the way that IO

Fig. 5.3 The brain is capable of mapping a wide range of sensor inputs into stuff it can make sense of

is done to muscles.

In an artificial neural network we model a neuron as a logistic unit:

This is a very (perhaps vastly) simplified version of the network.

Sometimes an extra input $x_0 = 1$ is drawn, and sometimes not. It's called the *bias unit*. The *activation function* is here the logistic function. In a neural network calls the θ parameters *weights*.

The diagram above represents a single neuron. A network is a group of neurons thrown together.

The first layer is called the *input layer*, The final layer is called the *output layer*. The layer(s)in between is called *hidden layer*(s) (not observable in the training examples).

In the network we call the unit $a_i^{(j)}$ the *activation* of unit i in layer j. The matrix $\Theta^{(j)}$ are all the weights controlling the function mapping from layer j to layer $j+1$.

5.1.2 A vectorized representation of neural networks

New notation : $a_1^{(2)} = g(z_1^{(2)})$ refer to items in layer 2 (the hidden layer) of the network.

We can use this to vectorize the operation of the calculation of neural network

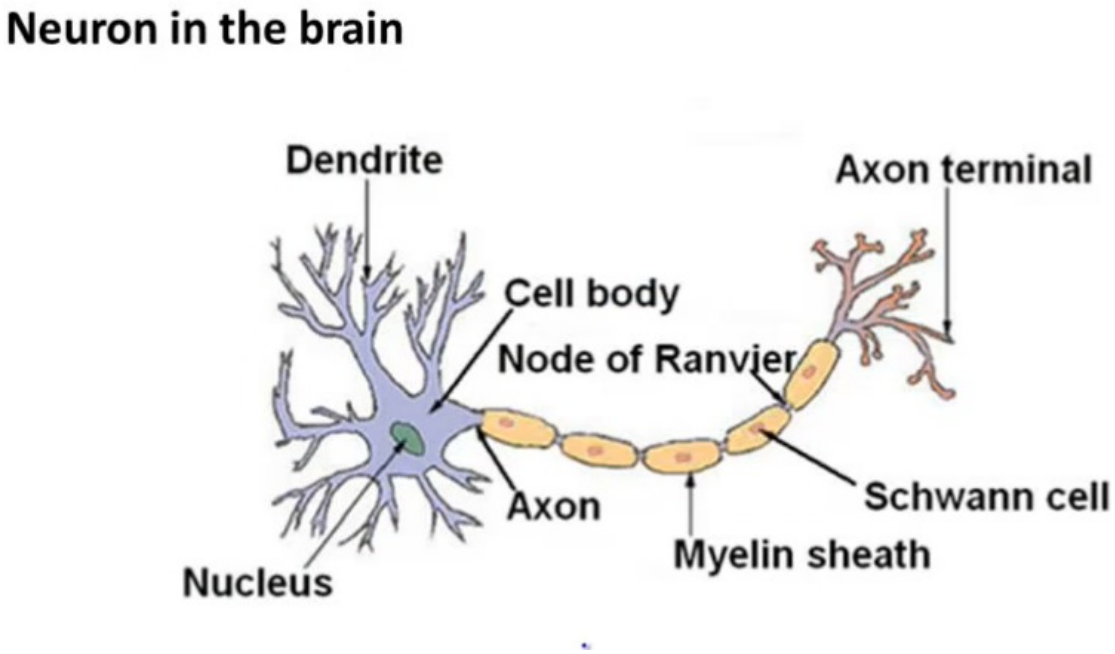


Fig. 5.4 The structure of a single neuron in the brain

values. The activation layer $a^{(1)}$ is defined to be the x value (from linear algebra convenson). We add a $a_0^{(1)} = 1$ as an extra bias input. Similarly we add a $a_0^{(2)} = 1$ to the hidden layer, being a bias unit there. To determine the output value we calculate $z^{(3)}$.

Rmz: *Why no thresholding?*

5.1.3 Intution about what NNs are doing

A covered up neural network (fig ??) is essentially a logistic regression facility. The features that are fed into the network are the hidden units instead of the (externally input) features.

The cool thing about this is that the mapping from layer one to layer two has learned how to recognize features on its own from the input. And it is these learned features that are then applied to the final logistic output function. This is a very flexible mapping function so it can do a lot more than e.g. a polynomial used in a logistic regression input.

the *architecture of a neural network* describes how the nodes are connected.

5.1.4 Non-linar classification xor/xnor

Landmark for the sigmoid function: When the x value is 4.6 the y value is 0.99.

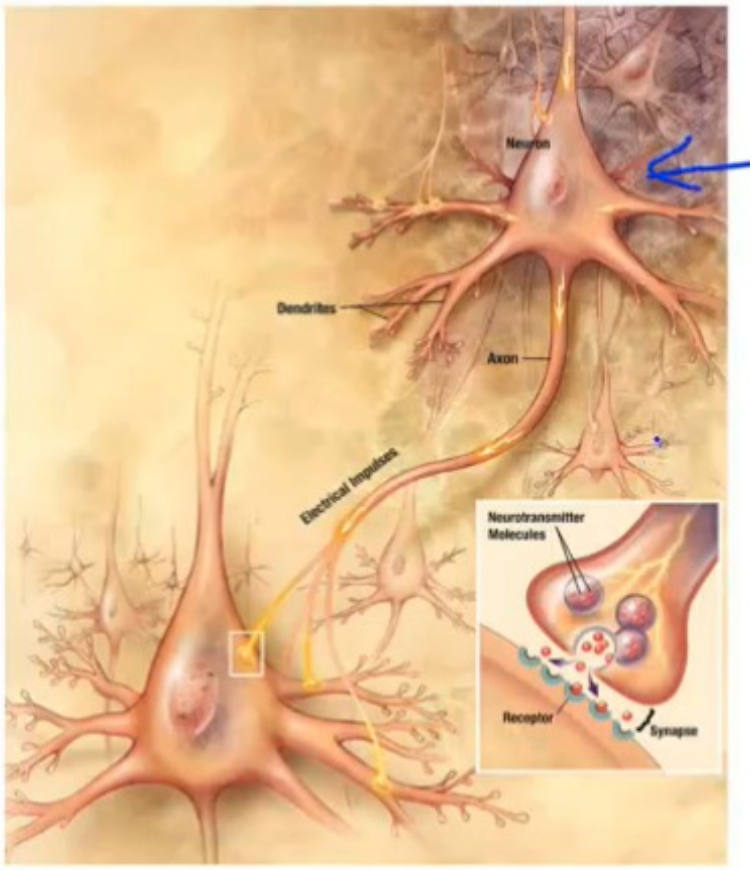


Fig. 5.5 Artist's representation of a neuron in a brain

We can also mae a network for "not"
In general, negation is implemented by putting a large negative weight.

5.1.5 Neural networks for multiclass classification

Digit recognition is a multiclass classificaiton problem. It's a version of the *one for all* classification method used for logistic regression. We assign a node in the output layer to each of the classes we wish to find. So we actually get four logistic regression classifisers representing the final step of the setting.

The way we do this we represent the training set as a vector with a single 1 in it (the rest zeros) representing the classification of the item.

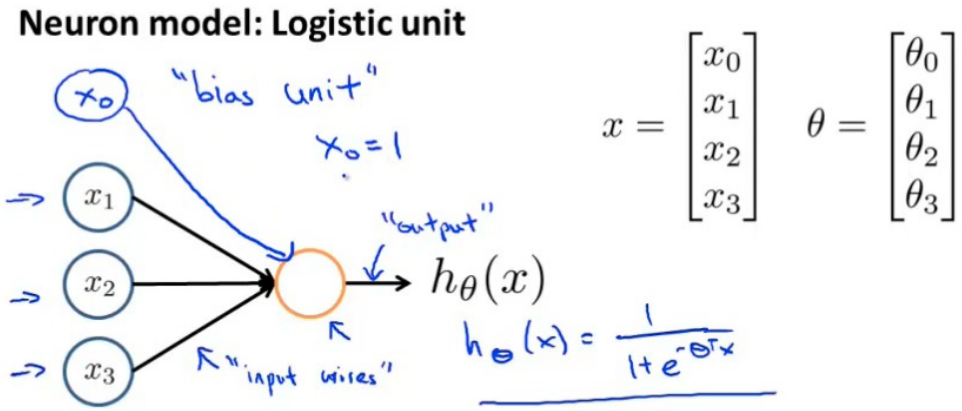


Fig. 5.6 Modelling a single neuron as a "logistic unit" (using the logistic function)

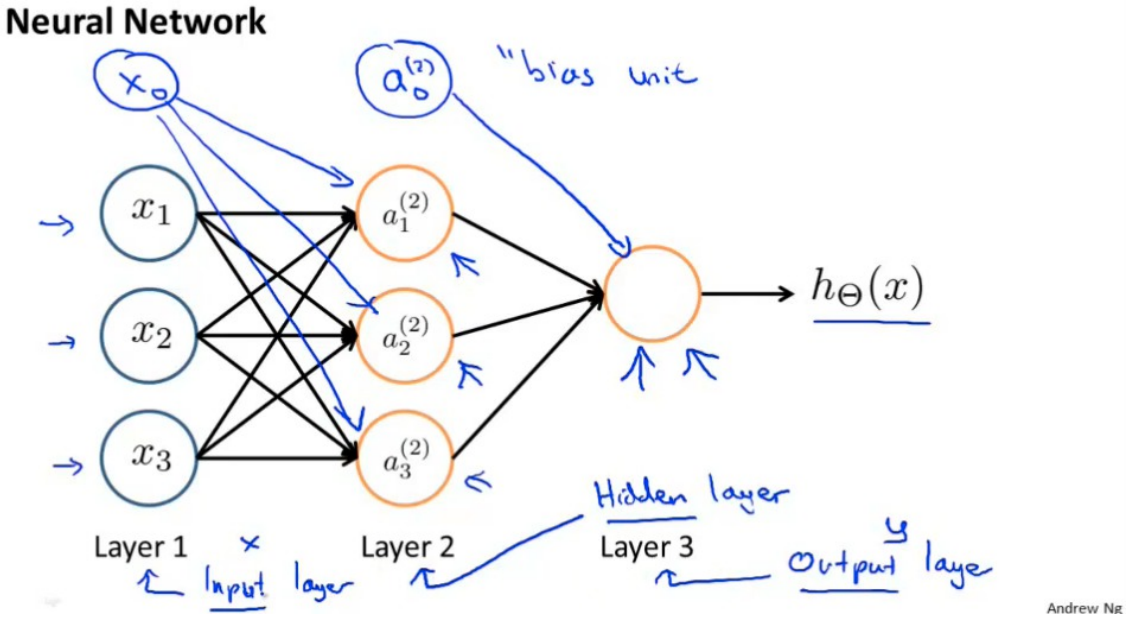


Fig. 5.7 A schematic drawing of a multi-layer neural network.

5.2 Cost functions of neural networks

NN is one of the most powerful learning algorithms we have. We'll start with the const function. We'll focus on classification problems.

We'll look both on binary classification and multi-class classifictaion. Multi-class has multiple output nodes, one per class.

We're going to generalize over the cost function we used for logistic regression.

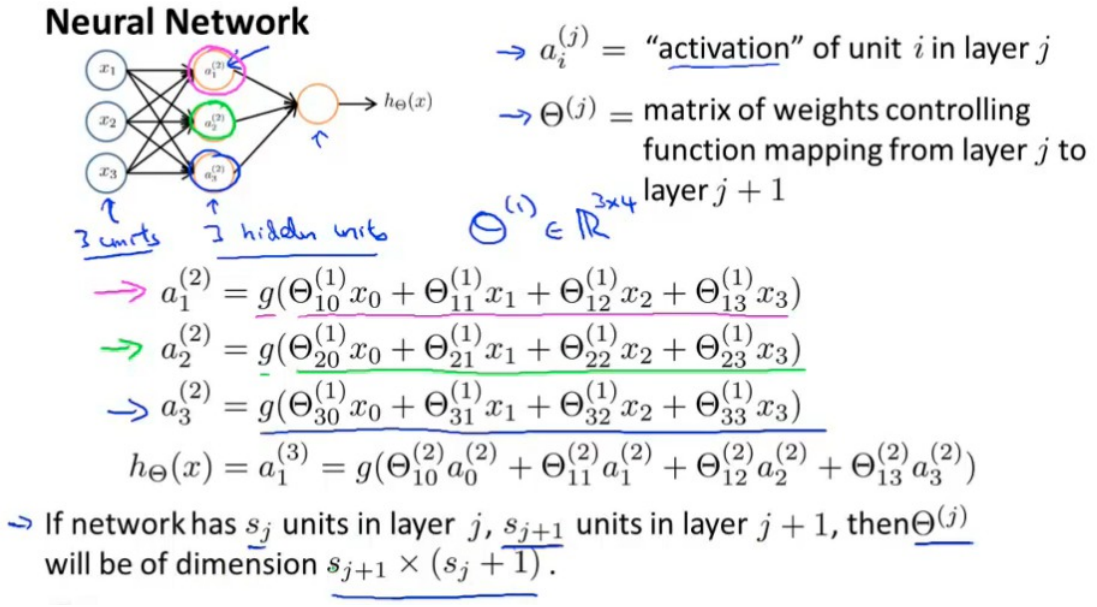


Fig. 5.8 Representing a neural network as a matrix of weights

Missing imagefile file forwardpropagation2

Fig. 5.9 Propagating values towards the output using "forward propagation"

Recall that that cost function was:

$$J(\theta) = -\frac{1}{m}\left[\sum_{j=1}^{n} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

For our neural network, we have:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{j=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log h_\Theta(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_\Theta(x_k^{(i)})\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

We sum over all the k output units, so we sum logistic regression functions, and that's basically the modification. The regularization term's summation sums over all the Θ values, except that we don't sum up the bias terms. Basically we are adding the square of the weights **Rmz:** *right?*

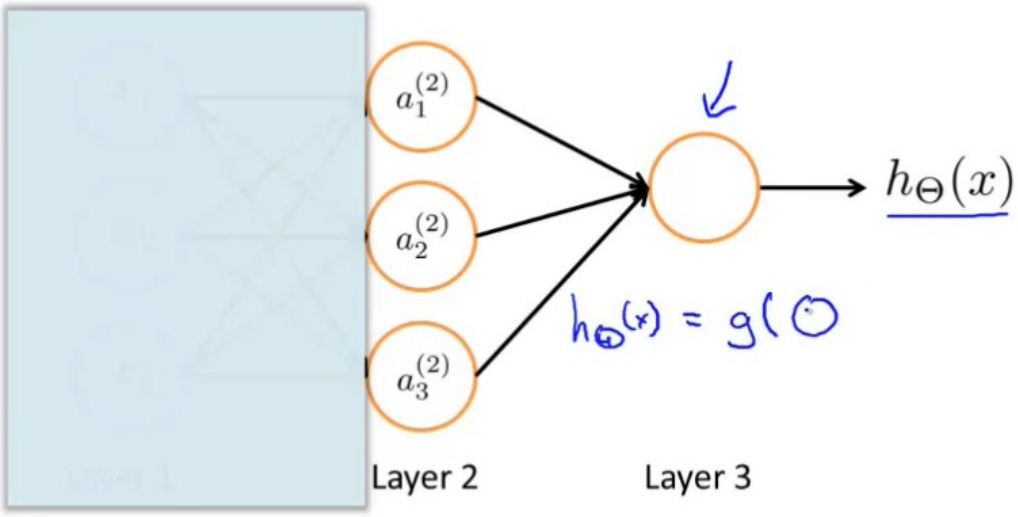


Fig. 5.10 A neural network where a couple of layers has been covered up

5.2.1 The backpropagation algorithm

We wish to find some algorithm to minimize $J(\Theta)$ so we need the partial derivatives of the cost function:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$

and those are what we'll concentrate on now. One training example case:

The *backpropagation* is an algorithm that calculates the gradient. Intuitively we have $\delta_j^{(l)}$ representing the "error" of node j in layer l. The activation of the corresponding node (for $\delta_j^{(l)}$) is $a_j^{(l)}$.

Concretely, for a L=4 node in the network seen in fig ?? is:

$$\delta_j^{(l)} = a_j^{(l)} - y_j = h_\Theta(x)_j - y_j$$

(setting $l = 4$)

A vectorized version of the equation above is $\delta^{(l)} = a_j^{(l)} - y_j = h_\Theta(x) - y$.

We then calculate the δ for the earlier layers:

$$\begin{aligned} \delta_j{}^{(3)} &= (\Theta^{(3)})^T \delta_j{}^{(4)} .* g'(z^{(3)}) \\ \delta_j{}^{(2)} &= (\Theta^{(2)}) T \delta_j{}^{(3)} .* g'(z^{(2)}) \end{aligned}$$

(where $.*$ is the elementwise multiplication operator).

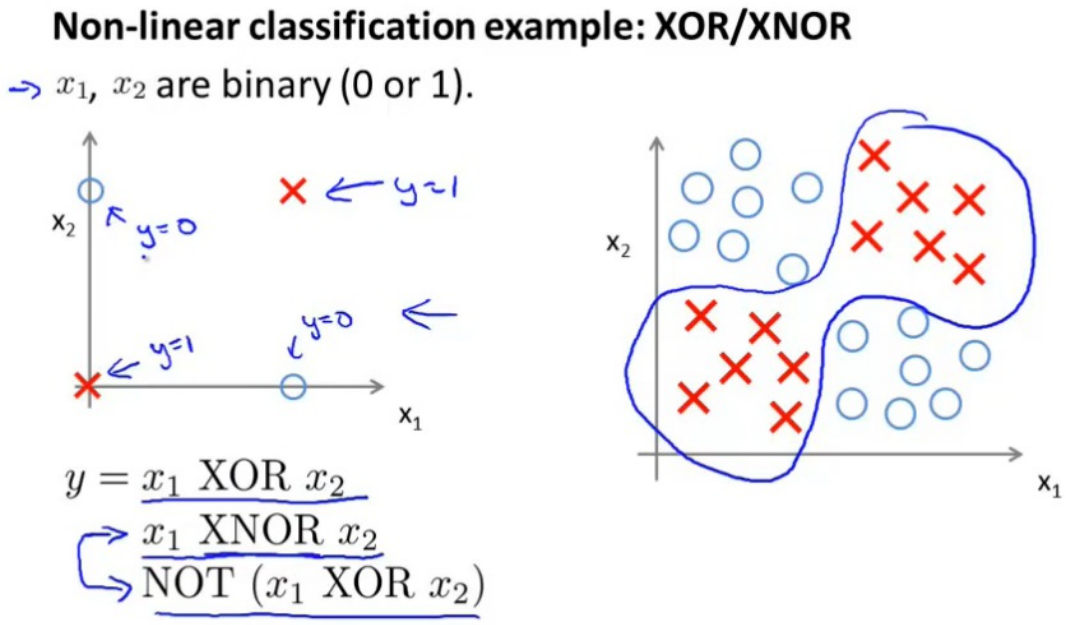


Fig. 5.11 A neural network capable of calculating the logical function exclusive or/ exclusive nor (XOR/XNOR)

And in general ${\delta_j}^{(k)} = ({\Theta^{(2)}}^T {\delta_j}^{(k)+1} .* g'(z^{(k)})$.

The function $g'(z^{(2)}$ is formally the derivative of the activation function evaluated at the input values given by

$$z^{(3)} = a^{(3)} .* (1 - a^{(3)})$$

There is no $\delta^{(1)}$ since that is the input.

The name *backpropagation* comes from the fact that the algorithm *propagate*s the errors from the output layer to the previous layers.

Finally, through a somewhat complicated mathematical proof it is possible to prove that:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l)+1}$$

(ignoring regularization terms).

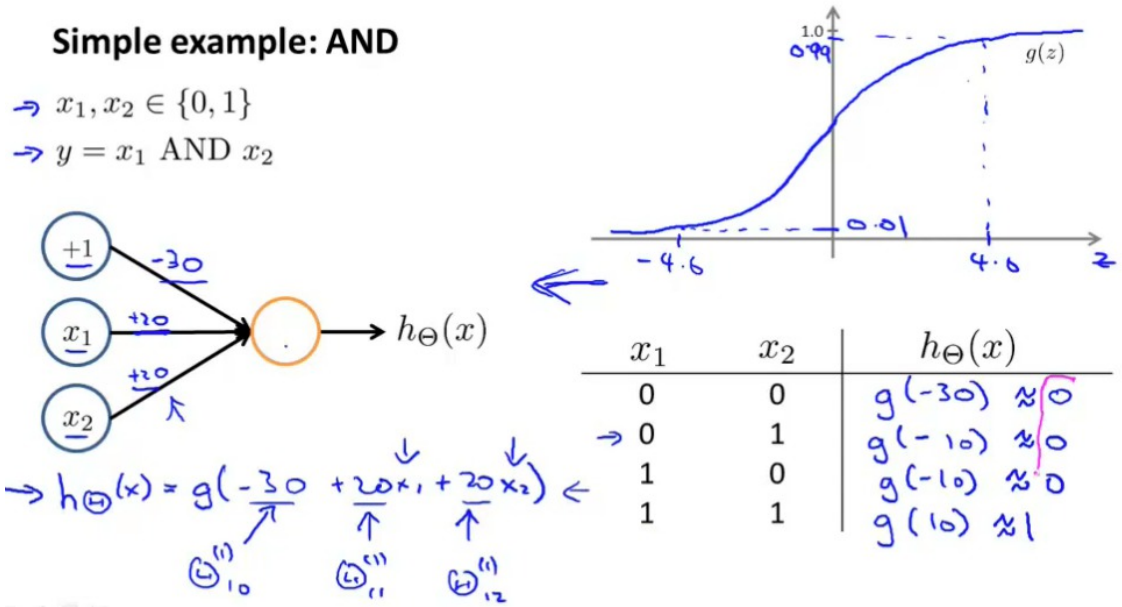
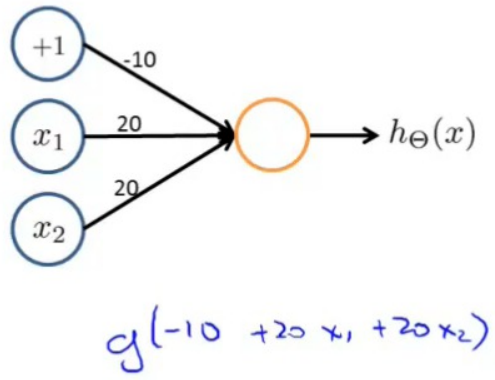


Fig. 5.12 A neural network computing the logical "and" of two parameters.

5.2.2 How to implement backpropagation for a large training set

- Assume a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(1)}, y^{(1)})\}$.
- Set $\Delta^{(l)}ij = 0$ for all l, i, j. This wil be used to compute the partial derivative term $\frac{\partial}{\partial \Theta^{(l)}_{ij}} J(\Theta)$
- Next we'll loop through the training set $(x^{(i)}, y^{(i)})$
- For $i = 1$ to m
  - set $a^{(1)} = x^{(1)}$
  - compute forward propagation to compute $a^{(1)}$ for $l = 2, 3, \ldots, L$
  - Using $y^{(i)}$, compute $\delta^{(L} = a^{(L)} - y^{(i)}$
  - Then compute $\delta^{(L)-1}, \delta^{(L)-2}, dots'()\delta 2,$
  - $\Delta^{(l)}_{ij} := \Delta^{(l)}_{ij} + a^{(l)}_j \delta^{(l)+1}_j$
- $$\begin{array}{lll} D^{(l)}_{ij} & := & \frac{1}{m}\Delta^{(l)}_{ij} + \lambda XXX^{(\Theta)} l_{ij} \quad \text{if} \quad j \neq 0 \\ D^{(l)}_{ij} & := & \frac{1}{m}\Delta^{(l)}_{ij} \qquad\qquad\qquad\qquad\quad \text{if} \quad j = 0 \end{array}$$

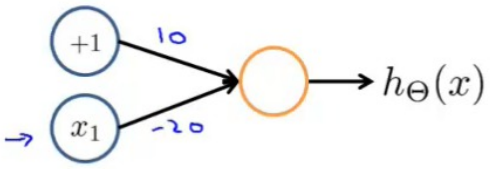
Example: OR function



x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	≈ 1
1	1	≈ 1

Fig. 5.13 A neural network computing the logical "or" function of two parameters

Negation: NOT x_1



x_1	$h_\Theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

$$h_\Theta(x) = g(10 - 20x_1)$$

Fig. 5.14 A neural network computing a logical negation

We use the Δ terms to accumulate the error terms. The final step can be vectorized using:

$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l)+1}(a^{(l)})^T$$

While the formal proof is quite complicated, it is in fact true that

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

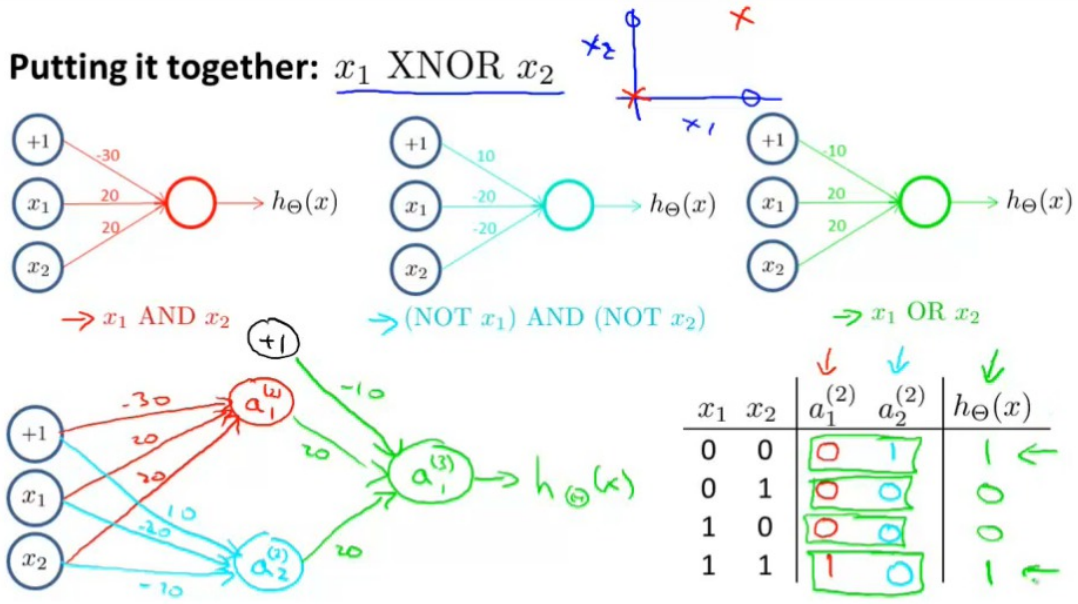


Fig. 5.15 A neural network computing the "exclusive nor" (xnor) logical function

This means that we an use these facts in both gradient descent and the more advanced optimization algorithms.

5.2.3 The mehanical steps of backpropagation

Backpropagation is a less mathematically clean algorithm, so it's ok to be a bit confused :-)

Backpropagation is a lot like running forward propagation backwards.

5.2.4 Implementation note: Unrolling

The *fminunc* like algorithms of this world requires vectors as inputs and deliver them as outputs. But in NNs we are working on matrices.

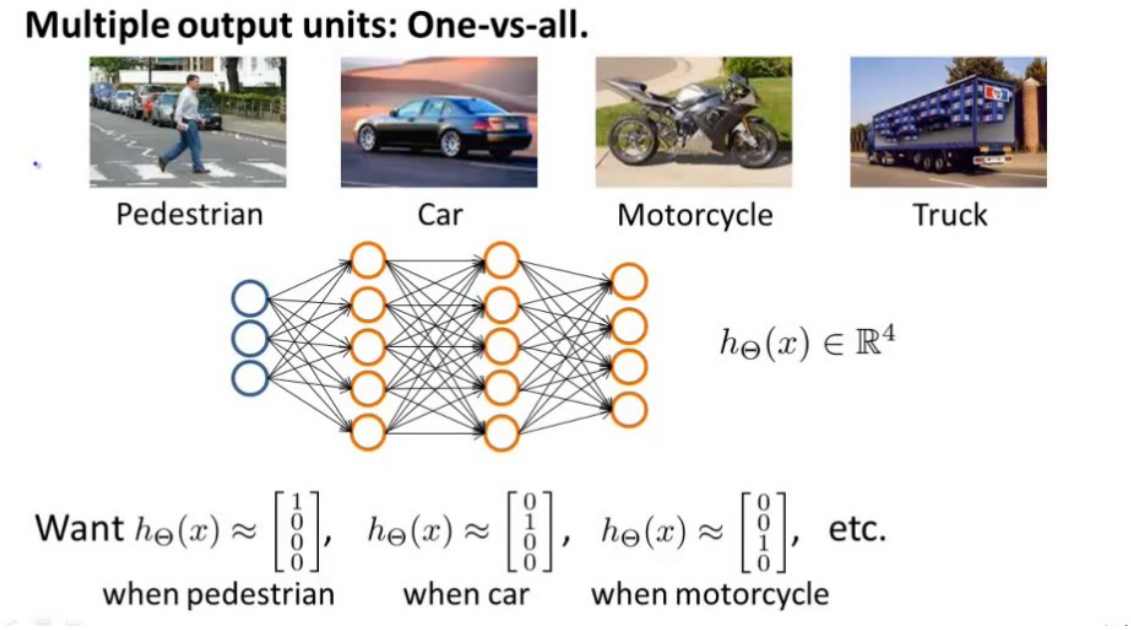


Fig. 5.16 A neural network for solving a a classification into multiple classes

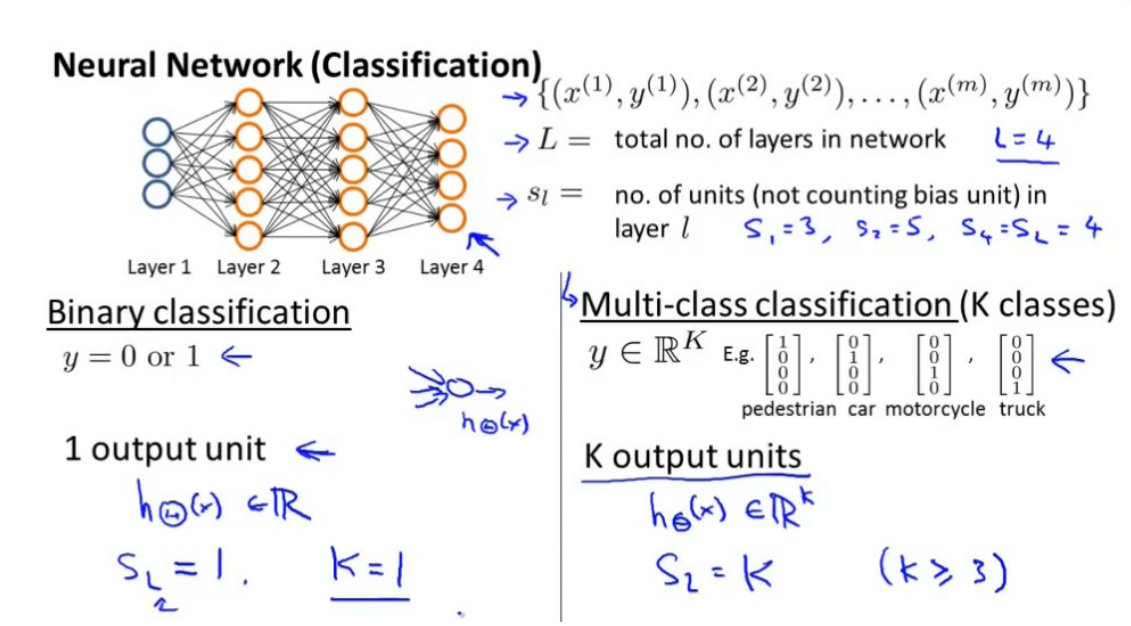


Fig. 5.17 The cost function for a neural network.

Cost function

Logistic regression:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

Neural network:

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log((h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Fig. 5.18 A cost function for a neural network with regularization terms.

Gradient computation

Given one training example (x, y):

Forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$

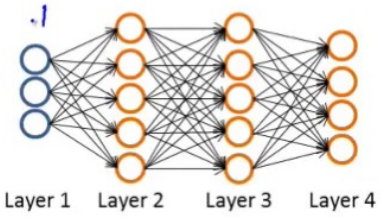


Fig. 5.19 Computing the gradient of a neural network

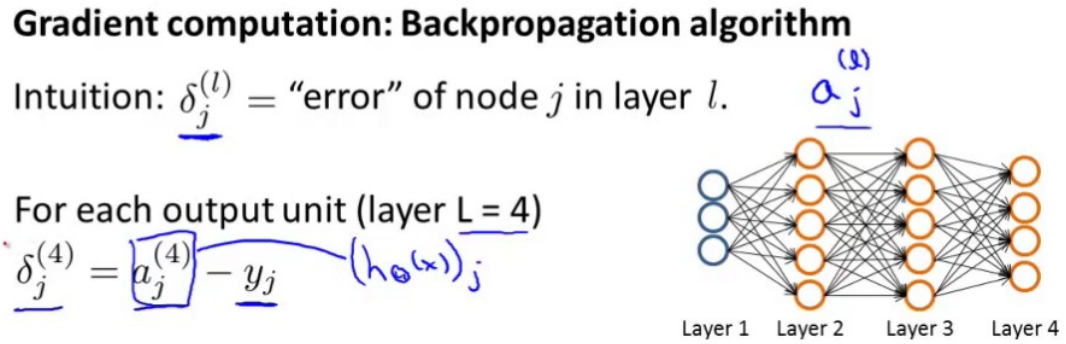


Fig. 5.20 Computing the gradient of a neural network using the backpropagation algorithm

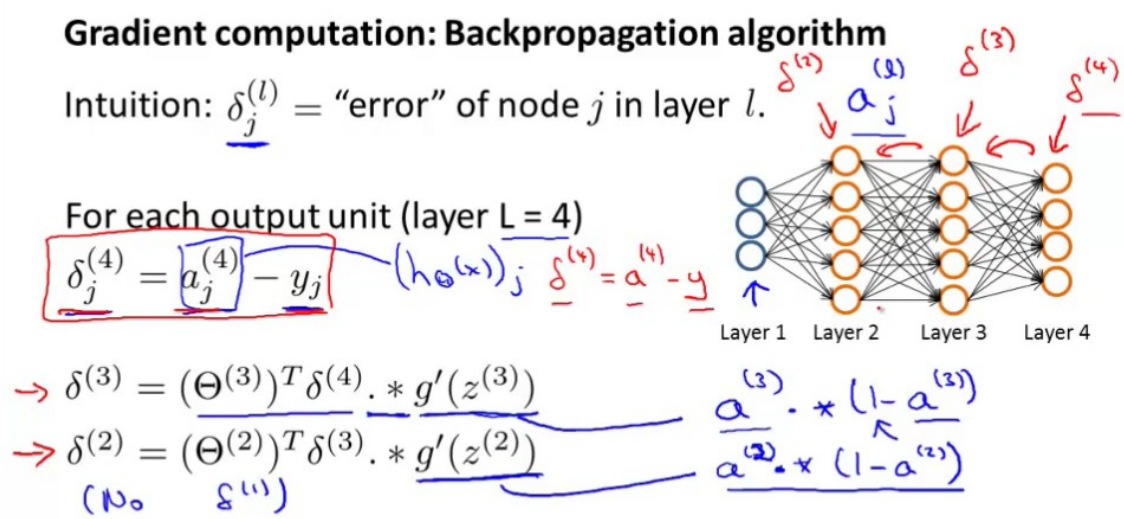


Fig. 5.21 Computing the gradient of a neural network using the backpropagation algorithm

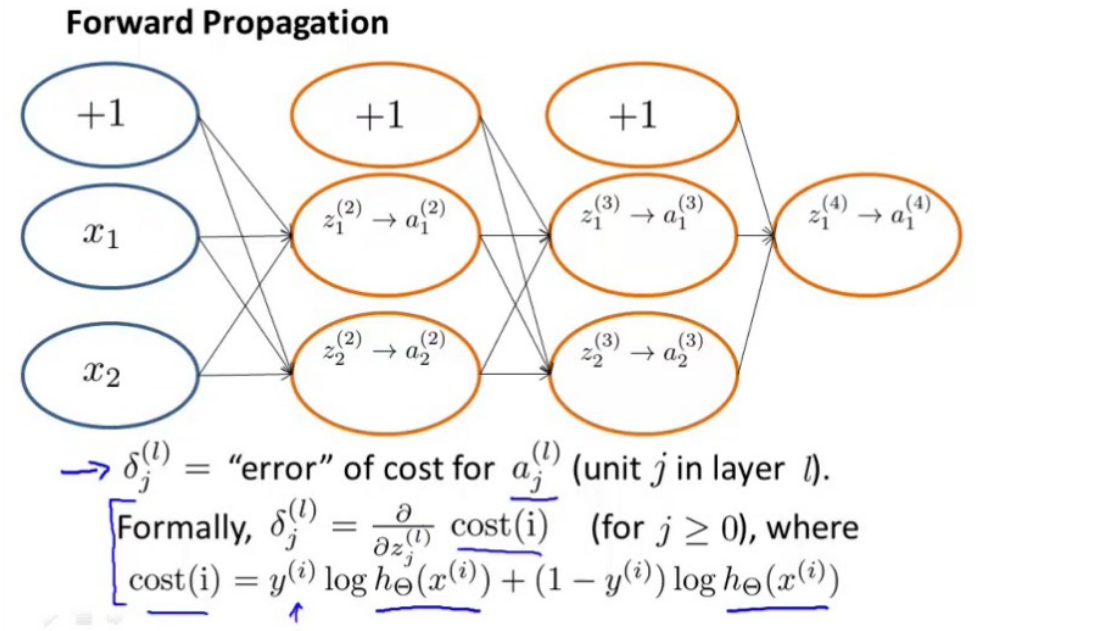


Fig. 5.22 NNForwardIntuition

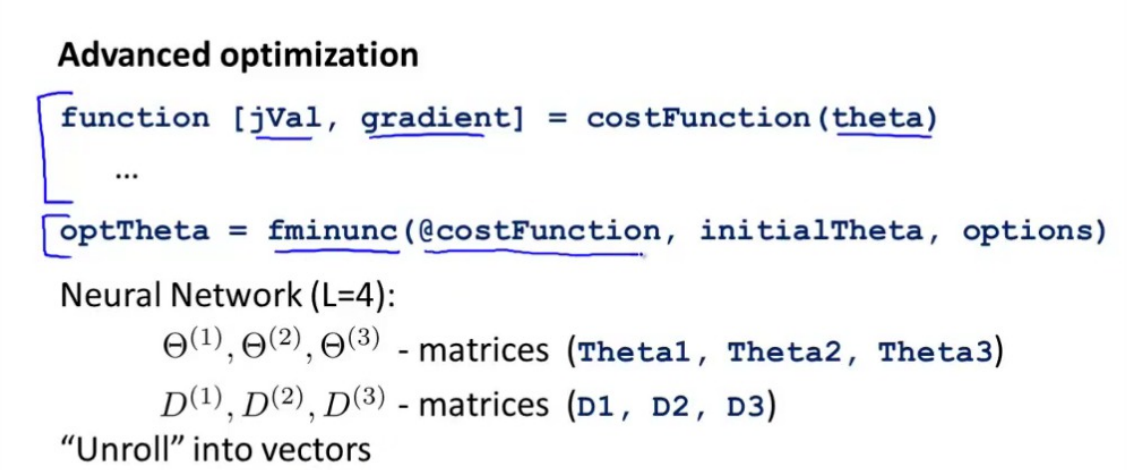


Fig. 5.23 The weight matrices are rolled into / unrolled from rows in matrices

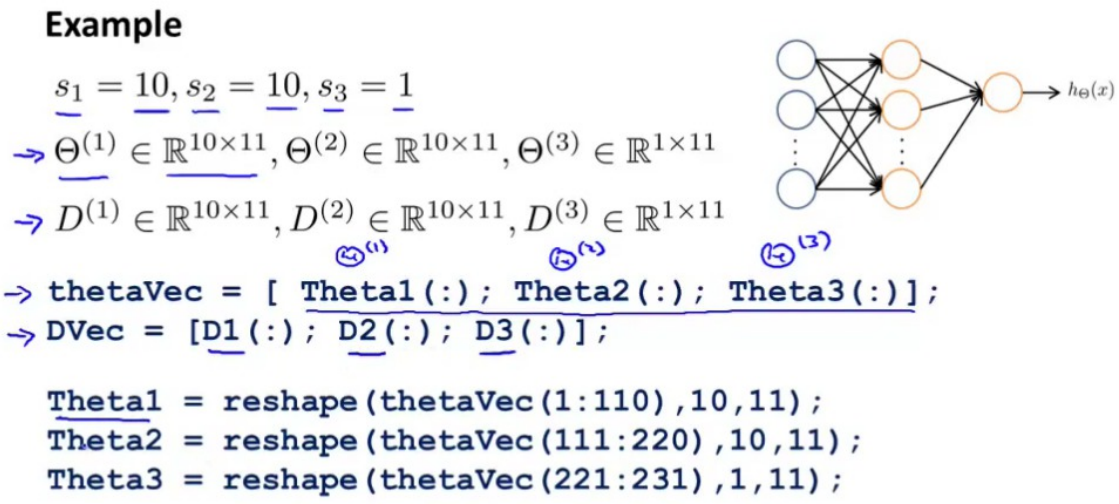


Fig. 5.24 With the unrolled thetas, we can use standard advanced optimization techniques to find parameters for the neural network

The fix is to unroll them into vectors. Remember that the indexes that are used when rolling/unrolling are *inclusive*, so as always beware of off-by-one errors caused by bogus intuition (it's an easy thing to do).

5.2.5 Gradient checking

There are many ways to get subtle errors in backpropagation. It's also not so easy to know about it. There is a trick called *gradient checking* that eliminates many of these problems and thus gives you a higher confidence that your implementation don't have bogus bugs :-)

Compute $\theta + /\epsilon$ and connect them through a straight line and use the slope as an approximation of the derivative. There is a range of epsilons that will work, but you don't want it to be *too* small since that will lead into numerical problems.

The two-sided estimate (+/) gives better estimate than the one-sided one. This gives us a numerical estimate of the gradient at some point.

We can then estimate all the partial derivative terms for a whooping big vector θ as shown in fig **??**. The trick is to calculate the partial change per variable, and use that to produce the partial derivative for the whole parameter set.

Take the gradient we get from backprop and check if it's more or less similar to the gradient approximation.

Prof Ng's standard way of implementing BP is:

- **Implementation note:**

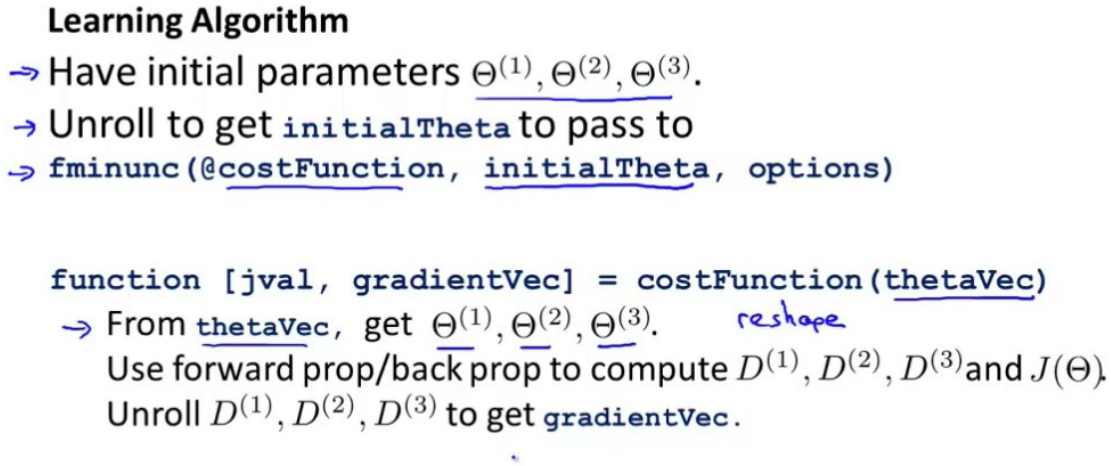


Fig. 5.25 The learning algorithm using an unrolled representation

  - Implement backprop to compute DVec (unrolled $D^{(1)}, \ldots, D^{(1)}$).
  - Implement numerical gradient check to compute gradApprox.
  - Make sure that they give similar values.
  - Turn off gradient checking. Using backprop code for learning

- **Important::**
  - Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of "costFunction", your code wil be *very* slow.

Backpropagation is actually a very computationally efficient algorithm to calculate the gradient.

5.2.6 Random initialization

There is one final idea we need to get this thing going, and that is *random initialization*. We need some initial theta. Zero is actually not an option for NN (even if it worked for logistic gregression) since it all deltas will be zero, and we won't get a gradient to traverse. All the hidden units computes the same values initially and that's not working. The delta values will all be the same. The partial derivatives will be equal. That means that the weights will be the same even if

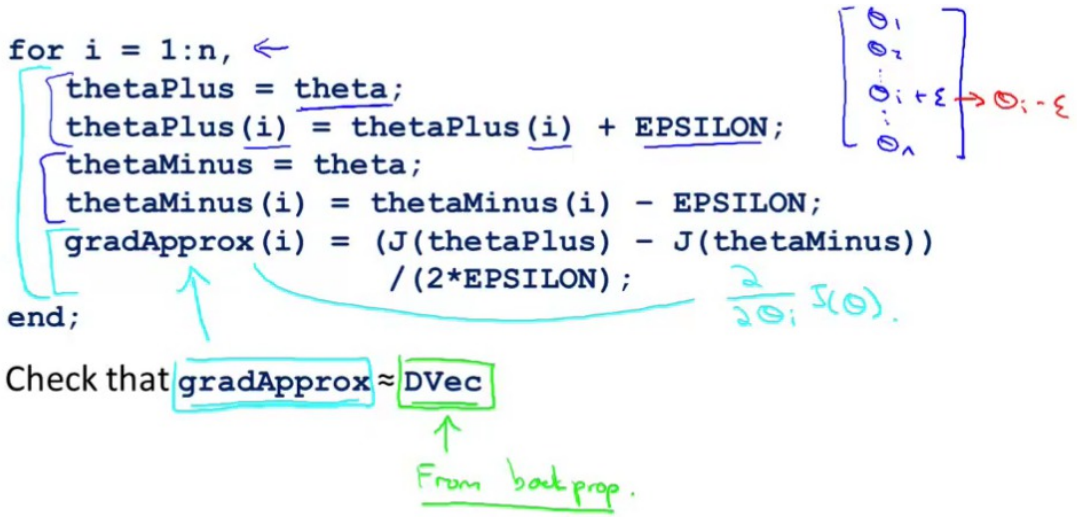


Fig. 5.26 Gradient checking of the NN algorithms, just to make certain that things are on track.

```
for i = 1:n, ←
  thetaPlus = theta;
  thetaPlus(i) = thetaPlus(i) + EPSILON;
  thetaMinus = theta;
  thetaMinus(i) = thetaMinus(i) - EPSILON;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                  /(2*EPSILON);
end;
```

Fig. 5.27 Even more gradient checking in Octave

they are non-zero. In essence, we won't get efficient learning. Not only will the weights be identical, but all the outputs from all the hidden units will compute the exact same feature. This is a highly redundant representation and it prevents the network from learning something interesting.

This is called the *problem of symmetric weights*.

We get around the problem by setting the initial weights to be some small random number (see fig **??** for an octave we perform *symmetry breaking*implementation), and this is how we perform *symmetry breaking*.

The ϵ we have here is different from the ϵ used during gradient checking, hence

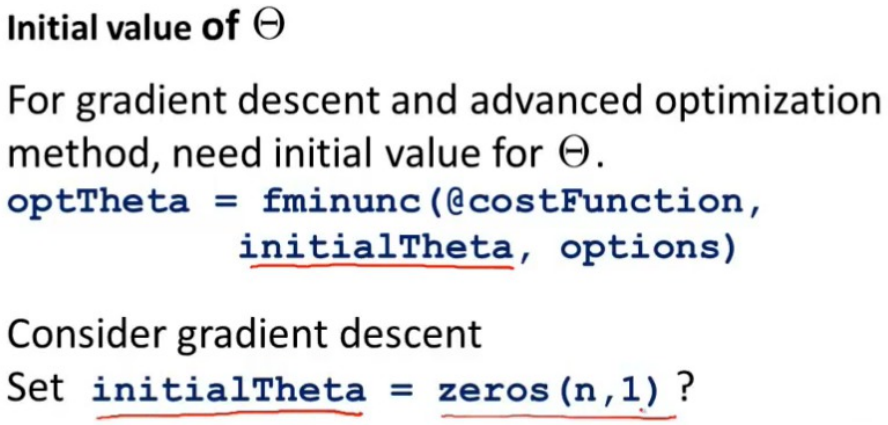


Fig. 5.28 Initializing the thetas using random numbers

the name `INIT_EPSILON`.

5.2.7 Putting it together

The architecture

Pick a network architecture. The number of features should be the number of output nodes. The number of input units should be the number of features. As a default, just have one hidden layer. If you have more the same number of hidden units in every layer. Usually the more the better but the computational cost can be prohibitive.

A bit more nodes than the number of input features is usually ok for hidden layers.

There will be more about how to do this later.

Training a neural network

1. Randomly initialize weights.
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$.
3. Implement code to compute the cost function $J(\Theta)$.
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta^l}_{jk} J(\Theta)$. Use a forloop like the one in figure ??.

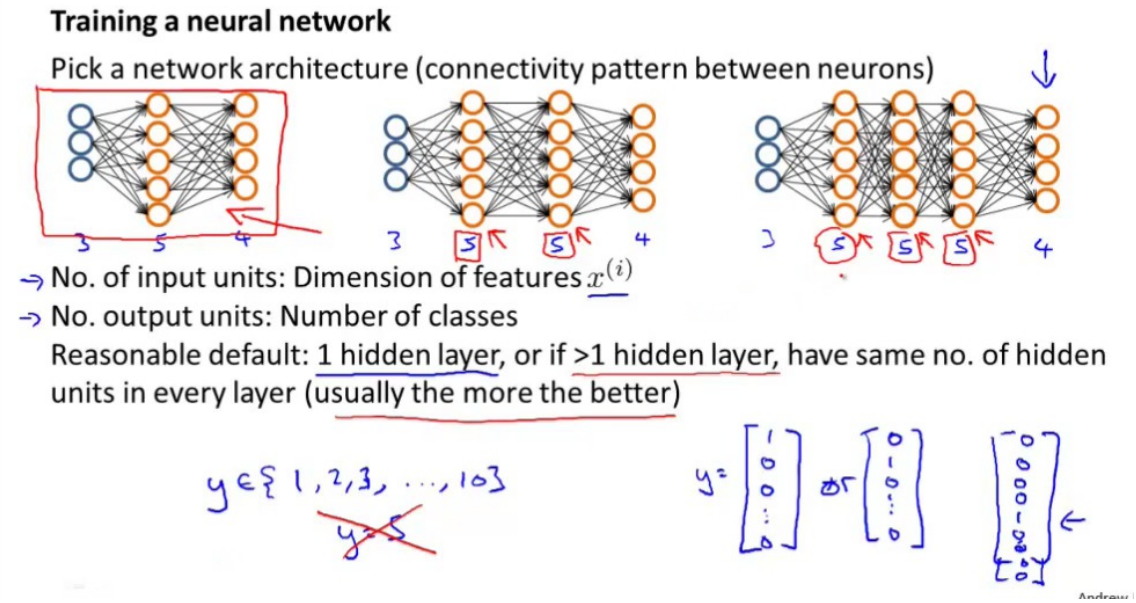


Fig. 5.29 The "architecture" of a neural network is the number of layers, and the way in which the layers are connectec

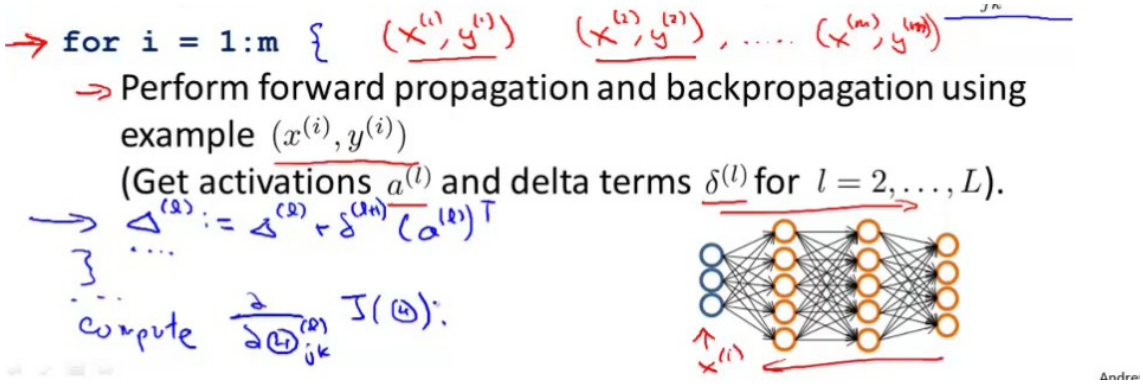


Fig. 5.30 NNTrainingForloop

5. Use gradient checking to compare $\frac{\partial}{\partial \Theta^{l}_{jk}} J(\Theta)$. computed using backpropagation vs using numerical estimate of gradient $J(\Theta)$. Then disable the gradient checking.

6. Use gradient descent or an advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ.

Even if the functions can't actually guarantee to find a global optimum, but in fact the results are usually quite well even when the error function is non-convex.

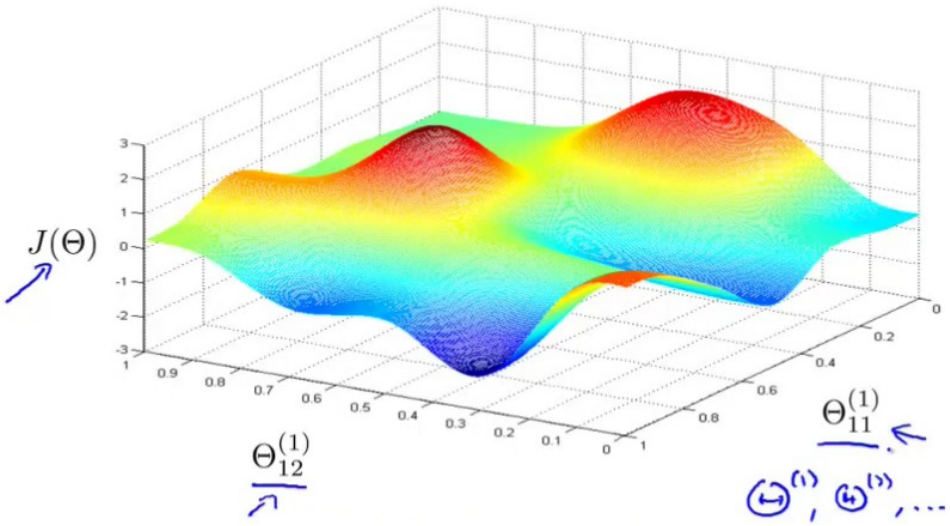


Fig. 5.31 errorsurface

The cost function measures how well the network matches the training data. There are parameter settings that approximate well and others that don't.

Neural networks is a complex algorithm. It's quite normal to not have a full grasp of what the algorithm does. It's hard to get a handle on it compared with for instance linear regression. However, backpropagation is a very powerfl learning method that is able to fit nonlinear data to your classifier and this is one of the most powerful learning algorithms we have today.

5.2.8 Autonomous Driving

Dean Pomerleau at *Carnige Mellon University* made a self-driving car based on neural networks. In figure ?? the topmost input on the left is the steering input from the human, the lower bar is the output from the neural network. lower left window shows the video input that the camera sees. Before the network starts learning it outputs a gray fuzz corresponding to the random initialization.

Only after it has learned for a while it starts steering in a distinct direction.

Takes a picture every second. Reduced to 30 times 30 image (900 pixels). A three layer netwrk. Initial steering response is random, but after about two minutes the network accurately replicates the steering reaction of the human driver.

When running it uses twelve images per second. There is a "run" switch that must be pushed. The network generates both a steering direction and a confidence value.

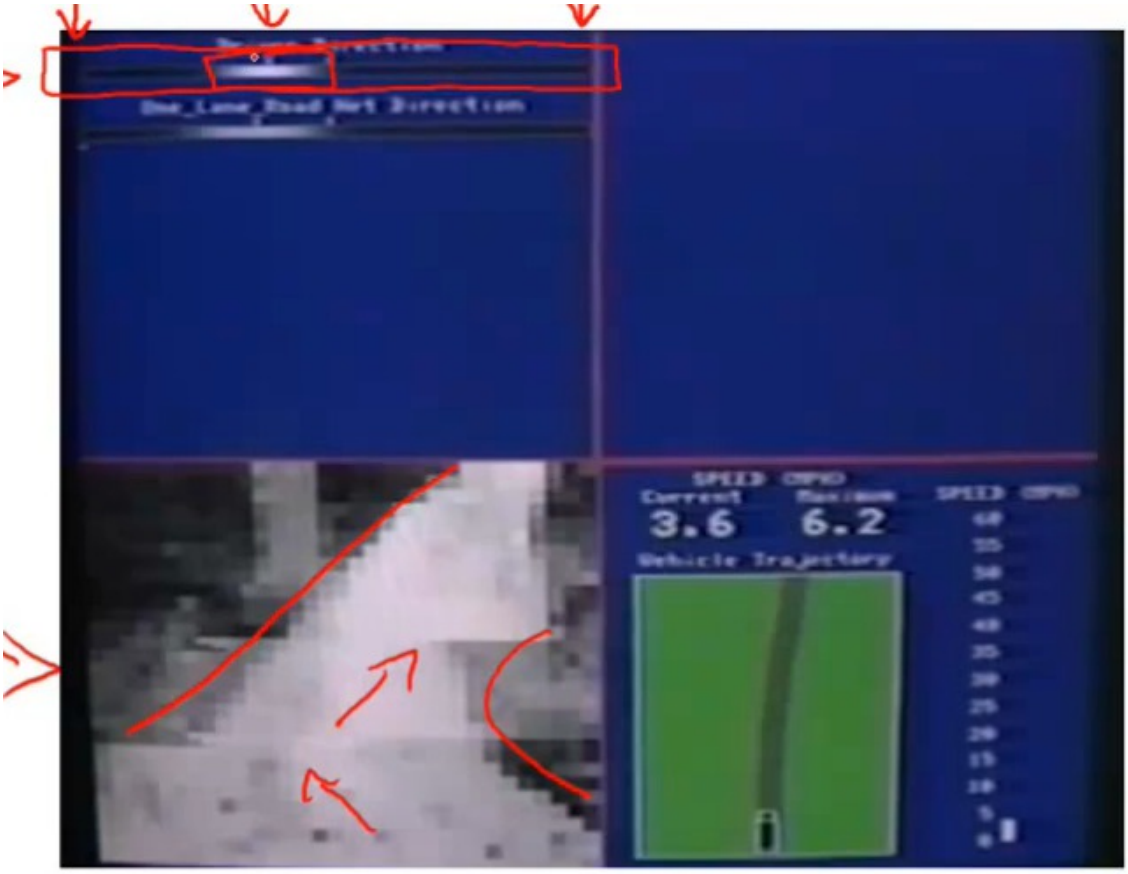


Fig. 5.32 autonomousdriving

Missing imagefile file alvin

Fig. 5.33 alvin

There are multiple networks running at the same time, and the most reliable is chosen. At an intersection As the vehicle approaches the intersection the confidane of the one-lane netowkr decreases. As it sees a two-lane road the two-lane network is selected to steer to sfey guide the network into its lane.

alvinfromtheinside2

There are more modern drivers than this, but it's still pretty amazing that a simple neural network can be trained to drive a car somewhat well.

Fig. 5.34 alvinsensors

Fig. 5.35 alvinontheroad

Fig. 5.36 singlelaneroad

Missing imagefile file alvinfromtheinside

Fig. 5.37 alvinfromtheinside

Missing imagefile file alvinfromtheinside2

Fig. 5.38 alvinfromtheinside2

6 Advice for applying machine learning

How to apply the right algorithms. How to avoid wasting time on non-promising avenues. This section will give a bunch of practical guidelines.

We'll consider minimizing regularized linear regression:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{i=1}^{n}\theta_j^2\right]$$

What to do if we find unacceptably large errors in the predictions? What should we do?

Some possible fixes are:

- Get more training data. Often it helps, sometimes it doesn't.
- Try a smaller set of features. If we have overfitting that is something we can do.
- Get more features. Could work, but it would be nice to know in advance if it would work.
- We could add polynomial features.
- We could increase λ.
- We could decrease λ.

Unfortunately the most used method for selecting among these is gut feeling. Unfortunately this is not very efficient. Fortunately there is a fairly simple technique that can easily rule out about half of these options. We'll learn about these *machine learning diagnostics*. Diagnostics can take time to implement, but doing so can be a very good investment since it helps making choices when improving the algorithm and many months can be saved that way.

6.1 Evaluating an hypothesis

When fitting data to the hypothesis we are trying to minimize the training error. Too many parameters in a polynomial will not be very useful since we are overfitting. For few parmeters we can use inspection, but for high number of parameters that isn't practical.

A more general method is to split the data set into a training set and a test set. A typical split is 70/30. The number m_{test} is the number of items in the test set. The training set is used to train the algorithm (learn the parameter vector)., and the test set is used to evaluate its performance. If there is any kind of ordering in the data, it is better to split the data randomly rather than use the sequence they arrived in.

- Learn the parameter $J(\theta)$ using the training data minimizing the training error. You use 70% of the test learning data for this.
- Then you compute the test set error:

$$J_{\text{test}}(\theta) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{\text{test}} \left(h_\theta(x_{\text{test}}^{(i)}) - yx_{\text{test}}^{(i)} \right)^2$$

  Use the squared sum of errors or something similar.

How about if we were using a classification problem? It's very similar:

- First we learn the parameter θ from the training data.
- Then we compute the test set error:

$$J_{\text{test}}(\theta) = -\frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \log h_\theta(m_{\text{test}}) + (1 - y_{\text{test}}^{(i)} \log h_\theta(x_{\text{test}}^{(i)})$$

  The same objective function that we always use for logistic regression, but use m_{test}.

  Most of the time this is a perfectly good way of calculating classification errors, but sometimes there is another metric that is also useful:

- Misclassification error (0/1 misclassification error): 0/1 denotes that an example is either right or wrong:

$$\text{err}(h_\theta(x), y) = \begin{cases} 1 & \text{if} \quad h_\theta(x) \geq 0.5, y = 0, \text{or} h_\theta(x) \leq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Or in other words. If the prediction was wrong the function gives "1" and if it was right it gives "0", so summing all the errors will give the number of actual errors. We can then let the tes error be defined as:

```
\[
\frac{1}{m_\mbox{test}} \sum_{i=1}^{m_\mbox{test}}
\mbox{err}(h_\theta(x_\mbox{test}^{(i)}), y^{(i)})
\]
```

The error is the fraction of the misclassifications.

That's the standard technique for testing how good a learned hypothesis is.

6.2 Model selection and training/validation/test sets

How to select the regularization parameter, number of terms in a polynomial etc.? These questions are called *model selection problems*. We'll learn how to split the data into *training set validation set* and *test set* and use these to do model selection.

We have already seen many examples of overfitting. This is why training set errors are not a good predictor for how the data will fit other data and generalize to other data not seen in the training set.

In general, the error on any error data is unlikely to be a good predictor of generalized error.

Consider the case where we are looking at polynomials with up to tenth degree terms. The model selection problem we are trying to solve is to figure out which degree of polynomial, denoted d we should use to model the problem.

Here is one thing we can do. First take the first model and minimize the training error, and do the same thing for increasing values of d. We can then take the parameters and compute the test set errors. We can then see which model has the lowest tett set error. The problem is that this will probably not be a fair comparison since we fit the parameter for giving the best possible performance on the test set, so the performance on the of the parmeter is likely to be overly positive, so we need another test set, called the *cross validation set* (CV), sometimes called *validation set*. A typical ratio would be 60/20/20. These numbers can vary but this is typical.

When faced with a model selection problem, we'll use the cross validation set to select the model. We'll select the model with the *lowest* cross validation error. Then finally we'll use the *test set* to determine the performance of the estimator.

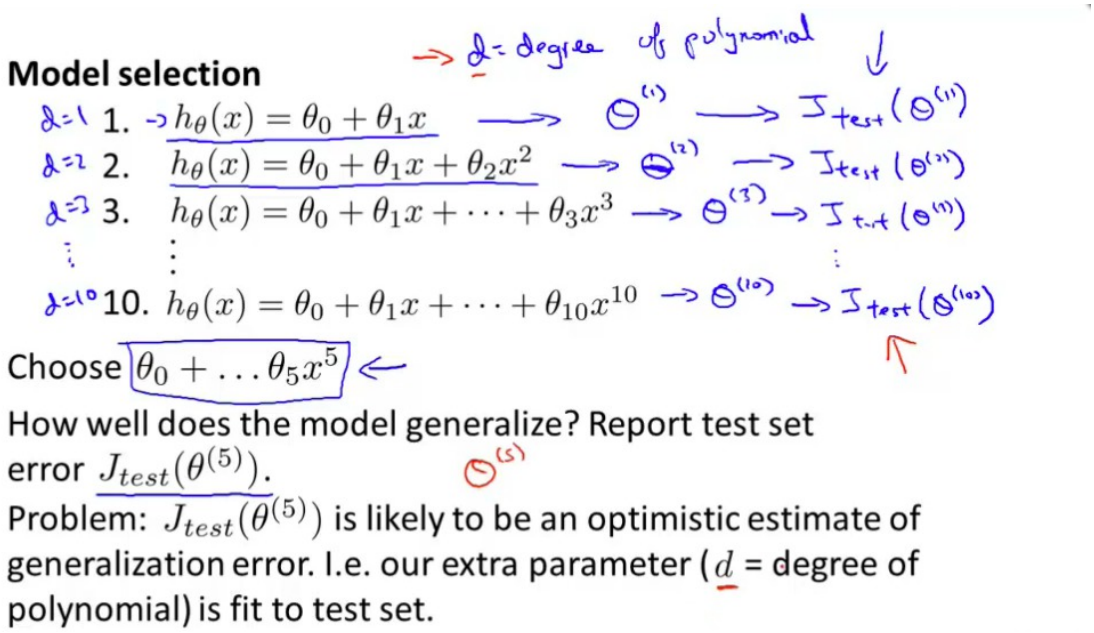


Fig. 6.1 A ten degree polynomial can fit many a crooked curve, but should it?

In machine learning as it is practiced the model is often selected using the test set and reporting the error on the testset. Most practitioners advise against this. If you don't have a really huge testset this is most likely a pretty bad idea.

6.3 Diagnosing bias v.s. variance

If you run a learning algorithm and it is not doing as well as hope, it is almost always because it has a *high bias problem* or a *high bias problem*. In other words, either an underfitting problem or an overfitting problem. It is very important to figure out which (or both) of these you have to improve your algorithm.

We'll now consider the error as a function of the model selection parameters. The plot can be found in figure ??.

Increasing the degree of the polynomial wil typically let the training error fall with increasing degree of the polynomial. However, that may not be the case for the crossvalidation error. That may in fact increase with higher model parameters.

In figure ?? the left side corresponds to a low bias case, the right end corresponds to a high variance problem. The high bias case (underfitting) case will give high both training and validation errors. If the training error is low and the cross validation error is low then there is a variance problem (overfit).

Train/validation/test error

Training error:

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Cross Validation error:

$$J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Test error:

$$J_{test}(\theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

Fig. 6.2 tvterrors

The key that distinguishes these two cases the training set error will be high in high bias problems and it will be low in high variance problems.

There are in fact four distinct failure modes:

- High bias (overfitting)
- High bias (underfitting)
- High variance (overfitting)
- High variance (underfitting)

It is also possible to suffer from both high variance and high bias .

Rmz: *Obviously I don't grok this yet*

6.4 Regularization and bias variance

Regularization can help against overfitting, but how does it help against bias invariance. Assume we have a high order polynomial with regularization.

With a high λ we'll get high bias and overfitting. With a zero value we get high variance.

We can use this to automatically chose the regularization value. Ng usually selects values of λ in steps of .02 starting at 0.01 up to 10.24 (for twelve models). The same procedure can be used as for the d parameters and test them using the cross validation set for validation.

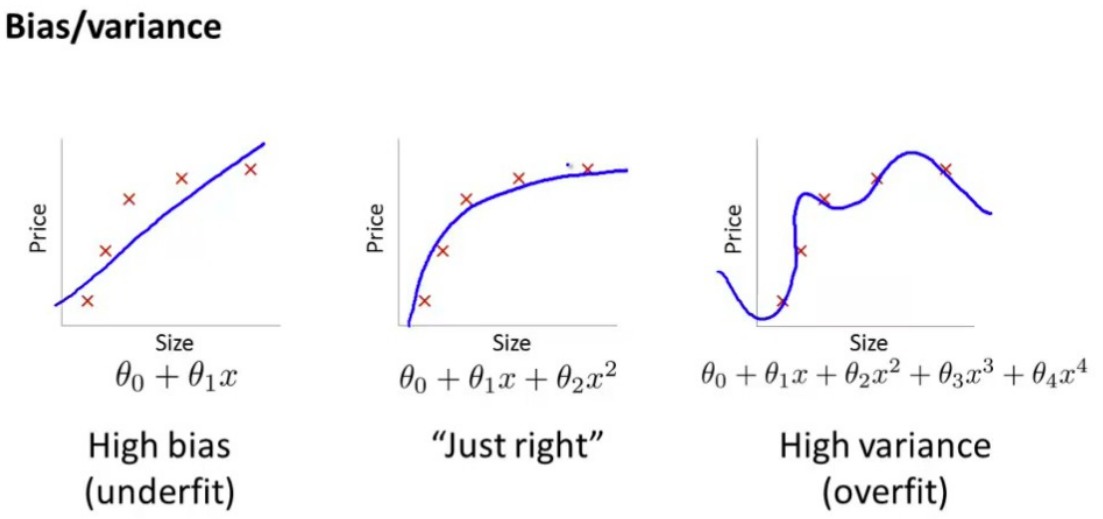


Fig. 6.3 biasvariance

One typical to test for would be

- Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$
- Penalty:
$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{i=1}^{m} \theta_j^2$$
- Range to test over $\lambda \in \{0, 0.001, 0.02, 0.04, 0.08, \ldots, 10.24\}$. (a sequence doubling the λ for each step).
- Calculate

```
 \[
\mbox{min}_{\theta} J(\theta) \rightarrow
  \theta^{(i)} \rightarrow J_\mbox{cv}(\theta^{(i)})
\]
```

  for all the λ_i values.
- Select the λ that gives the lowest crossvalidation error.
- Finally calculate the test error for the selected $\theta^{(j)}$.

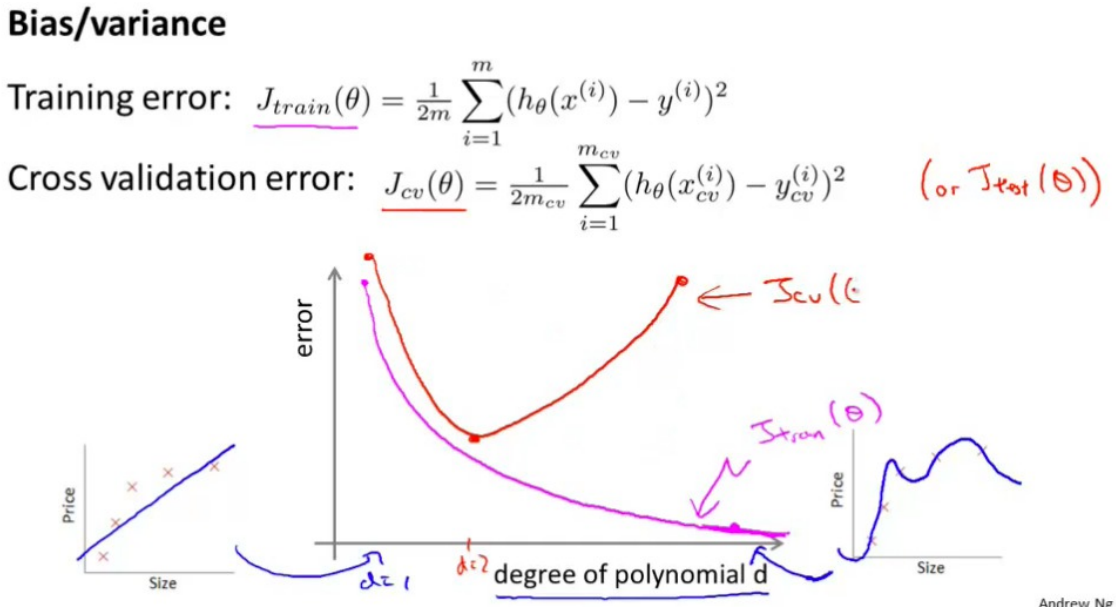


Fig. 6.4 biasvarianceplot

When choosing the regularization parameter, we use a training error, cross validation and test error defined *without* the regularization factor (half average squared error, typically).

It often helps to plot a figure like the one in fig ??.

Next we'll look at *learning curves* to diagnose problems with learning algorithms.

6.5 Learning curves

Learning curves is a very useful tool to gauge the performance of a learning algorithm

A learning curve is a plot of the training and cross validation errors as a function of the training set. The trick is to deliberately reduce the size of the training set.

With small training sets the training errors will be very small. With larger ms there will be larger and larger errors for the training set.

For small training sets the cross validation error will tend to decrease the larger the training set is.

If your model can't match the data, the cross validation error will drop a bit, but then it will flatten and will stay high. For high bias, the crossvalidation error and the training errors are high, and relatively similar. Consequently, if a learning algorithm has high bias, adding more data to the training set *will not improve*

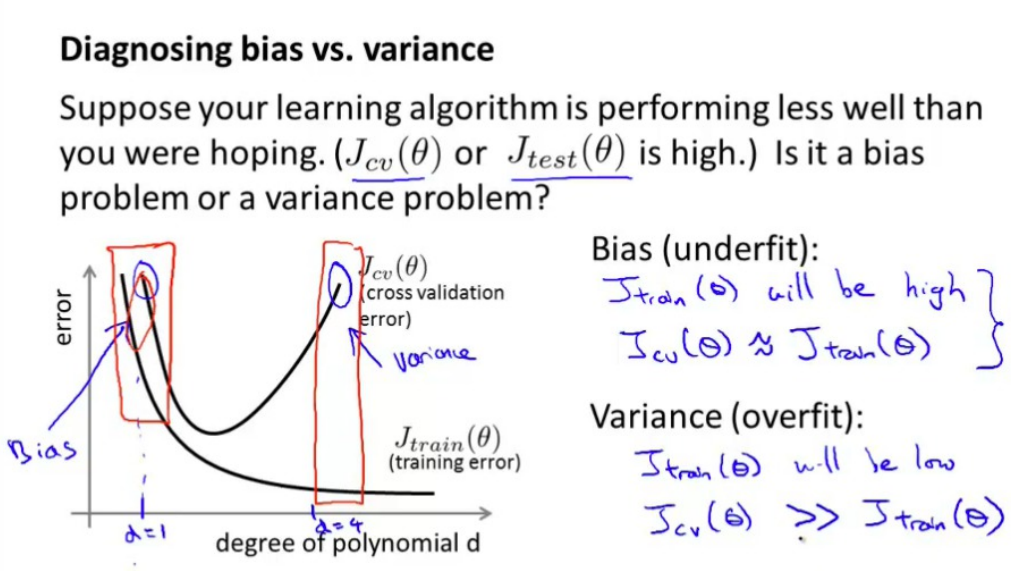


Fig. 6.5 biasvariance2

performance. This is a useful thing to know before setting off to getting tons of extra datan (in this case it may be useful).

If we are fitting a high order polynomial and a small λ we will have little training error but as the training set increases the training set error will still be pretty low. However, the cross validation error will be high. The indiciative diagnostic is that there is a high gap between the training error and the cross validation error. However, adding more data will decrease the gap, so with high variance learning algorithms adding more data will indeed help performance.

Plotting learning curves can be useful to figure out if the learning algorithm is suffering from bias or variance or both. Plotting learning curves is always a good idea.

6.6 Deciding what to next revisited

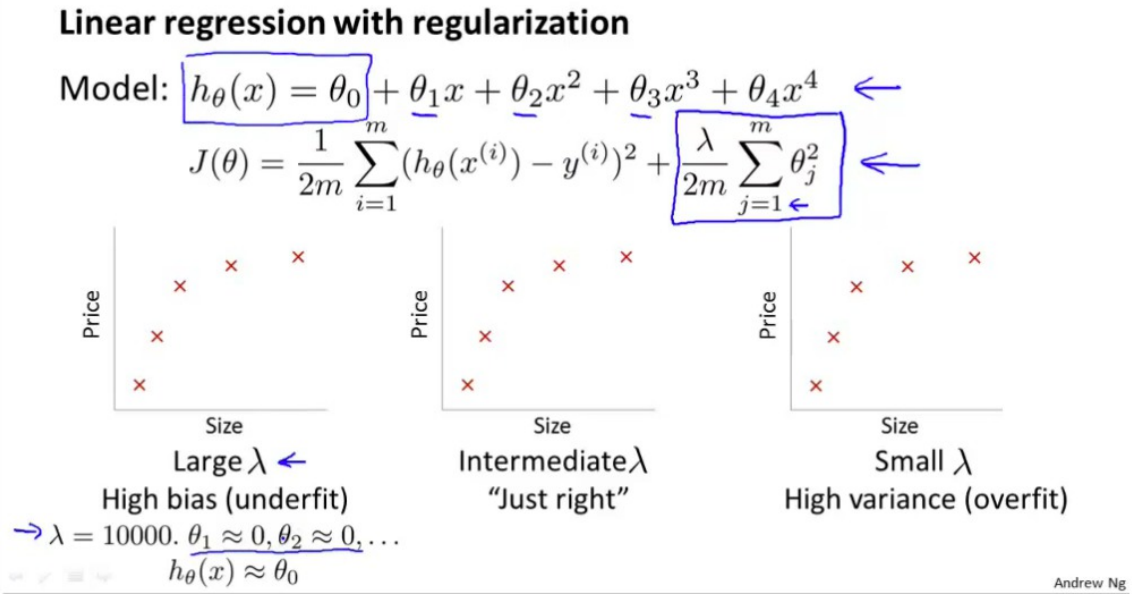


Fig. 6.6 linregularization

Cure	**Fixes problem type**	**Indication**
Get more training examples	High variance	CV error much large
Try smaller set of features	High variance	As above.
Try getting additional features	High bias	???
Try adding polynomial features (x_1^2, (x_2^2, x_1x_2,, etc.	Highbias	
Try decreasing λ	High bias	Easy to do, just try
Try increasing λ	High variance	As above

Neural networks are also subject to over and underfitting. Small networks are more prone to underfitting, but are computationally cheaper. Large networks are more prone to overfitting but are also more computationally expensive. Use regularization (λ) to address overfitting.

Using a larger network with regularization is usually the best option.

A single hidden layer is usually a good choice, but if you want to try other architectures, using the cross validation method to perform this type of model selection is also an option.

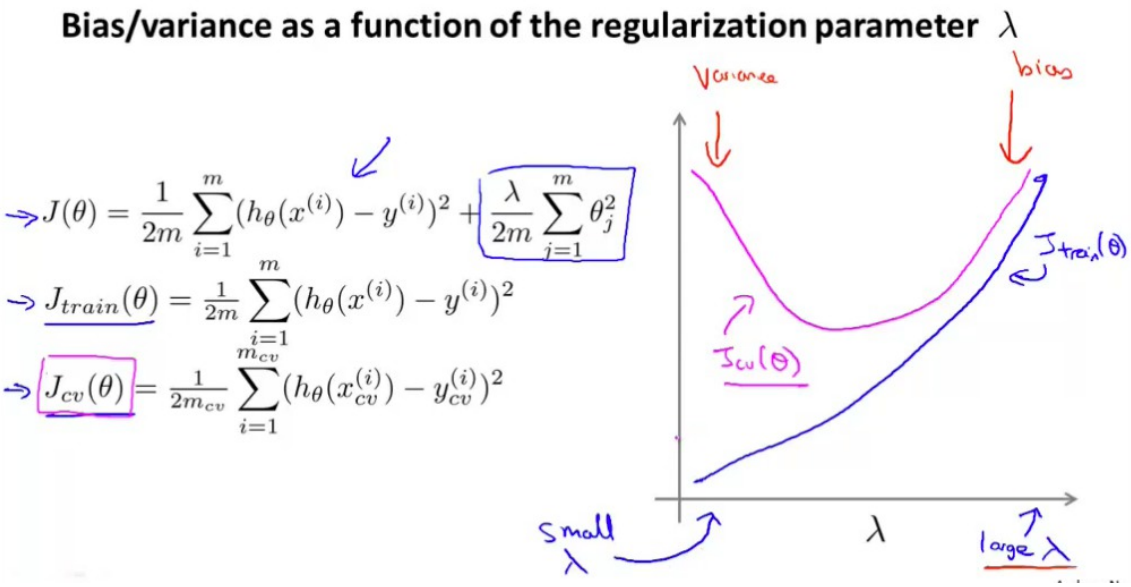


Fig. 6.7 regularizationbiasvar

Missing imagefile file lcurves

Fig. 6.8 lcurves

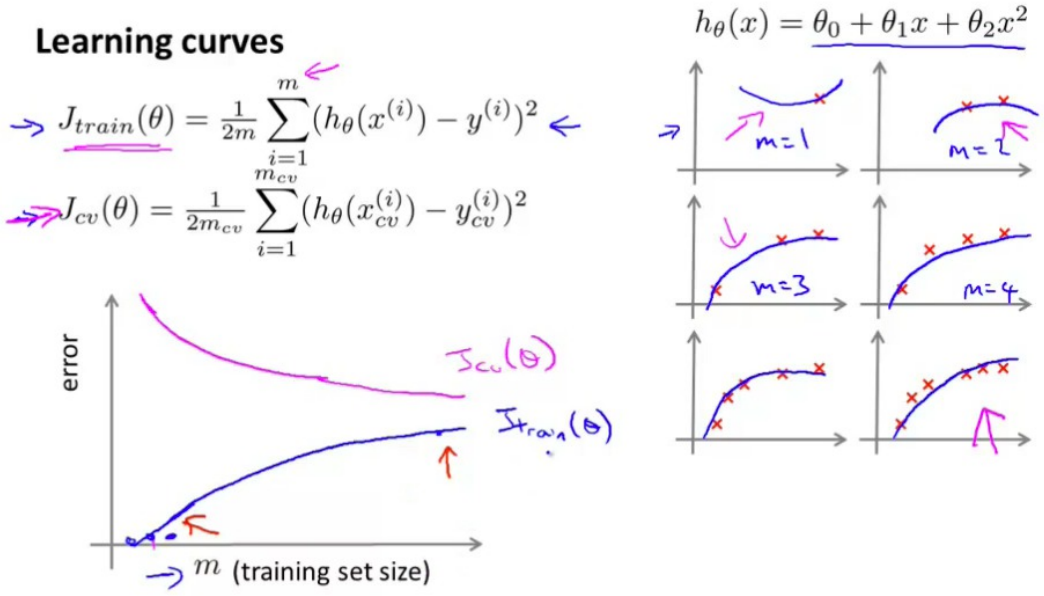


Fig. 6.9 quadlearningcurve

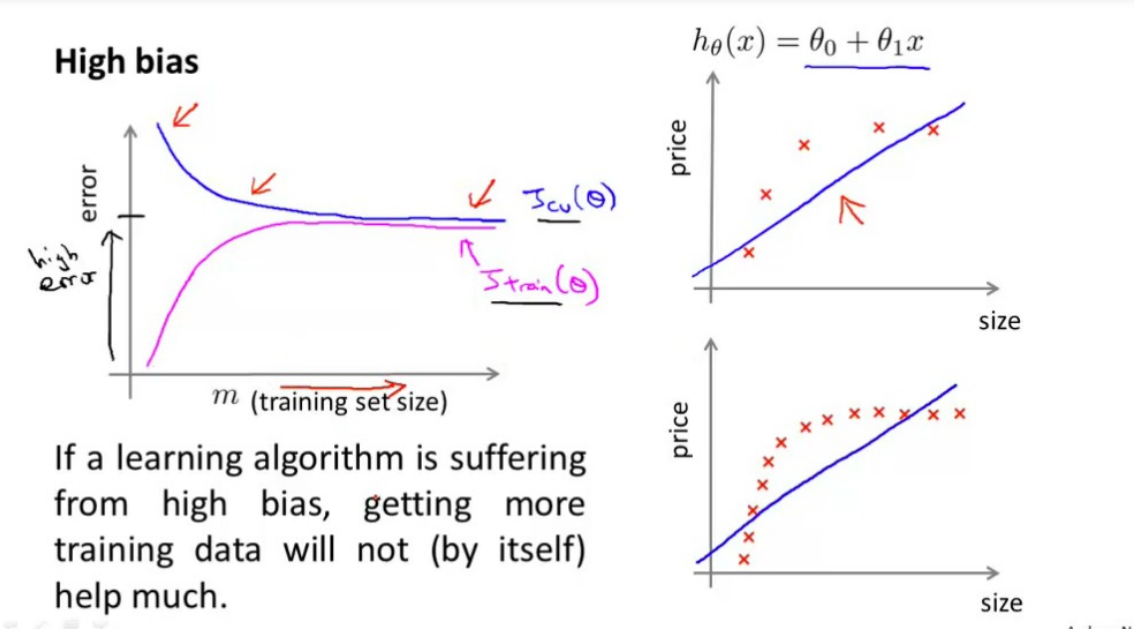


Fig. 6.10 A learning curve that's typical in a high bias situation

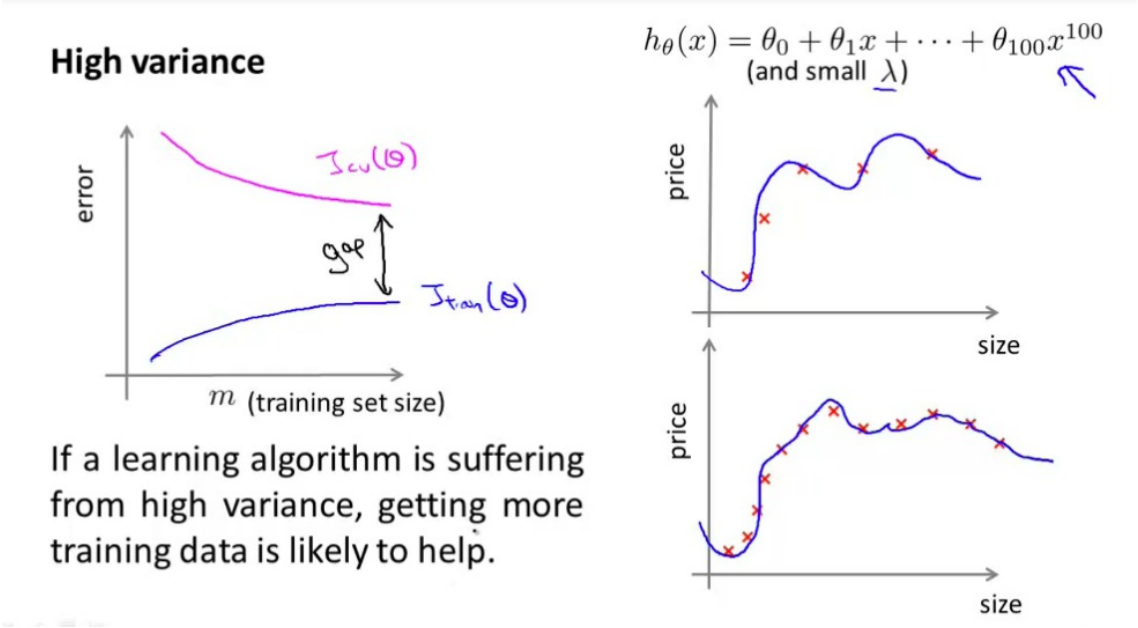


Fig. 6.11 A learning curve that's typical in a high variance situation

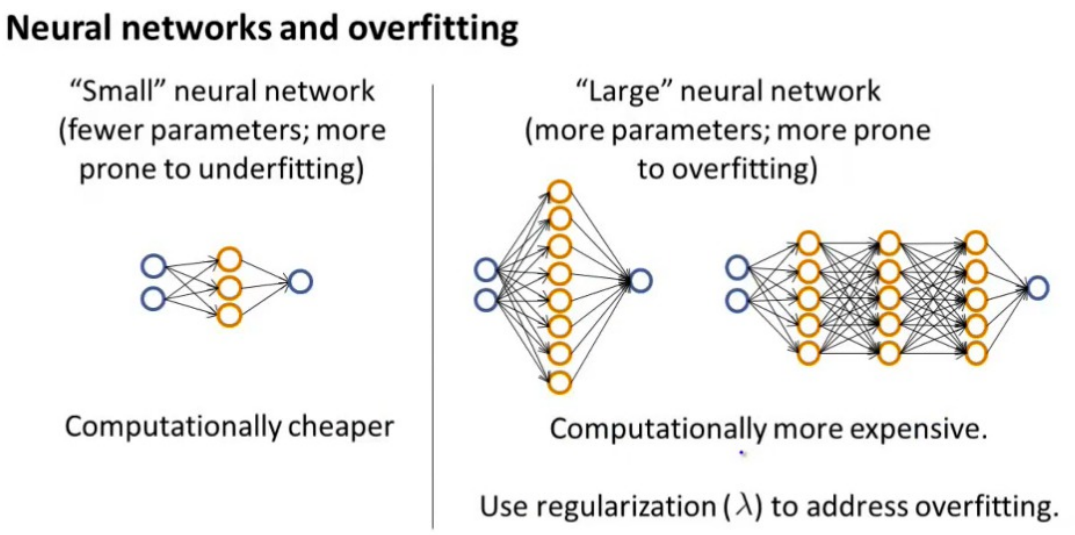


Fig. 6.12 How to protect a neural network against overfitting: Add more hidden nodes and use regularization :-)

7 Machine learning system design

This main issues facing us when we are designing a machine learning system. These videos may seem a bit disjointed, and less mathematical. However these issues will most likely be huge timesavers for real operations.

7.1 Prioritizing what to work on

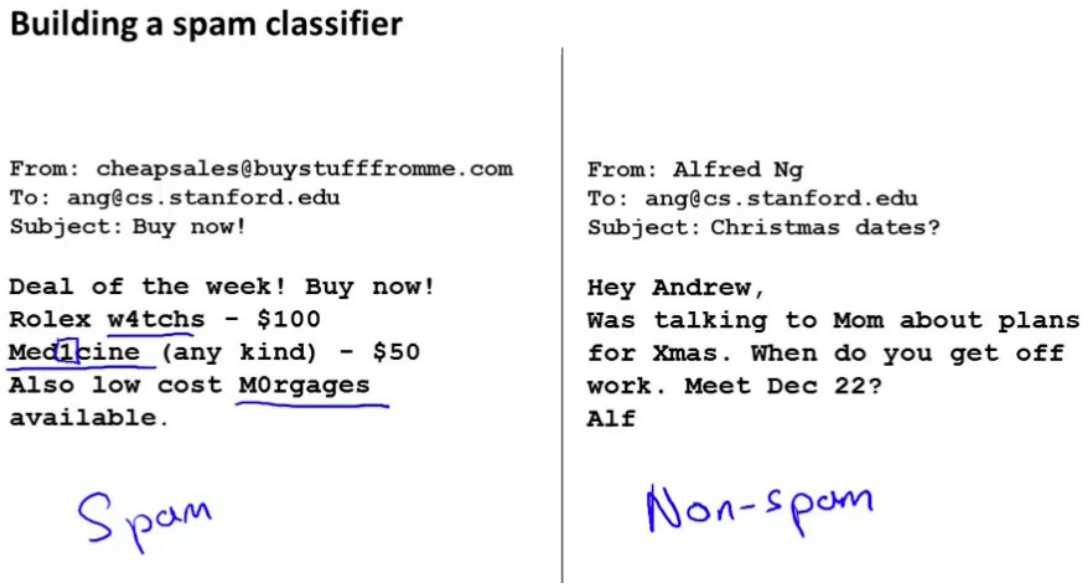


Fig. 7.1 An example of Spam, and an example of non-spam (a.k.a. "ham"

Assume that we are working on a spam classifier. We classify the examples. How do we apply supervised learning? The first thing we must decide is how to represent the features.

One way is to choose 100 words that are indicateive of spam/not spam (e.g. "deal", "buy", "discount" etc.). Whereas email containing a correctly spelled first name may be indicative of non-spam.

Given a piece of email we can then encode it as a feature vector. We can then define a feature to be the word count for the words in our vocabulary. We can even let the word-count be binary, "0" if the word doesn't appear, and "1" if it appears, however many times the word appears. This means that x_j says if the word j is present or not. Typically the number of word is much higher, say ten to fifty thousand automatically picked for high frequency.

Question: How should we spend our time?

- Collect lots of data (e.g. "honeypot" project). This will only be useful sometimes.
- Develop sophisticated fetures basd on email routing information (from the email header). Often spammers try to obscure the origin, use unusual routes etc. The header can be used to develop signals.
- Develop sophisticated features for message body, should "discount" and "discounts" be treated as the same word? How about "deal" and "dealer"? Features about punuctuation?
-

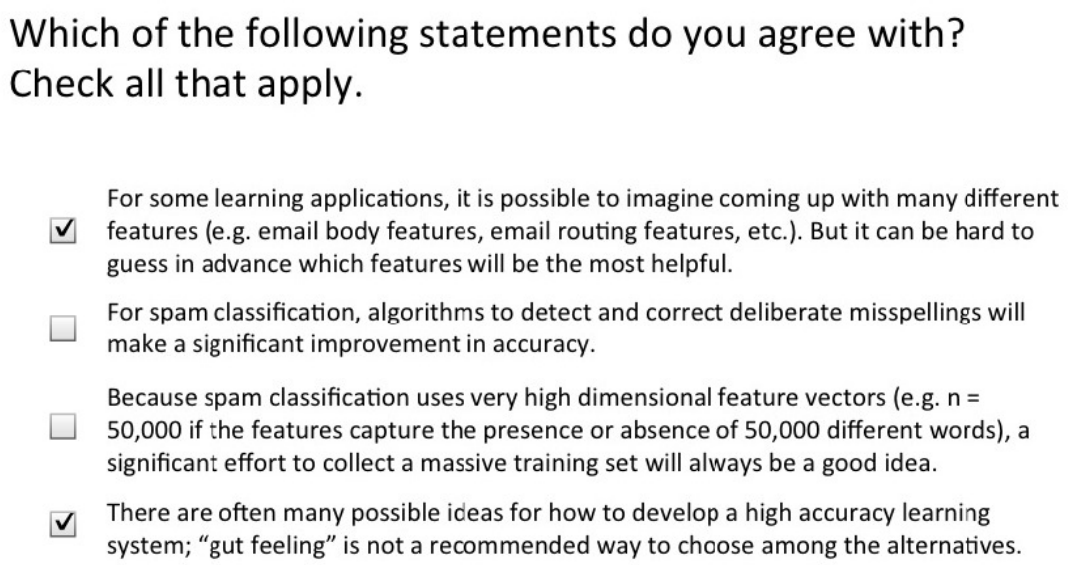


Fig. 7.2 How to select the best way to solve a machine learning problem.

Very often one can brainstorm a lot of interesting things to try to use. However, in general it is not easy to say which possible predictor works the best. Listing up

the options think about them is a good idea. The most commonly used method is to just get up one morning and then just randomly fixate on some method one day and then spend six months pursuing that method is not such a great idea.

Techniques of error analysis to select the options to pursue will help us guide the method selection :-)

7.2 Error analysis

The recommended approach:

- Start with a simple algorithm that you can implement quickly. Implement it and test it on your cross-validation data. At most one day (24 hours). Quick and dirty. Test it on the cross validation data.

- Plot learning curves to decide if more data, more features etc are likely to help. Figure out if the algorithm is suffering from high bias, high variance and so on.

  In general it is very difficult to in advance, in the absence of any evidence, to see where time should be spent. It is often by a very simple implementation and interpreting this that these decision can be made. Think of this as avoiding *premature optimization*.

  We should let evidence rather than gut feeling guide where we should spend our effort.

- Error analysis: Manually examine the examples (in the cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

  This process will often inspire us to design new features or improvements.

As an example of error analysis. Ng is quoting an example where you have 500 crossvalidation samples. The algorithm misclassifies 100 emails. *Manually examine* each of the errors and categorize them based on what type of mail it is, and what cues (features) you think would have helped the algorithm to classify them correctly.

Example: Pharma, replica, steal passwords (phishing). Count them up, and classify for all of these classes. Then concentrate on the most important stuff (e.g. the ones with password stealing). Some possible features are deliberate misspeppling (m0rgage, med1cine), unusual routing, unusual punctuation etc. Look for strong signals, and concentrate on those. "hillclimbing" through a manual process.

What we want to figure out is what is the most difficult to classify and concentrate onthose.

The importance of numerical evaluation: Should discount/discounts/discountes be treated as the same word? We can use *stemming* sofware, e.g. the *Porter stemmer*. But will it help? It can hurt since there are false positives (universe = university). Error analysis may not be h elpful to deciding. The only solution is to try it to see if it works.

In order to do this test it is very imprtant to have a numerical way, so we can run with and without stemming (*A/B testing*). For this example the single row number cross validation is a good test, but in other cases it is not and it is more difficult to figure out what to do.

Another case, should we distinguish between upper and lower case? Do the test, do the math, decide rationally. If we need to do manual evaluation of everything it is harder to try out new ideas than if we can automatically do error analysis on the cross validation set.

To wrap up: Quick and dirty. Usually people spend too much time on this implementation. Don't worry about being too dirty or too quick. Then use the tool to decide where to spend the time on further development. If we have a single row number evaluation metric it's easy to evolve the algorithm.

7.3 Error metrics for skewed classes

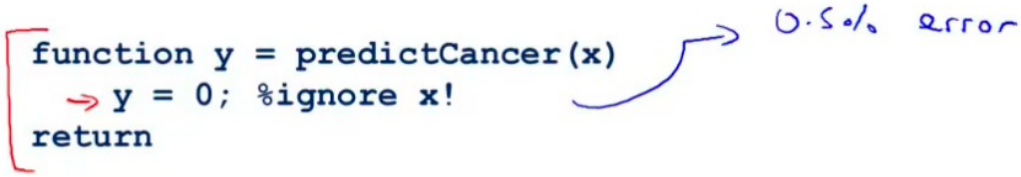


Fig. 7.3 cancer predictor

We train a logistic regression model to 1 percent error, 99 percent correct diagnosis. However, only 0.5 percent of the patients really have cancer. This means that a predictor that always predicts no cancer will only have .5 percent error, and that makes our one percent predictor looking not so hot.

This is the case of *skewed classes* where one of the classes is much more common than the other. The problem with using classification accuracy is that it's hard to use just classification accuracy to determine the efficiency of the classifier.

We could use another metric is the *precision/recall* classifier. where y=1 in presence of a rare class that we want to detect. The actual class is going to zero or

one. Our learning algorithm will predit som e value for the class, and our learning algorithm will predict either one or zero. We get both true and false positives and negatives. one way of prediction the performance is to calculate precision and recall:

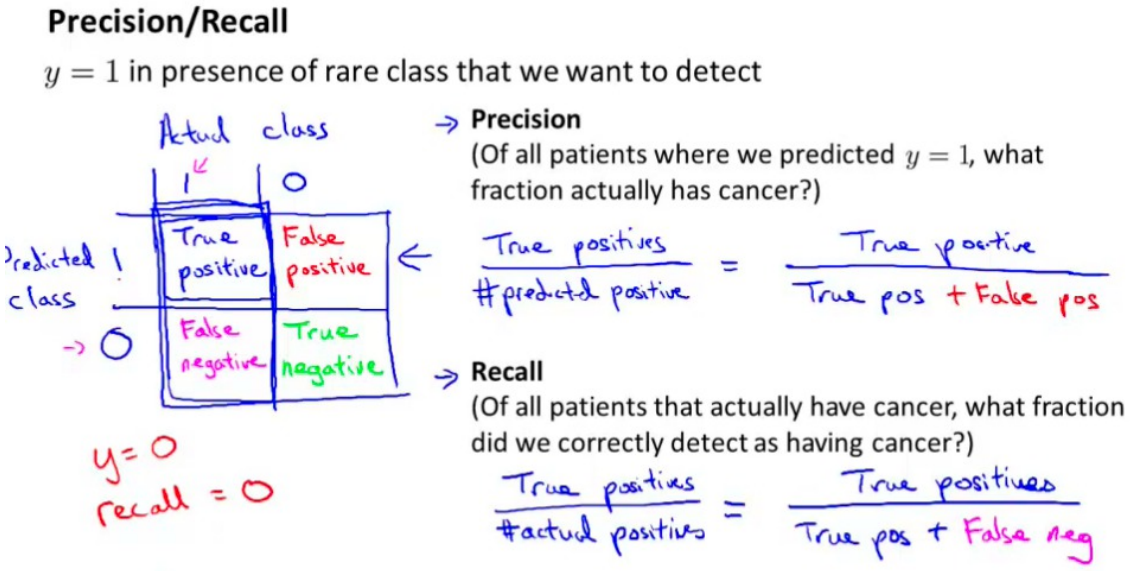


Fig. 7.4 Precision/Recall

- Precision. The number of true positives divided by the number of predicted positives (true positives + false positives).
- Recall. The fraction of patients that actually have cancer, what fraction did we correctly detect as having cancer? True positives/(actual positives).

More generally it is not possible for an algorithm to cheat by doing some simple thing by setting values to a constant. We are much more certain that a classifier with high precision and recall is a good a classifier.

7.4 Trading off precision and recall

In some cases we wish to trade off between precision and recall and here we will learn a bit more about how to do that. First a bunch of definitions:

- Logistic regression $0 \leq h_\theta(x) \leq 1$.

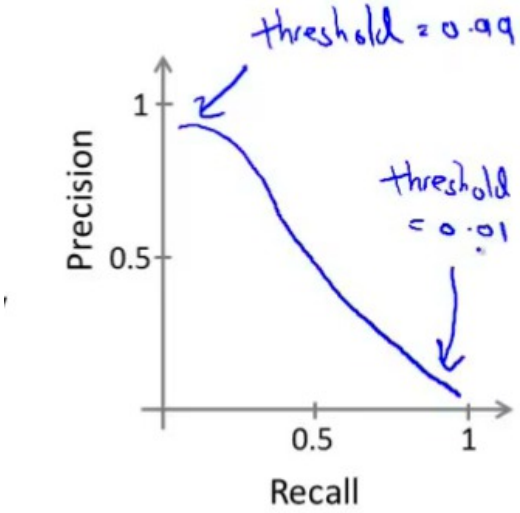


Fig. 7.5 A tradoff between precision and recall

- Predict 1 if$h_\theta(x) \geq 0.5$
- Predict 0 if$h_\theta(x) \leq 0.5$
- Precision:

$$\frac{\text{True positives}}{\text{No of predicted positive}}$$

- Recall

$$\frac{\text{True positives}}{\text{No of actual positive}}$$

We'll continue with the cancer classification (1 = cancer). What if we want to predict y=1 (cancer) only if we are very confident. One way to do this is to modify the algorithm by setting the threshold value at .7 not .5. If we do this then we will predict cancer only when we are more confident, so precision will go up. On the other hand, there may now be more cases that are actually cancer but we don't recognize as such, so recall may go down. We can even set the threshold to be .9, which means that we'll get eve higher precision, but even less recall.

Suppose instead that we wish to make really certain that we get as many as possible patients with cancer. so we want to try hard to avoid false negatives, then we can do that too by tweaking the threshold, this time by decreasing it. Perhaps .3 (30 percent).

Higher precision, increase threshold, but in general the precision/recall curve looks different for each classifier.

	Precision(P)	Recall(R)	Average	F_1 score
Algorithm 1	0.5	0.4	0.45	0.444
Algorithm 2	0.7	0.1	0.4	0.175
Algorithm 3	0.02	1.0	0.51	0.0392

Fig. 7.6 Various ways of comparing precision/recall

A question is. Can we choose the precision/recall parameters automatically? The F_1 score (F score) helps us doing that. It is really useful to have a single real number to indicate how good our classifier is, but with precision/recall we have actually lost that and instead gotten two numbers. One thing to try is the average of precision and recall, but that is in fact not a very good option since it gives classifiers that just give out constants really good scores (illustrated by algorithm 3 in figure ??). Instead, there is a formula called "F_1 score" that is much more useful.

$$F_1\text{score} = \frac{P \cdot R}{P + R}$$

It takes the product, so if both precision and recall are zero, the result is zero. If both precision and recall is one (perfect), then the F score would be one.

There are many possible ways of combining precision and recall, but traditionally the F-score has been used in machine learning. Also the name doesn't mean anything (the F has no function, so to speak).

Also, it's a good idea to do the scoring on the cross-validation data, not on the training set. Testing various thresholds based on f-scores on the cross-validation set would also be a reasonable way to selecdt a classifier.

7.5 Data for machine learning

How much data to train on? Ng cautions against always collecting a lot of data in every case. In some cases it is not necessary, but on some cases it is not. In this section we'll discuss this some more.

Banko and Brill in 2001 evaluated different algorithms (perceptron (logistic reression), winnow, memory-based and naive Bayes) to separate between confusable words in english sentences. The exact details of the algorithms are not that important for this example. They then tried the algorithm on various training set sizes, and the results can be viewed in figure ??. The trend is very clear. Initially all of the algorithms give really similar performance. Also the performance monotonically increase with larger training set. It's also true that an otherwise

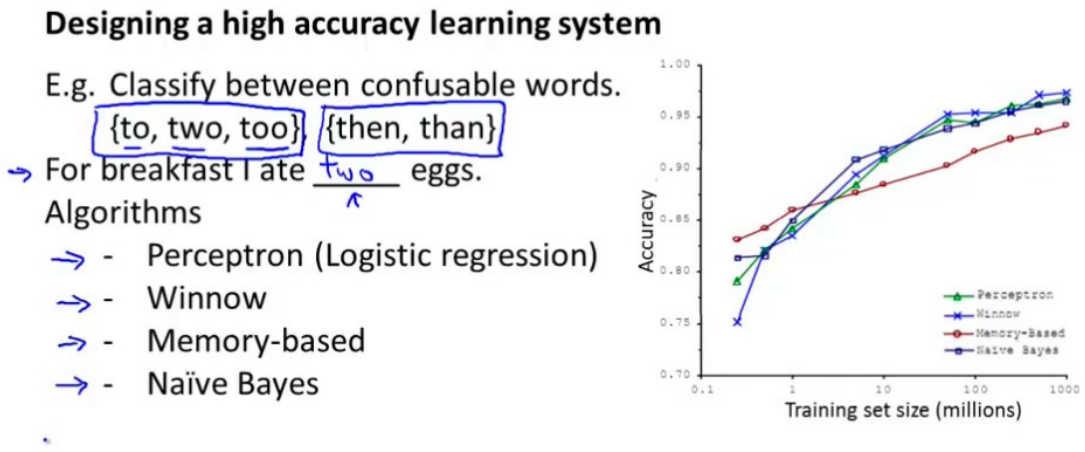


Fig. 7.7 Banko an Brill's result: There is no data like more data. More data will often turn a mediocre machine learning algorithm into a decent one.

inferior algorithm will give very good results given a larger training set. This finding has given cause to a saying in machine learning:

> *It's not who has the best algorithm that wins. It's who has the most data.*

But when is this true and when is it not true?

7.5.1 A rationale for large data

Assume a feature "x" in an n+1 dimensional real space, and that gives enough information to predict an "y" accurately. Example: "For brekfast I ate XXX eggs". Counterexample: Predict housing price from only size in square feet and no other features.

Useful test: Given the input x can a human expert confidently predict y? If no, then more data won't help much.

Use a learning algorithm with many parameters (e.g. logistic regression / linear regression with many features or neural network with many hidden units). This will be algorithms with a low bias. If we train this algorithm on a large sample, so the training error will be small. If the training set is much larger than the number of parameter we are likely to not overfit, so the training set will hopefully be similar to the test error.

We are addressing the bias problem with a low-bias algorithm, and by using a large learning set we get a low variance (hopefully).

- Can a human look at the data and confidently predict things?
- Can we get a large learning set?

Can we do both this will often give us a high performing algorithm.

8 Support vector machines

8.1 Optimization objective

SVM sometimes gives a cleaner way of learning nonlinear functions than the methods we have seen so far. This is the last algorithm for supervised learning we will spend significant time on. The support vector machine is in its essence a modified logistic regression method.

Recall that $h_\theta = \frac{1}{1+e^{-\theta^T x}}$, for simplicity we will let $h_\theta = g(z)$ where $z = \theta^T x$. If $y = 1$ we want $h_x \approx 1$, and $\theta^T x \gg 0$. The reason is the output of logistic regression becomes close to one. Conversely we wish If $y = 0$ we want $h_x \approx 1$, and $\theta^T x \ll 0$. The cost function for logistic regression is designed so that each pair (x, y) in the training set contributes (ignoring 1/m).

$$-(y \log h_\theta(x) + (1-y) \log(1-h_\theta(x))) = -y \log \frac{1}{1+e^{-\theta^T x}} - (1-y) \log(1 - \frac{1}{1+e^{\theta^T x}})$$

Missing imagefile file svnerrorfunction

Fig. 8.1 The error function for a support vector machine

Now, if y=1 only the first term (on either side of the equality) counts. That means that the function shown in ?? is what drives (one) part of the error. We will now modify that to create the support vector machine basis .-) The modification is to make a pickwise linear function (drawn in purple in the figure) that is zero from z values of 0 and upwards, and follows a linear approximation if the (component) of the error function for logistic regression downwards to infinity. The slope of the straight line is btw not very important. The fact that it's linear gives us increased computation efficiency.

Missing imagefile file svmerrorfunction2

Fig. 8.2 The error function for a support vector machine

For the other part of the error function we use another function that is basically the mirror of the first. We call these functions "$\text{cost}_1(z)$" and "$\text{cost}_0(z)$". Armed with this we can now build the support vector machine.

The error function for logistic regression is:

$$\min_\theta \frac{1}{m}\left[-(y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For the support vector machine it is:

$$\min_\theta \frac{1}{m}\left[-(y^{(i)}\text{cost}_1(\theta^T x^{(i)})) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{1}{2m}\sum_{j=1}^{n}\theta_j^2$$

And this defines our minimization problem. By convention the formula above is reparameterized a bit when used in SVMs, the $1/m$ term is removed, since it is just a constant. Furthermore for logistic regression the objective function is parameterized as $A + \lambda B$ by modifying λ we can trade the number of parameters with precision. For SVMs we will instead optimize a function parameterized by $CA + B$. This is just a different way of controlling the tradeoff.

$$C\min_\theta \left[-(y^{(i)}\text{cost}_1(\theta^T x^{(i)})) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

That gives us our overall cost function for a support vector machine. In a sense one can say that $C = 1/\lambda$, or at least the role it plays is similar, using either one will not change the optimal value that we will find.

Finally, unlike logistic regression the SVM doesn't output a probability. Instead it just makes a prediction:

$$h_\theta(x) = \begin{cases} 1 & \text{if} \quad \theta^T x \geq 9 \\ 0 & \text{otherwise} \end{cases}$$

8.2 Large margin intuition

SVNs are sometimes called *large margin classifiers*. A logistic classifier switches its prediction at zero (when $\theta^T x \geq 0$ becomes or stops being true). An SVM's cost function components on the other hand requires $\theta^T x \geq 1$ to predict and $\theta^T x \leq -1$.

This builds an extra safety margin into the SVN machine. Let's consider a case where the constant C is very large, the minimation objective will be highly motivated to find a value where the cost functions (not the regularization terms) is very close to zero. Now, what will it take to make that happen?

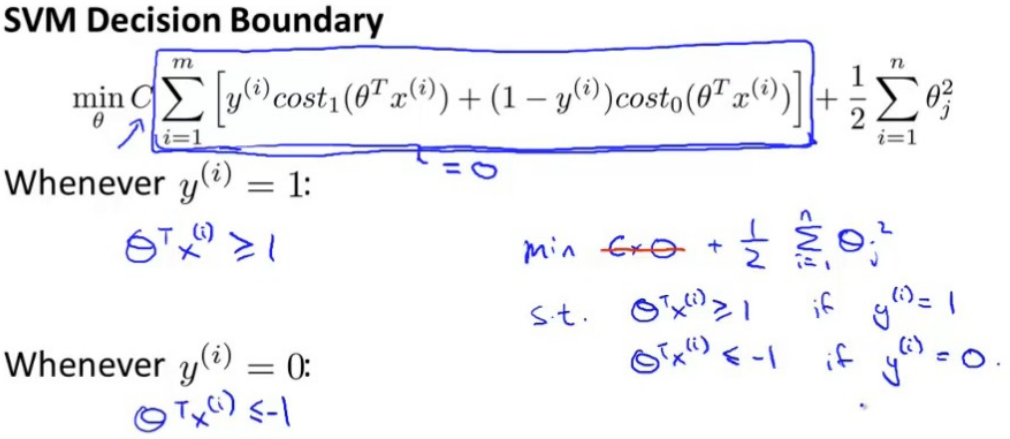


Fig. 8.3 svmdecisionboundary

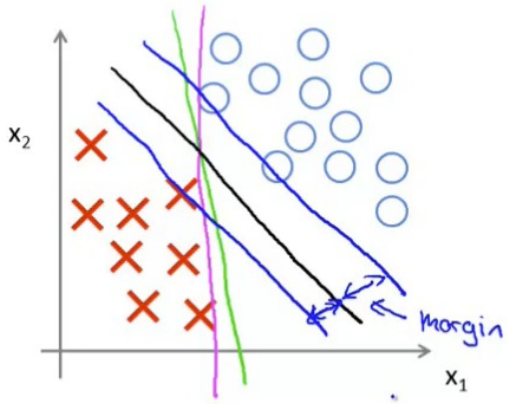


Fig. 8.4 A support vector machine with a linear kernel

For an SVN the decision boundary will be linearly separably. The decision boundary for the SVN has a large *margin* between the classes (drawn in blue in ??). The optimization problem we've just seen actually creates this *large margin classifier*. The SVN classifier will try to separate the positive and negative examples with as big a margin as possible.

SVM is actually more sophisticated than it seems. If the C is very large, then the classifier will be very sensitive to outliers, but if a more reasonable C is used,

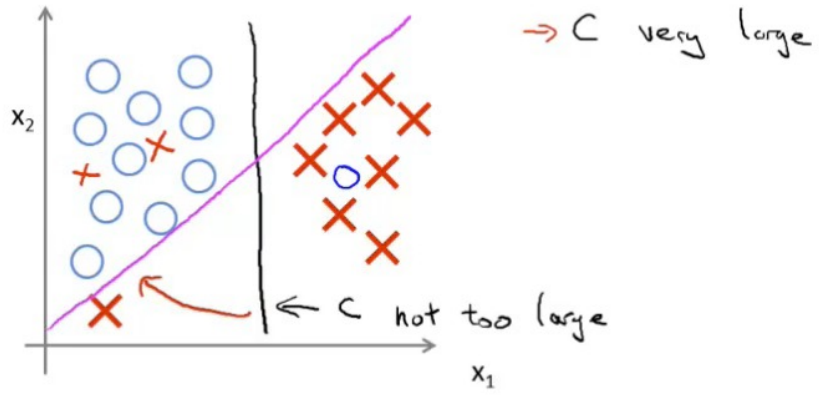


Fig. 8.5 SVM and outliers, how sensitive should one be to them?

then it can be made less sensitive to outliers.

8.3 The mathematics behind large margin classification

We use inner products $u^T v$ and the norm of a vector $||u|| = \sqrt{u_1^2 + u_1^2}$. The inner products are positive is the angle between the vectors is less than ninety degrees, and negative if it is larger (you know what I mean).

The decision boundary (for an appropriate value of C I suppose) is:

$$\min_\theta \frac{1}{2}\sum_{j=1}^{n} \theta_j^2 = \frac{1}{2}(\theta_1^2 + \theta_2^2) = \frac{1}{2}(\sqrt{(\theta_1^2 + \theta_2^2)^2} = \frac{1}{2}||\theta||^2$$

Rmz: *This doesn't look right?*

The parameter vector θ will be at right angle with the decision boundary. The projections of the sample set down onto the parameter vector. The mechanics of it is that by having the decision boundary as much as possible at right angles to the objects, that means that the projection down onto the θ can be large. Since the decision boundary is defined by $p^{(i)}||\theta||$ this means that θ can be small, and since our penalty function contains the sum $\sum_{j=1}^{n} \theta_j^2 = ||\theta||$, that means that we can get a decision with a minimally small θ .

Rmz: *That at least made sense when I wrote it.*

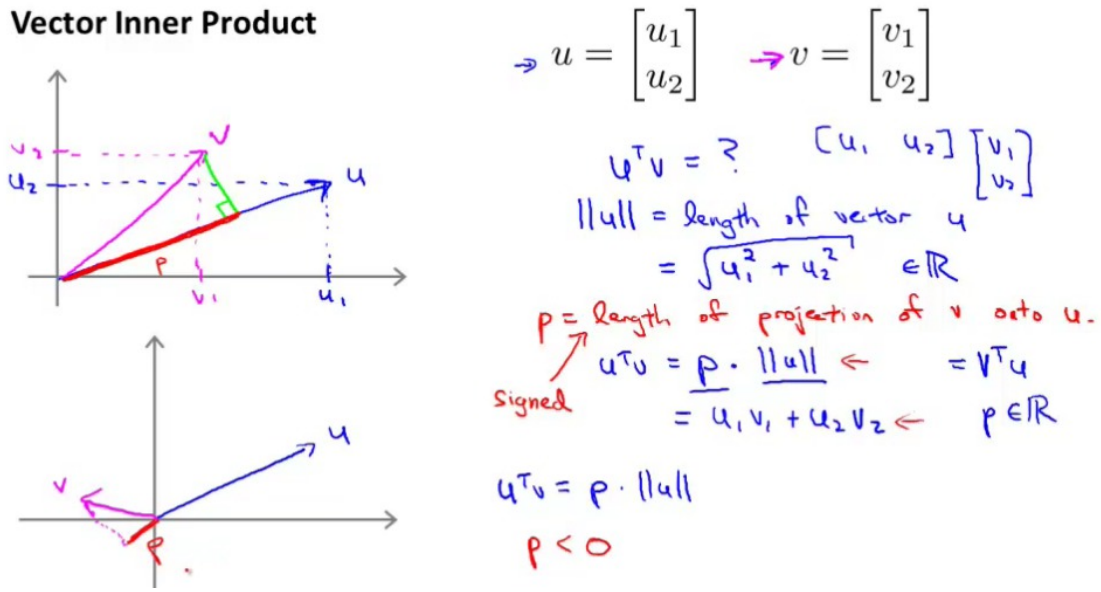


Fig. 8.6 The vedto rinner product

8.4 Kernels I

We'll use kernels to produce complex non-linear classifiers. We can use complex polynomial features. A question is how to find better features than polynomial terms? One idea is to define new features using *kernels*. The idea is to start with a bunch of *landmarks* $l^{(1)}, l^{(2)}, l^{(3)}$. We will now define the features to be some similarity between the landmark and the sample:

$$f_1 = \text{similarity}(x, l^{(1)}) = \exp\left(-\frac{||x - l^{(1)}||^2}{2\sigma^2}\right)$$

The similarity function is called a *kernel* and this particular kernel is called an *Gaussian kernel*. The kernel is sometimes denoted as $k(x, l^{(i)})$ (ignoring the intercept term).

For a Gaussian kernel, if the x is close to the landmark, then the euclidian distance in the kernel wil be close to zero, so the feature will be approximately 1. Conversely, if the x is far from the landmark, the feature will be close to zero.

The feature measures how close something is to the landmark. We can modify the σ to make the kernel "sharper" i.e. give a larger penalty for being large distance from the landmarks. A larger σ gives less penalty, and a smaller σ gives larger penalty (the probability distributions has a maximum at 1 and w width proportional to σ.

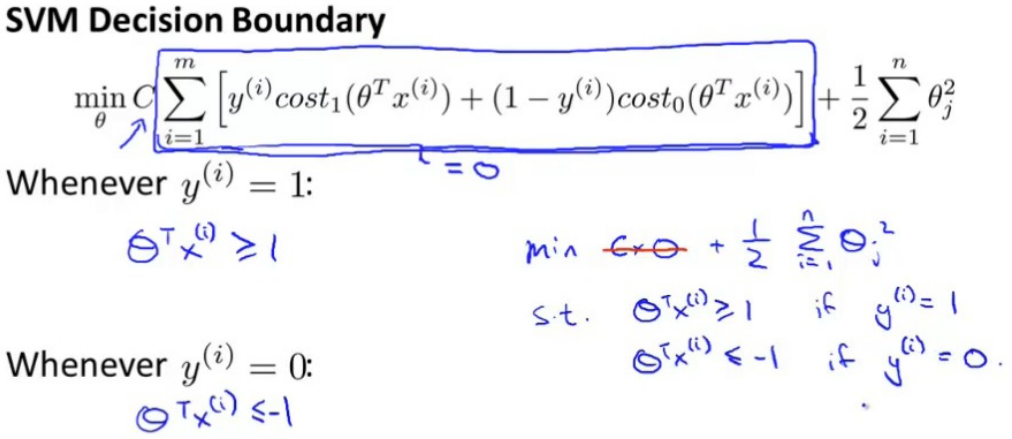


Fig. 8.7 The decision boundary for a support vector machine

Using this Gaussian kernel we can learn pretty complex decision boundaries.

How do we choose landmarks, and which other similarity functions can we use? In the next section we'll learn about that, and how we can combine it in a support vector machine that can learn complex decision boundaries.

8.5 Kernels II

We'll se how to use kernels and how to do bias/variance tradeoffs.

8.5.1 How to find landmarks?

In practice, we have some positive and negative examples. For every training example we have, we are going to put landmarks exactly at the location of the training examples. This gives us m landmarks (one for each training examples). This will give us a classifier that measures how close things are to our training set.

Rmz: *For a class?*

For each sample we will then get a feature vector. For a training example $(x^{(i)}, x^{(i)})$ we will get a feature vector:

$$\begin{bmatrix} f_1^{(i)} \\ f_1^{(i)} \\ \dots \\ f_m^{(i)} \end{bmatrix} = \begin{bmatrix} \text{sim}(x^{(i)}, l_m^{(1)} \\ \text{sim}(x^{(i)}, l_m^{(2)} \\ \vdots \\ \text{sim}(x^{(i)}, l_m^{(m)} \end{bmatrix}$$

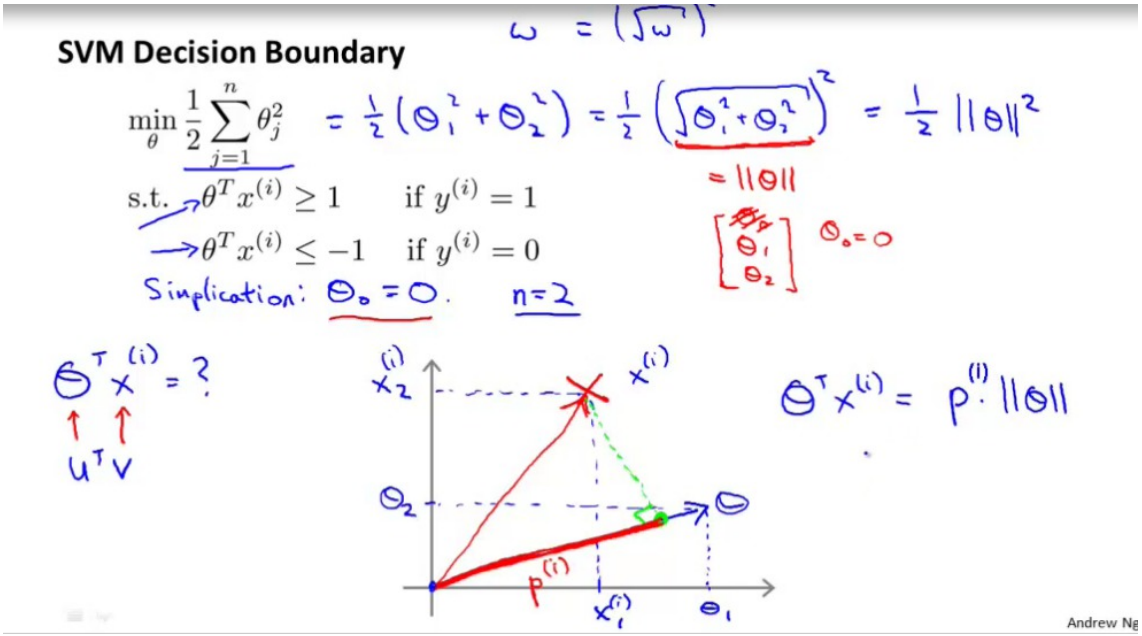


Fig. 8.8 The decision boundary for a support vector machine

We can now plug this into the SVM. We will be working on a feature vector that gives us a similarity to the individual points in the training set. We will then be useing a minimization functinon

Not:

$$C\min_\theta \left[-(y^{(i)}\text{cost}_1(\theta^T x^{(i)})) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

But:

$$C\min_\theta \left[-(y^{(i)}\text{cost}_1(\theta^T f^{(i)})) + (1-y^{(i)})\text{cost}_0(\theta^T f^{(i)})\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

In this case it is also true that $m = n$. There is one mathematical aside here. We see trivially that

$$\sum_{j=1}^{n}\theta_j^2 = \theta^T\theta$$

However, in most implementations of SVMs the actual term used is:

$$\theta^T M\theta$$

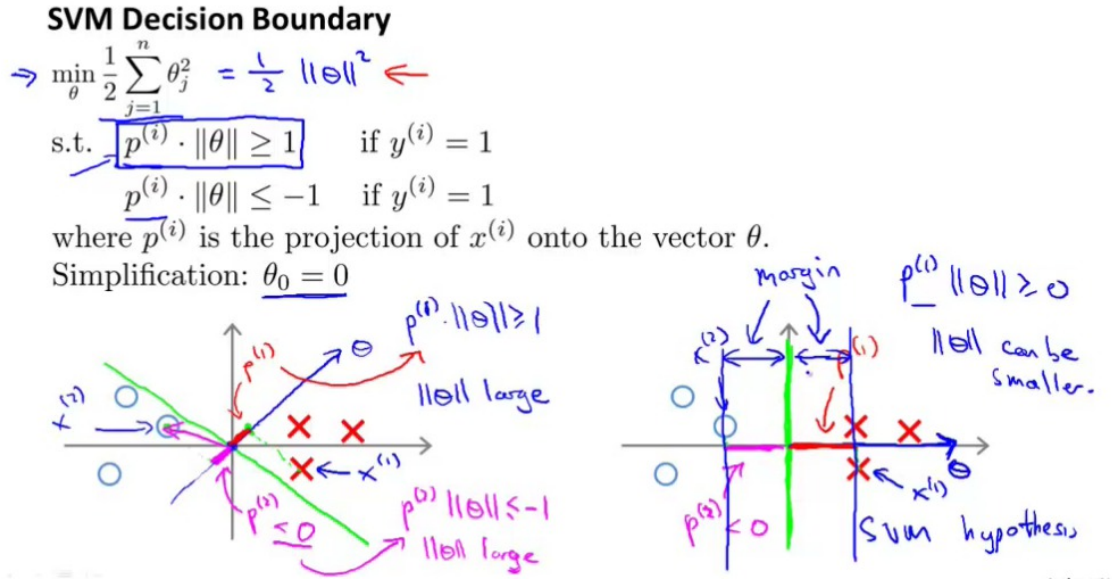


Fig. 8.9 The decision boundary for a support vector machine

This is a slightly different distance metric where M is dependent on the kernel we use. A rescaled version of the parameter vector θ. The reason this is done is that we can use much bigger training sets. If $m = 10000$ which means that θ becomes really big. Solving for all of these parameters becomes expensive. It does change the optimziation criterion as well, but that doesn't matter very much.

We can in fact use the kernel trick for other classifiers as well, including logistic regression but the efficiency we can get from the SVM doesn't translate to other classifiers. SVNs and kernels are a match made in heaven (math made in heaven? :-)

It'a recommended to not implement the SVN machine yourself, it's better to use a library someone else has sweated over first :-) (same as inverting a matrix or computing a square root).

8.5.2 Bias and variance management

$C \approx \frac{1}{\lambda}$ large C, lower *bias* high *variance* (analogous with small λ).

Small C, higher bias, low variance (analogous with large λ). Small values of C more prone to *overfitting*, large values of C more prone to *underfitting*.

The other parameter we need to choose is σ^2:

Large σ^2 implies that features f_i varies more smoothly. Higher bias, lower variance. Wide Gaussian. More prone to underfitting.

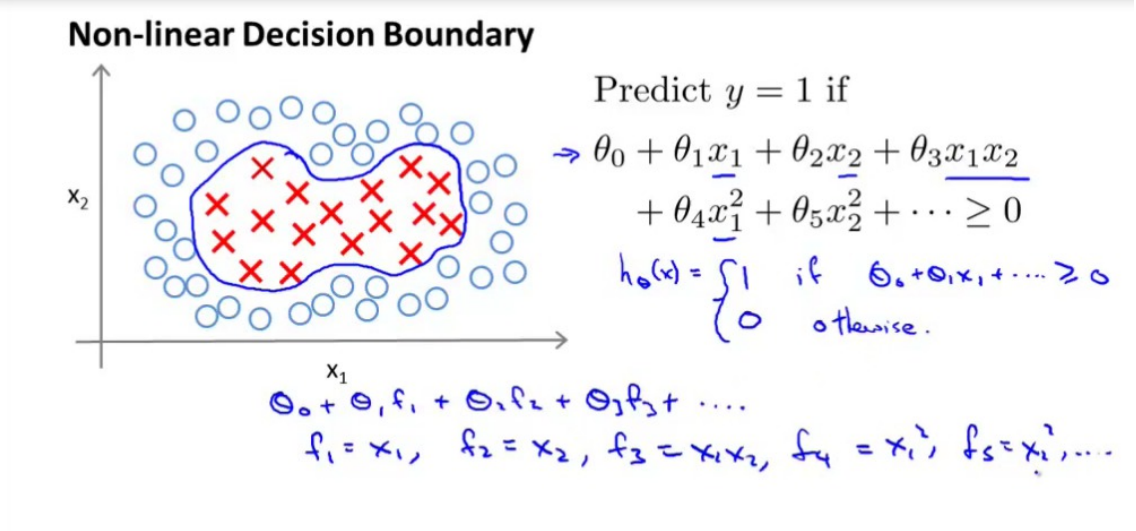


Fig. 8.10 A nonlinear boundary using a nonlinear kernel

Small σ^2 implies that features f_i varies more abruptly. Lower bias, higher variance. narrow Gaussian. More prone to overfitting.

8.6 Using an SVN

How to practically solve SVN optimization software. In essence: Use a library (liblinear, libsvm), there are many other for many languages.

However, there are some things that needs to be done:

- Choose a a parameter C.
- Choose a kernel.
  - One choice is no kernel, a.k.a. *linear kernel*: "predict y=1" if $\theta^T \geq 0$. ("an SVM with a linear kernel"). This is the version of SVM that just gives you an ordinary linar classifiser. Many libraries can do this. It is a reasonable choice if you have a huge number of features and a small number of items in your training set.
  - Another choice is the *Gaussian kernel*:

$$f_i = \exp -\frac{||x - l^{(i)}||}{2\sigma^2}$$

where $l^{(i)} = x^{(i)}$. In this case we need to choose σ^2 as a parameter.

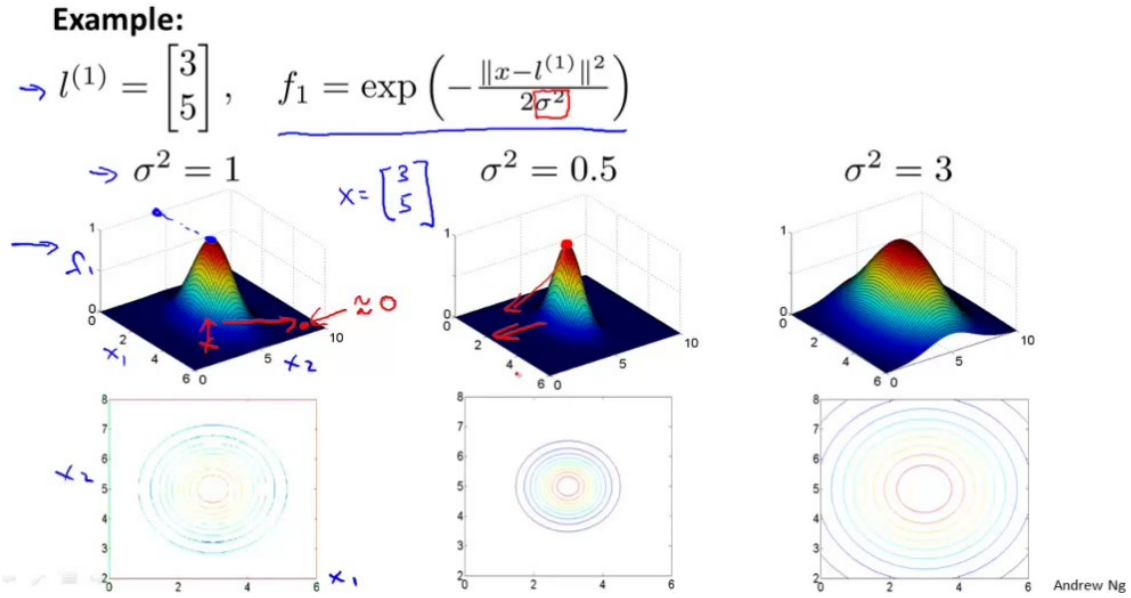


Fig. 8.11 How does a Gaussian look with some different parameters?

> A Gaussian kernel is reasonable if $x \in \mathbf{R}^n$, where n is amall and m (the number of training samples) is large. E.g. a two dimensional training set with a nonlinear boundary.

If you choose to use a Gaussian kernel, you need to provide the kernel. Something of this nature:

```
function f  = kernel(x1, 2)
    f = exp(- (norm(x1 - x2)^2 / (s*sigma^2)))
return
```

x1 is a test example, x2 is a landmark. The return value f is a real number. It's important to perform feature scaling before using the Gaussian kernel. If the features take on very different values, the difference $x_1 - l_1$ might be much smaller than $x_2 - l_2$ which would mean that the x_1 parameters wouldn't be given any attention. Some SVM implementations provides both the linear and Gaussian kernels.

Not all similarity functions make valid kernels. They actually need to satisfy a condition called *Mercer's theorem* to make sure that the optimizations run correctly and do not diverge. This is actually a design decision that was done when SVMs were specified. Some other kernels are:

Rmz: *Very clever Â·-)*

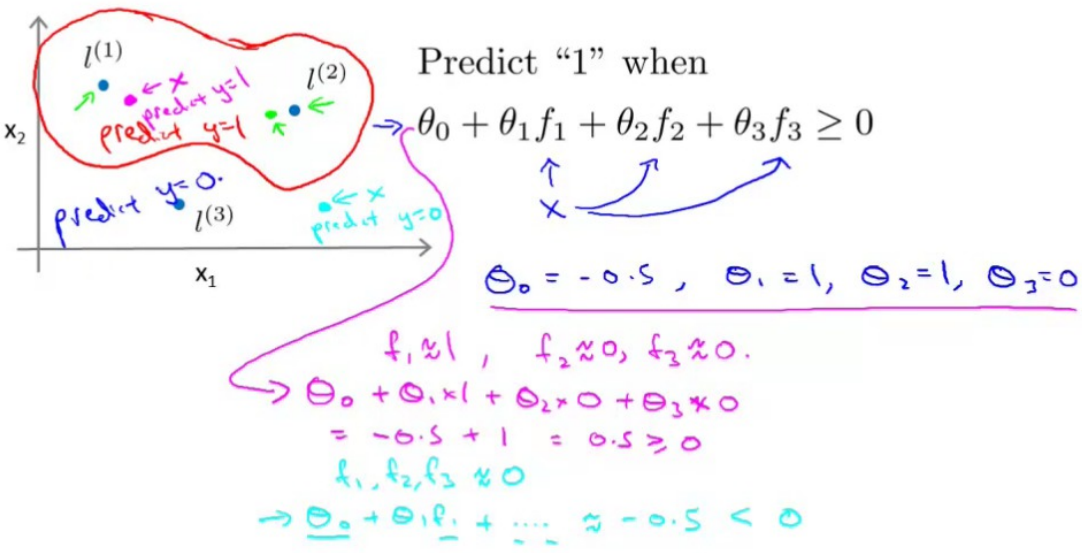


Fig. 8.12 Using Gaussian to learn complex decision boundaries

- *The polynomial kernel*

  $$k(x, l) = (x^T l)^2$$

  This is a slightly unusual kernel, but it is used sometimes. Some other examples are $(x^T l)^3$, $(x^T l+1)^3$, $(x^T l+5)^4$. There are a couple of parameters: The more general form is $(x^T l + \text{constant})^{\text{degree}}$.

  The polynomial kernel almost always performs worse than the Gaussian kernel. Always use it when all the parameters are strictly non-negative, so there can never be negative similarities

- Other kernels (more esoteric): String kernel (text input data), chi-square kernel, histogram intersection kernel etc.

8.6.1 Multi-class classification with SVM

Many svm packages already have multiclass classification. Otherwise one can use the *one-vs.-all method*. Make k svms, one for each class, pick the one with the largest $(\theta^{(i)})^T x$.

8.6.2 Logistic regression v.s. SVMs

If n is number of features and m number of training examples.

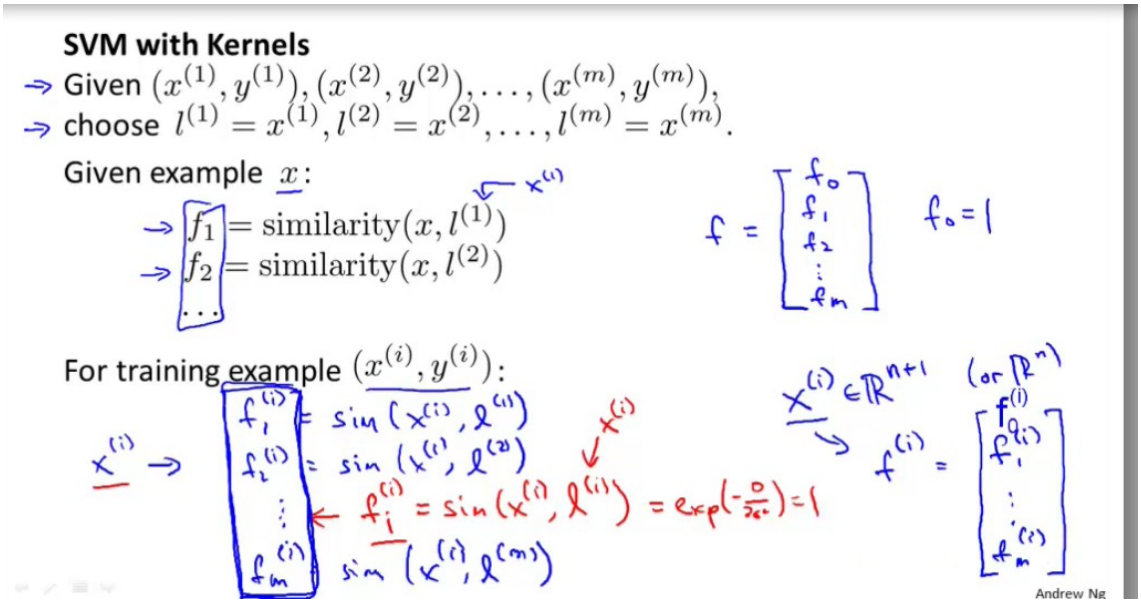


Fig. 8.13 Support vector machines with kernels

If $n \geq m$, e.g. in a text classificaiton problem then use logistic regression or SVM without a kernel. A linear function will do fine, and we don't have enough data to fit a fancy kernel.

if n is in the range from 1-10000 and m is in the range 10-10000 (but not a million), then use SVM with the Gaussian kernel.

If n is small but m is large (50K, a million or large), then create more features, then use logistic regression without a kernel.

Logistic regression and SVM without a kernel will usually do pretty similar things and get similar performance. If one works, then the other will probably work pretty well too.

With about ten thousand to fifty thousand learning examples SVMs with gaussian kernels really shine.

Neural networks are likely to work well for most of these settings, but may be slower to train. SVMs can be very much faster.

Also the SVM packages have a convex optimization problem, so we will always find the global optimum.

These guidelines are a bit vague, but that's ok. The algorithm does matter, but error analysis, the size of the data set, getting the right features and those things often mattes more. SVMs are still conidered to be very efficient. With logistic regression SVM, neural networks are basically the state of the art of machine learning systems for a wide range of problems.

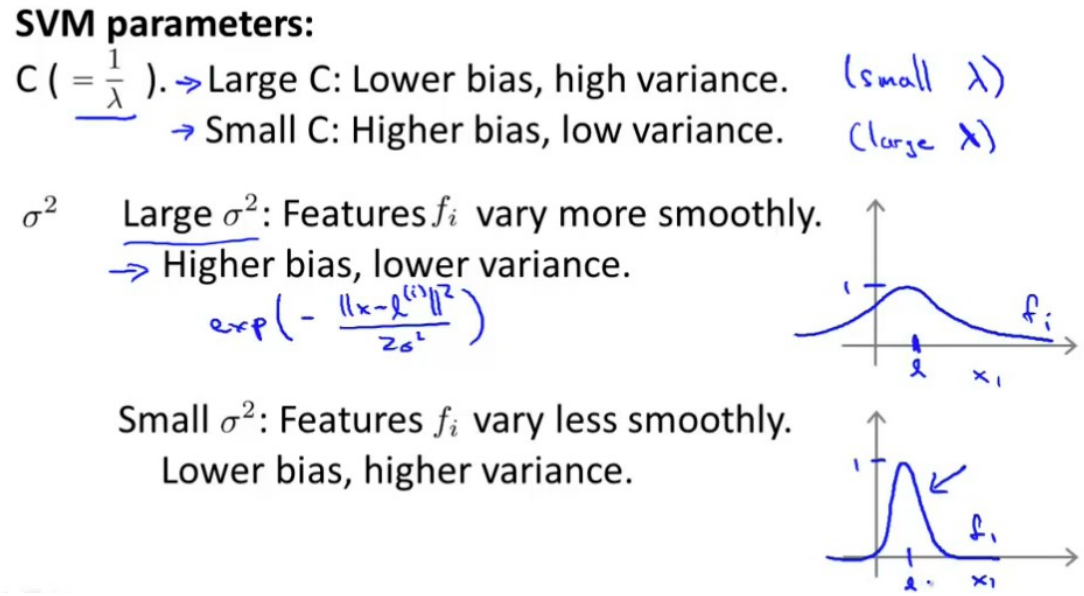


Fig. 8.14 Parameters for the SVN algorithm

9 Unsupervised learning: Clustering

The problem is that we are given data without any labels to it, so our training set is just a bunch of data. In unsupervised learning we give this type of data to an algorithm and asks it to find some structure. One type of structuring is *clustering*, but there are others.

There are many applications of unsupervised learning: Market segmentation, social network analysis, organizing computing clusters (if you know which computers work tightly together, you can optimize layout). and astronomical data analysis.

9.1 K-means algorithm

We would like to cluster data into coherent clusters. K-means is by far the most used clustering algorithm. The algorithm goes like this:

- first randomly select the *cluster centroids* (as many or few as you like, the algorithm will not determine the optimal number of clusters for you).
- Cluster assignment step: Assign the datapoints to the centroid that is the closest. (use a voronoi tessalation of the plane)
- Move centroid step: Move the centroids to the average (mean) location of all the points in the cluster.
- Go back to the cluster assignment step.
- Repeat until convergence.

That was it :-) K-means does a pretty good. We'll drop the intercept term. Upper case K is used to indicate the number of cluster centroids, lowercase k used to indicate a particular cluster center. The squared distance is used by custom (minimizing square or unsquared doesn't change the result of the algorithm).

If you get one cluster centroid you can either eliminate that centroid (the ordinary thing to do), or just randomly place the centroid.

K-means for non-separated clusters, it sometimes works well like in the T-shirt example in figure ??. You get a market segmentation out of the box. It may even be useful.

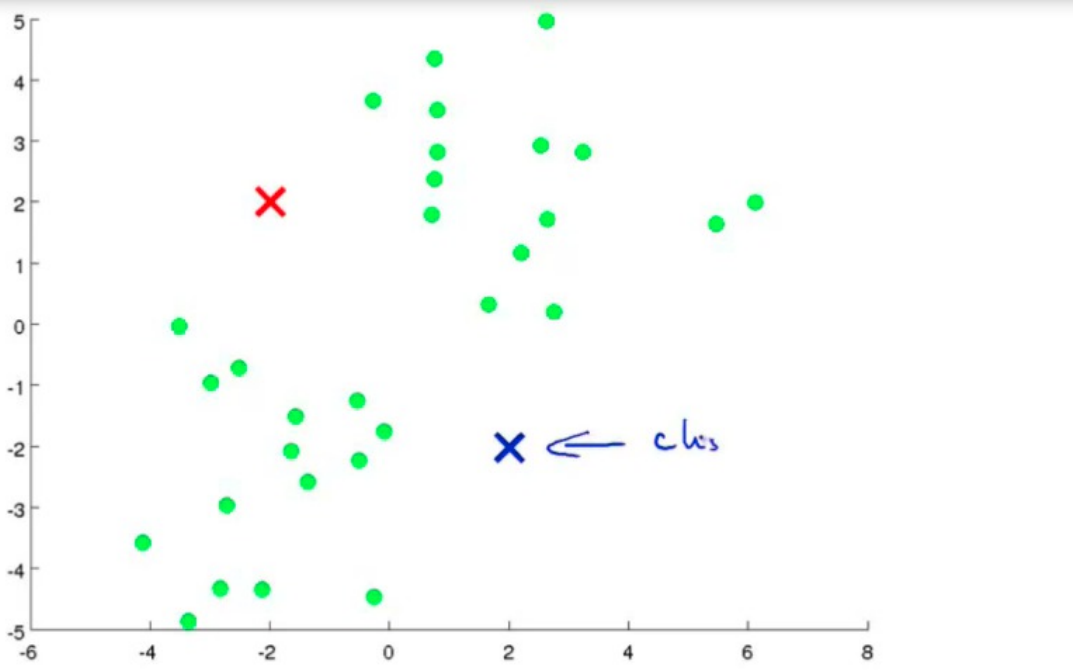


Fig. 9.1 K means clustering

9.1.1 Clustering optimization objective

Clustering also has a cost function. This is useful because it helps us debug the algorithm, but it can also help us to find better clustering and avoid local optima.

$c^{(i)}$ = a cluster of $(1, 2, \ldots, K$ to which example clustering and avoid local optima$x^{(i)}$ is curre

μ_k = cluster centroid k ($\mu_k \in \mathbf{R}^n$)

$\mu_c(i)$ = Cluster centroid of cluster wo which example clustering and avoid local optima$x^{(i)}$ has

With this we are ready to type out the optimization objective for the clustering and avoid local optimak-means algorithm, and in figure clustering and avoid local optima?? it is.

The average of the squared distance of the distance between the training examples and the centers of the clusters they are assigned

The cost function is called the *distortion function*. In a sense what the k-means algorithm does is to partition the parameters and then use the assignment step to minimize one part (the cluster assignments) and then to minimize the other part (the assignments), and then repeat until convergence.

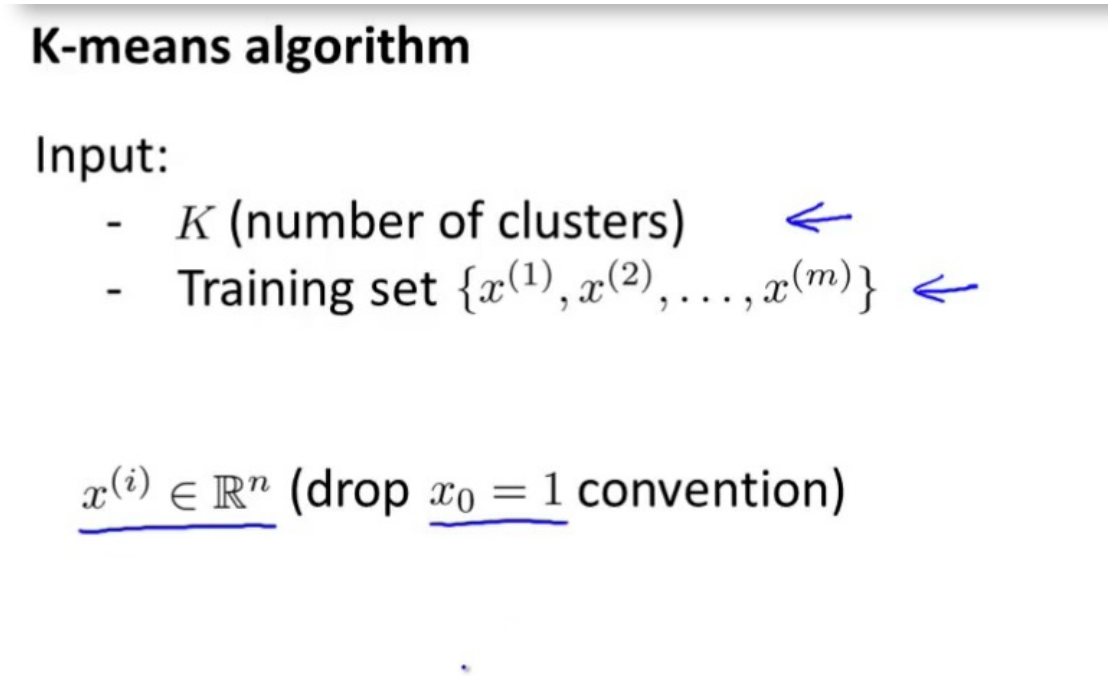


Fig. 9.2 K means clustering algorithm

9.1.2 Random initialization

There is one highly recommended way to pick cluster centroids. First, let $K < m$. Then randomly pick K training examples and let $\mu_1, \ldots, \mu_K$ be equal to these K samples. Easy peasy.

K-means can end up at different solutions depending on which centroids we start at (local optima of the distortion function). To avoid this, we can try multiple random initialization to make sure that we get as good a global optimum as possible. Perhaps fifty to hundred times. We will then *compute the distortion* for each of the candidates. We then pick the clustering with the lowest cost.

It turns out that if the number of cluster is small (to to ten) can sometimes find a better optimum, but if K is large then random initilization will improve the solution since the initial solution will be pretty good.

9.2 Choosing the number of clusters

There isn't a good way to do this automatically. The most common way to do this is to do it manually.

It's like this since it is often genuinely ambiguous how many clusters there

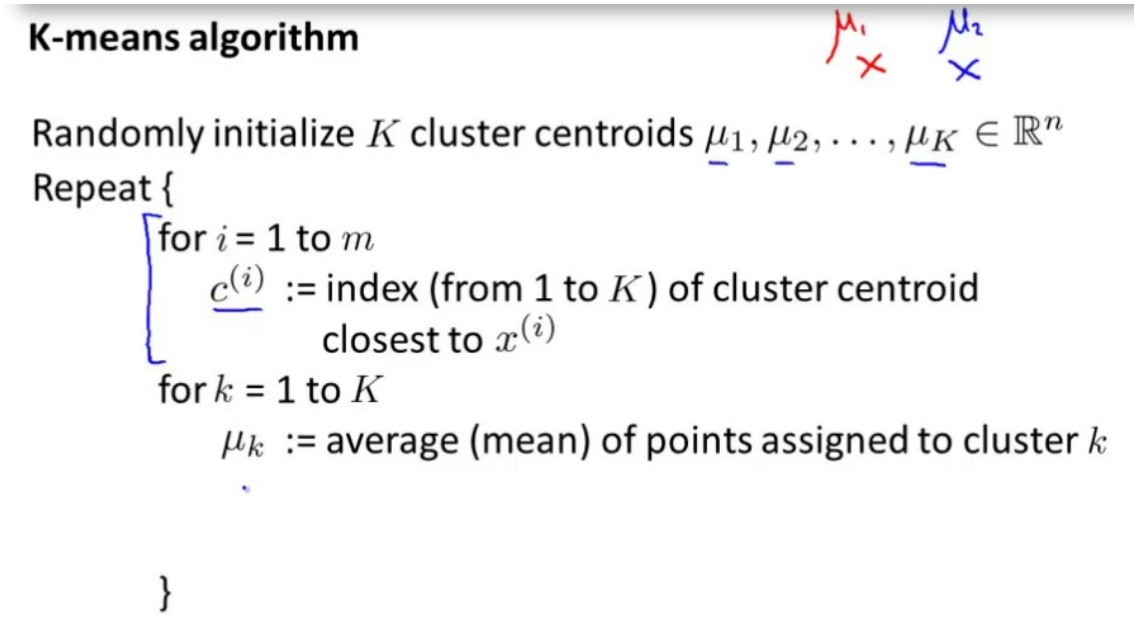


Fig. 9.3 K means clustering algorithm

are. There is an automatic method called the *elbow method of cluster number determination.*

The idea is that plot of the cost and the number of clusters, and then look for an "elbow" where the drop in the error function seems to be leveling out. That's where the cutoff point is placed.

When it works and you really get an "elbow", use it and be happy. However, it can often not work. It's worth a shot, but it doesn't actually have a high expectation of giving a clear cut answer for any particular problem.

Another way to choose K to use some other metric. E.g. how well the downstream method (market segmentation, computer cluster configuration or something) fares. If there is an evaluation method available in the downstream problem, one should consider using that.

For instance, we could have three or five sizes of T-shirts (fig. ??).) This is actually a business-question that isn't obvious from the cluster metric.

Tradeoffs between filesize and compression distortion in a k-means cluster based image compression method is another example of using the downstream evaluation method.

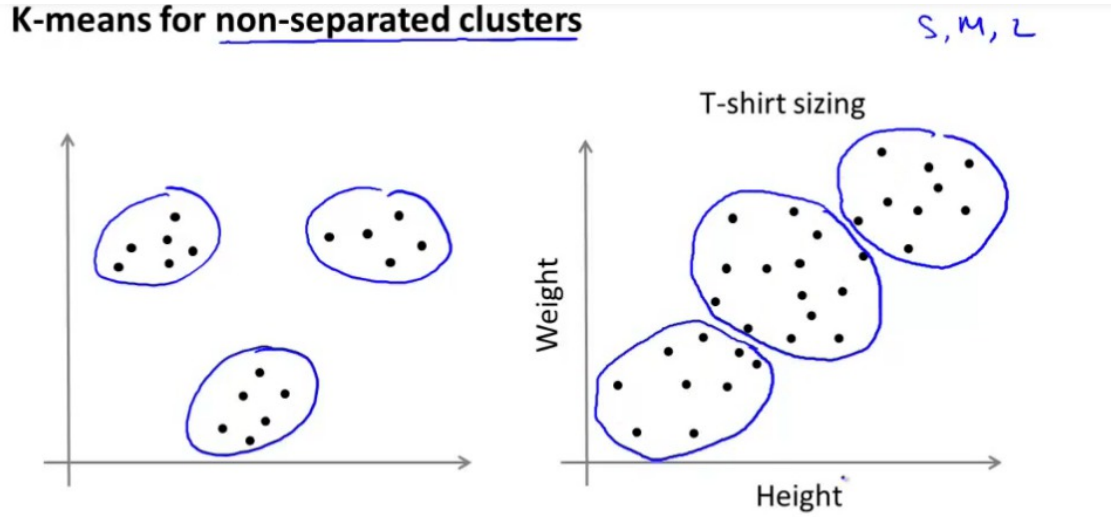


Fig. 9.4 One can cluster also when the data isn't clearly "clustered", for instance the graph on the right might be useful for clustering T-Shirt shizes

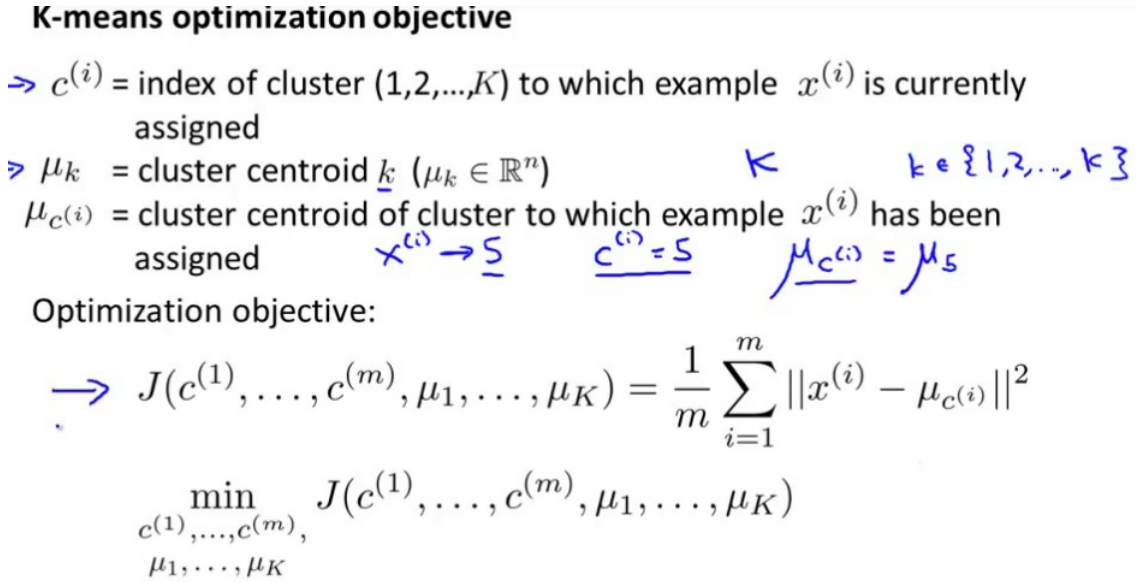


Fig. 9.5 The optimization criterion for K-Means clustering

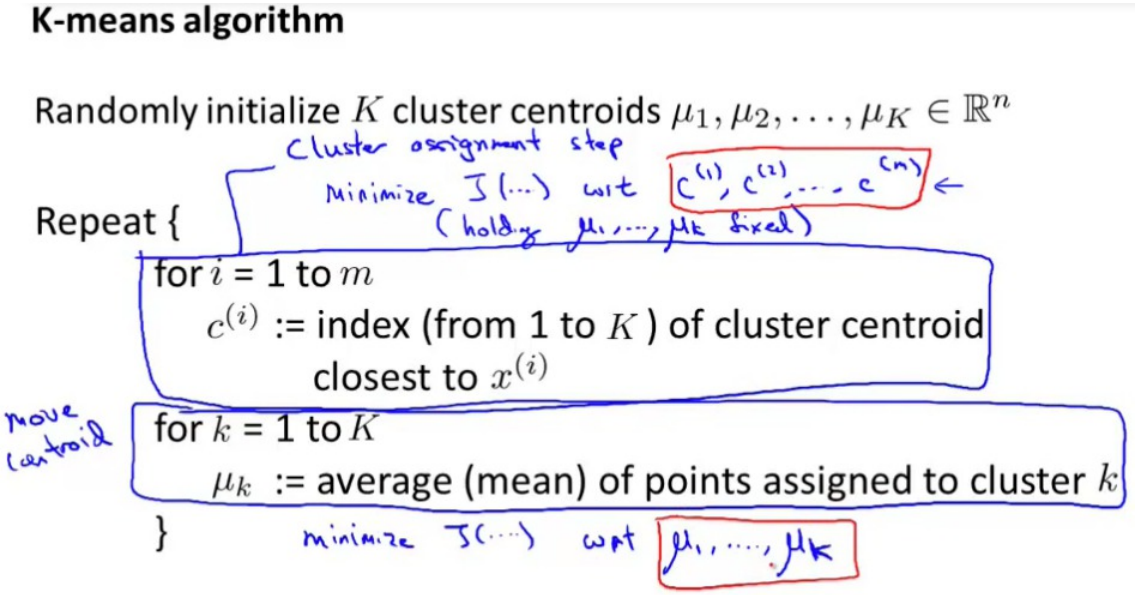


Fig. 9.6 The K-Means algorithm as pseudocode

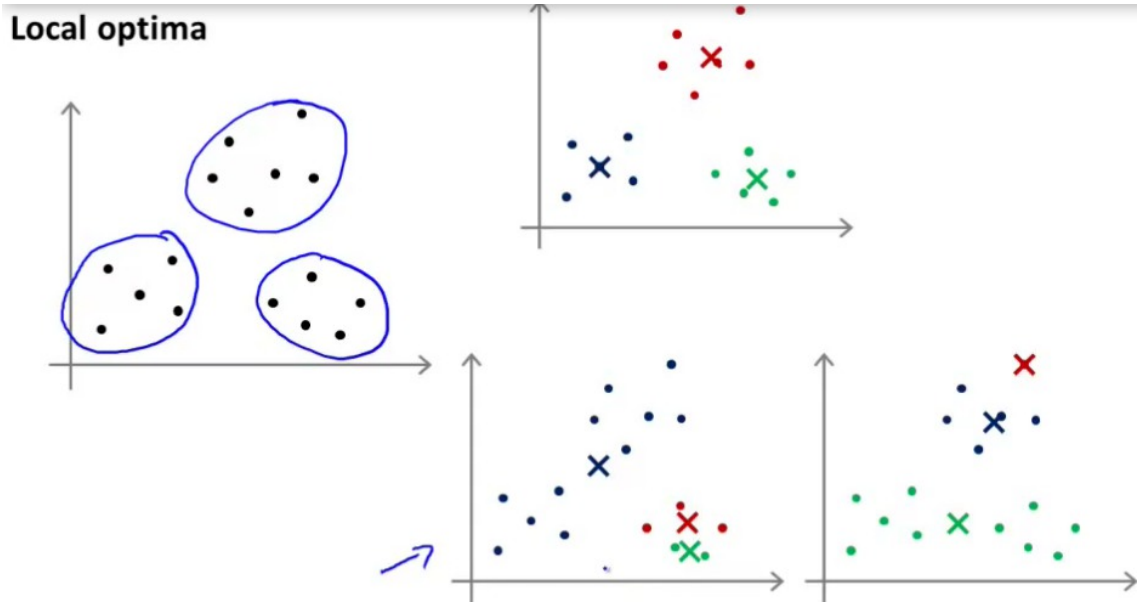


Fig. 9.7 localoptimakmeans

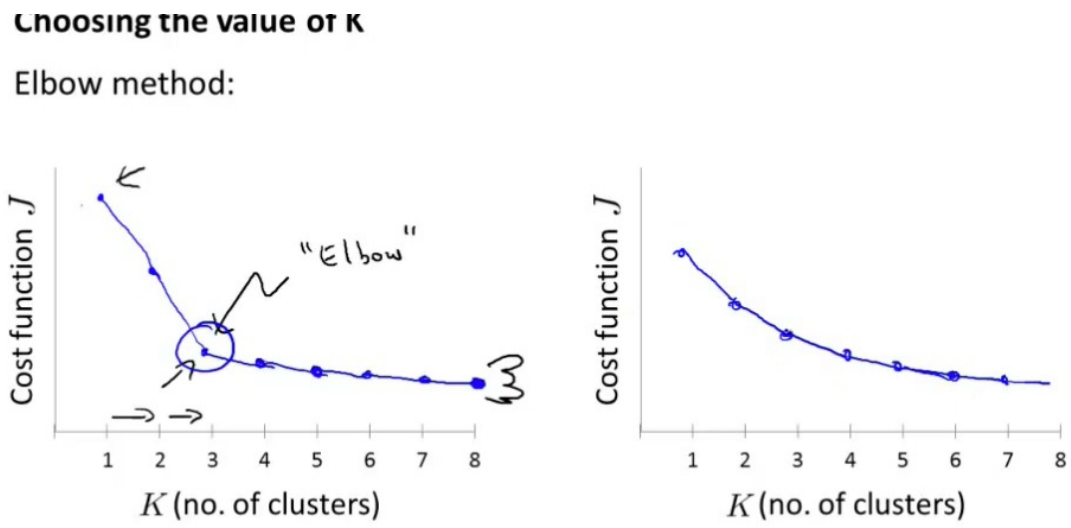


Fig. 9.8 elbowmethod

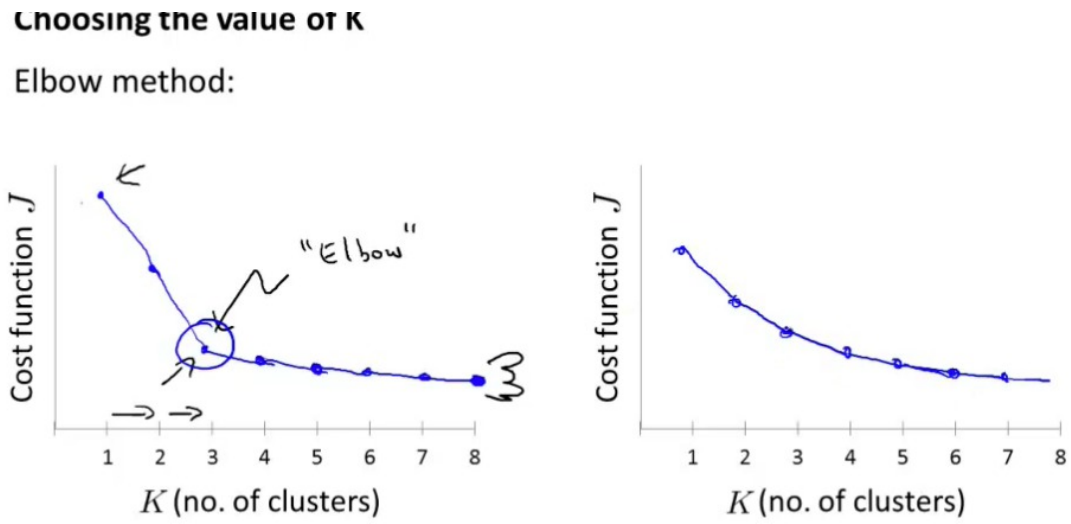


Fig. 9.9 The "Elbow" method for choosing the number of clusters looks at how the cost function varies with the number of cluster. If an "elbow" pattern can be observed, it can be used to select the appropriate number of clusters, but often there is no clear elbow and then the method is not applicable.

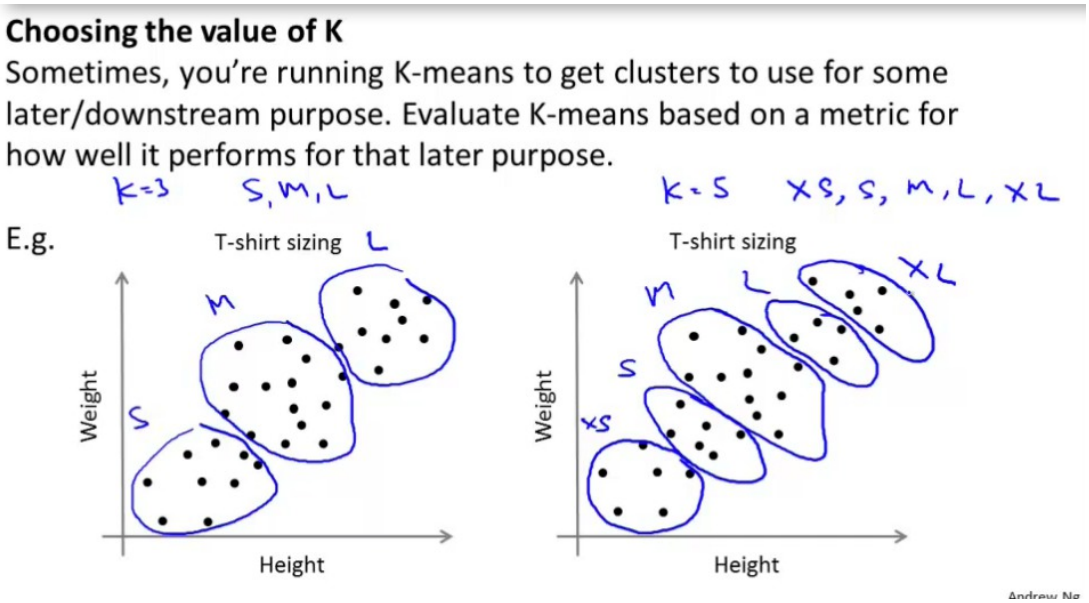


Fig. 9.10 Choosing the number of clusters is based on some metric on how well some downstream metric matches the number you choose. How many sizes of T-Shirt to produce may or may not be directly dictated by the variation in the population of T-shirt consumers.

10 Dimensionality reduction

DImensionality reduction is another type of unsupervised learning. We can use it for compression but will also help us to improve the performance of our learning algorithms.

If you have a bunch of features (several thousand from different sources) so it's common to have highly redundant parameters. (measuring inches and cm, for instance). A couple of other things that might by highly correlated is pilot enjoyment and pilot skill, so instead one could use a common variable to describe them both (and perhaps call it "pilot aptitude", although it isn't necessary to give these things explicit names in order to make use of them).

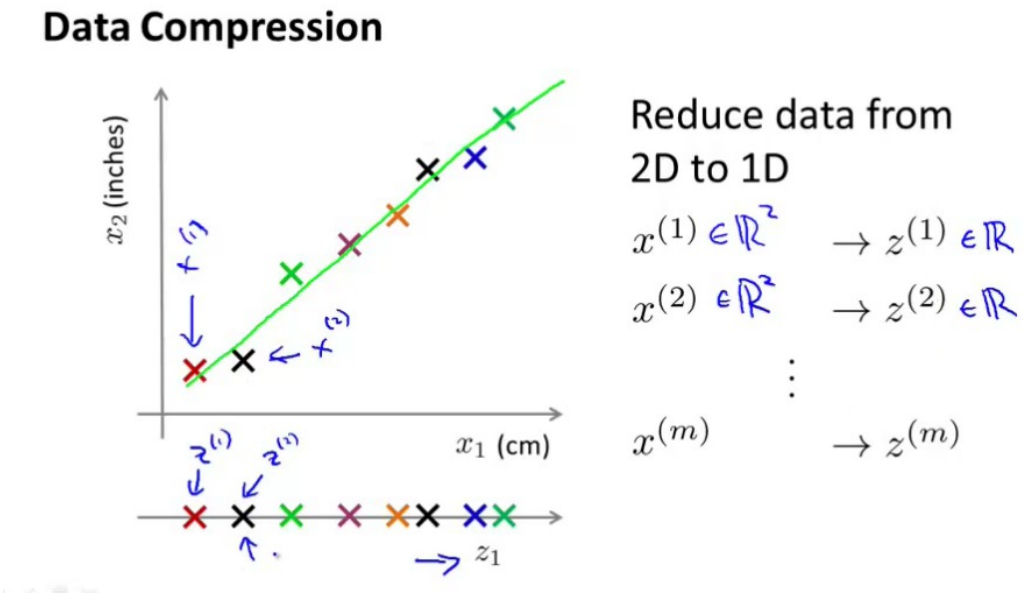


Fig. 10.1 PCA Principal component analysis, is used to find the directions in the dataset where the variation is maximal. Projecting the data points along these directions (vectors) will reduce the number of coordinates that are necessary to describe the dataset with high accuracy.

We approximate the original dataset by projecting down on the synthesized feature vector.

We can also compress data from 3D to 2D (or ten thousand dimensions to a hundred dimensions, but that's harder to show on a screen)

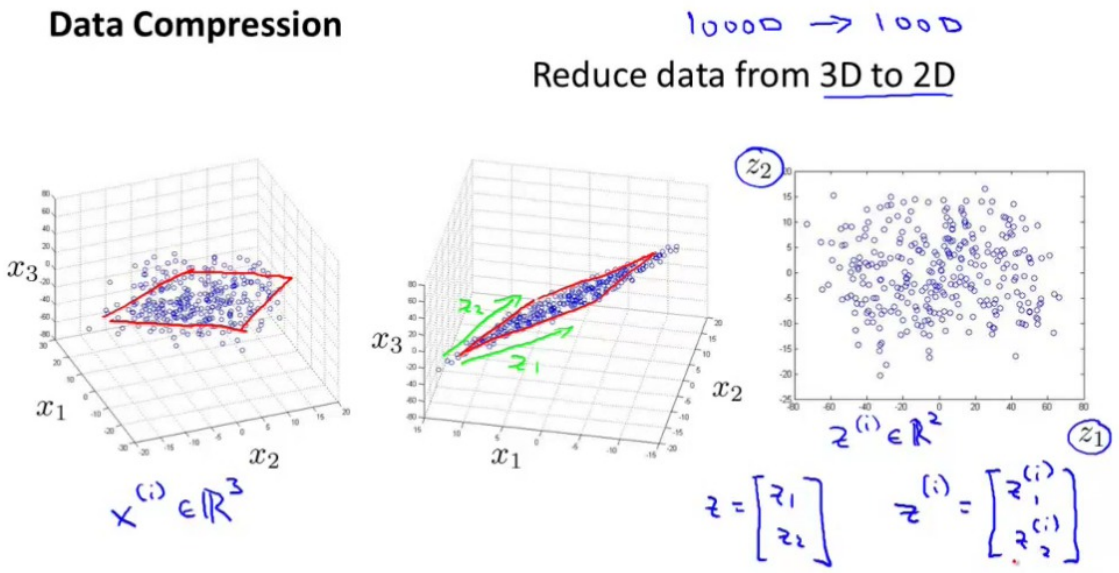


Fig. 10.2 The three dimensional data in this dataset all lie mostly in a plane. The points in this plane can be described by being projected on two vectors in the plane, and thus only two, not three coordinates are necessary to describe most of the variation in the dataset.

In fig ?? the original 3D data is located in a plane (a subspace of 3D), ao we can project all of the data down on that plane, and get 2D coordinates instead of 3D coordinates.

10.0.1 Visualization

Visualization is a really nice way to help us understand the data we use. If we look at the data in fig ?? it's difficult to see any kind of structure in that data, at least at first glance. If there are fifty or more data per country the problem gets worse :-)

Using dimensionality reduction we can project this data down onto a lower dimensional space with e.g. two or three dimensions. If that representation can summarize the fifty numbers, then we can plot the data and understand the variation.

Usually the new dimensions must be interpreted by us (humans). We typically get something like what is shown in fig ??, but we must figure out what the

Data Visualization

Country	x_1 GDP (trillions of US$)	x_2 Per capita GDP (thousands of intl. $)	Human Develop-ment Index	Life expectancy	Poverty Index (Gini as percentage)	Mean household income (thousands of US$)	...
Canada	1.577	39.17	0.908	80.7	32.6	67.293	...
China	5.878	7.54	0.687	73	46.9	10.22	...
India	1.632	3.41	0.547	64.7	36.8	0.735	...
Russia	1.48	19.84	0.755	65.5	39.9	0.72	...
Singapore	0.223	56.69	0.866	80	42.5	67.1	...
USA	14.527	46.86	0.91	78.3	40.8	84.3	...
...	...	...	...	...	...	...	

[resources from en.wikipedia.org]

Andrew N

Fig. 10.3 Gdp v.s. the size of the country

dimensions really mean.

10.1 Principal component analysis

PCA minimizes the sum of squares of the distances from the dataset to a surface. This error is called the *projection error*. A standard practice is to perform mean normalization and feature scaling, so that the features have zero mean and a comparable range of values.

Formally the goal of PCA is to find a direction onto which to project the data to minimize the projection error. It doesn't matter if the direction of the vector goes in a positive or negative direction, since the surface containing the vector is the same.

We're out to project the data onto the linear subspace defined by our vectors.

Btw, PCA is not linear regression even though there are some cosmetic similarities.

In PCA there is no distinguished "y" we are trying to find.

10.1.1 The PCA algorithm

Before using a training set $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$, it's important to perform feature scaling/mean normalization. First find the mean:

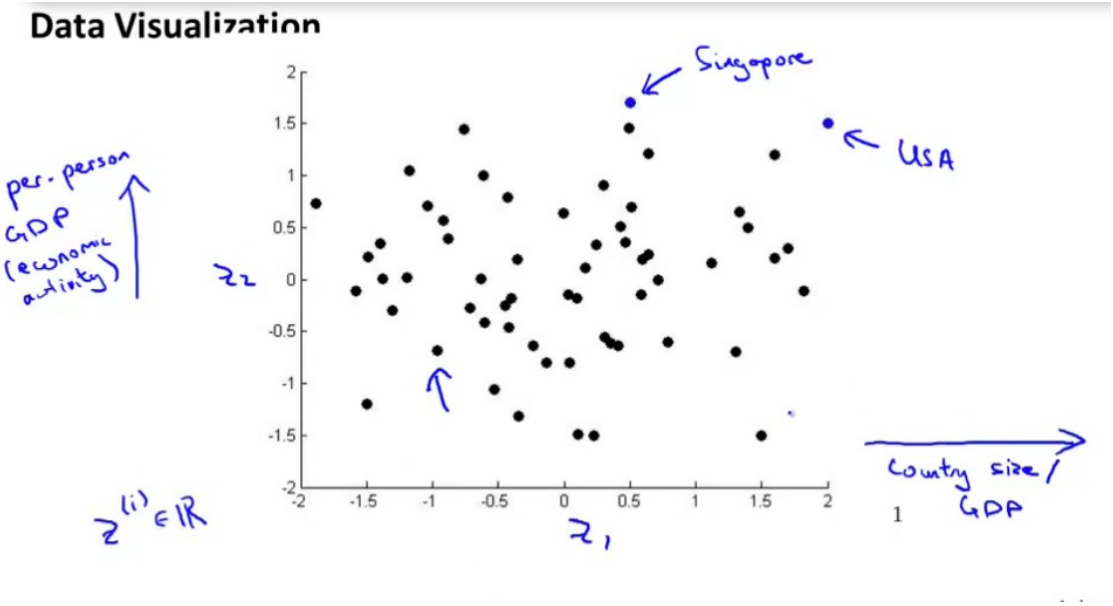


Fig. 10.4 worlddata2d

$$\mu_j = \frac{1}{m} / sum_{i=1}^{m} x_j^{(i)}$$

Then We should replace each $x_j^{(i)}$ with $x_j - \mu_j$. If different features are on different scales (e.g. sizes of houses in square feet and number of bedrooms), it will be hard to compare values. This is the same process used for supervised learning.

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$$

where s_j is the maximum, minimum or most commonly standard deviation of the x_j values.

The procedure is pretty simple:

- Compute the *covariance matrix*

$$\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T = \frac{1}{m} X^T X$$

- Compute the *eigenvectors* of the matrix Σ:

```
[U,S,V] = svd(Sigma);
```

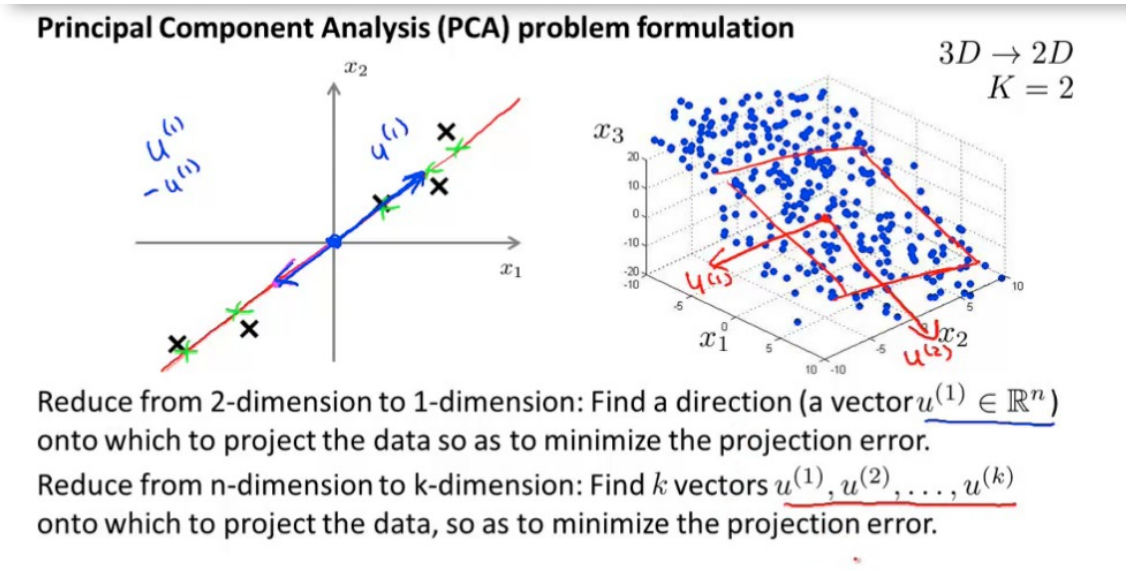


Fig. 10.5 The objective of the PCA algorithm is to find the directions that minimizes the projection error.

The difference between "svd" and "eig", but "svd" is a bit more numerically stable, but applied to a *covariance matrix* it will always give the same answer. All covariance matrices are *symmetric positive semidefinite* but that's not important right now :-) That wasn't so hard was it?

The covariance matrix will be an n by n matrix. The U matrix is an N by N matrix containing columns that are the vectors we want to project down onto. If we want to project only onto k vectors, then we pick the k first vectors (columns) in the U matrix. We stack these columns and call this n by k matrix the U_{reduce} matrix. We can then compute the recued data set Z like this:

$$Z = U^T_{\text{reduce}} X$$

10.2 Choosing the number of dimensions to extract using PCA

The parameter k, the number of *principal components* that we are extracting using PCA is something we must determine. Here are some ways to do that. There are some things that are of interest

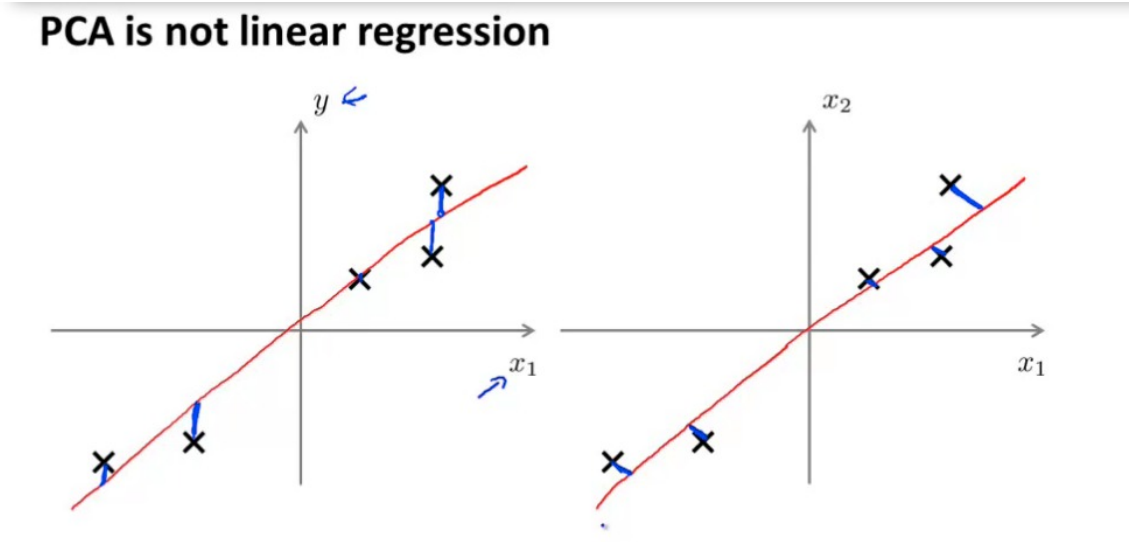


Fig. 10.6 PCA may superficially look similar to linear regression, but they are in fact quite different: PCA optimizes the minimum of the distance to the line, but linear regression optimizes the minimum distance between data points in the line only in the vertical direction.

- Average squared projection error:

$$\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x^{(i)}_{\text{approx}}||^2$$

- The total variation in the data:

$$\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2$$

- The typical rule to use: Retain 99 percent of the variation:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x^{(i)}_{\text{approx}}||^2}{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2} \leq 0.01$$

Some other values would be 0.05 (five perdent of the variance retained) etc. Perhaps as low as 0.85, but 95-99 perent is useful. For many real data datasets it is often easy to get these numbers with relatively small values of k.

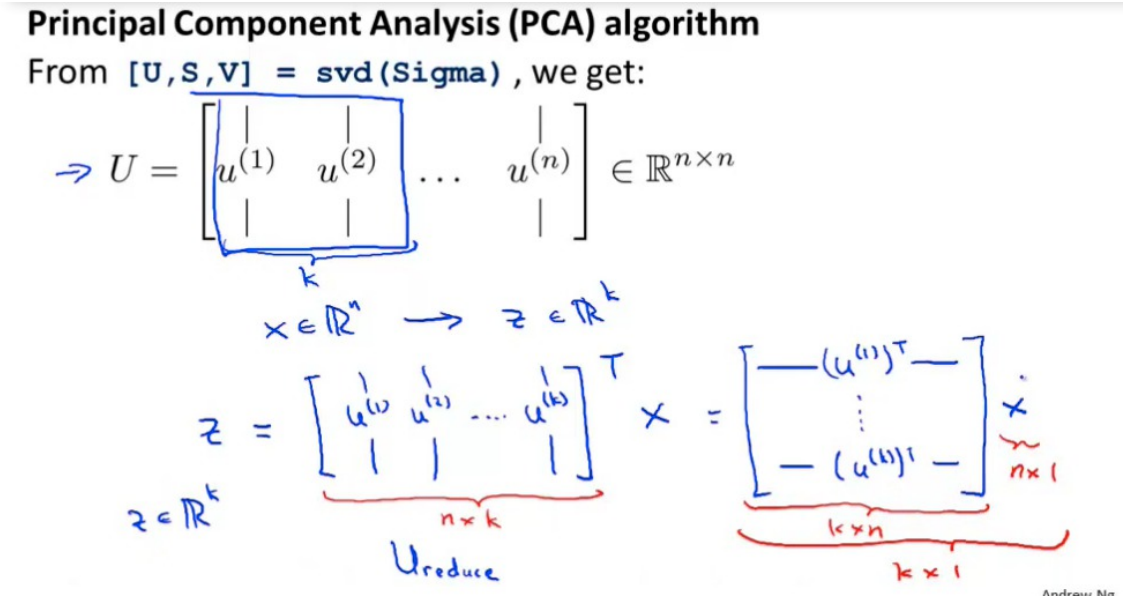


Fig. 10.7 Restoring data from PCA

One algorithm is to perform PCA on the training set until the variance target is reached and then select the k that gave the wished-for number. However this is inefficient. Fortunately PCA gives us a value that can be used. The inner-loop gives us these results.

```
[U, S, V] = svd(Sigma)
```

The S value is a diagonal matrix of *eigenvalues* (*singular values* in this case), and it so happens that we can use that to compute the quantity describing the retained variation much more simply than the hairy expression above. Explicitly, assuming that the S matrix has nonzero elements $(s_{11}, s_{22}, \ldots, s_{nn})$ then

$$\frac{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)} - x_{\text{approx}}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m} ||x^{(i)}||^2} = 1 - \frac{\sum_{i=1}^{k} s_{ii}}{\sum_{i=1}^{n} s_{ii}}$$

And (assuming that calculating all the eigenvalues is efficient), then this will be an efficient method.

Plotting the s_{ii} values in a histogram could also be used in the "*elbow*" method described in a previous section to see if there are any kinks in the value of variance retained vs k and select the number where the explained variance levels off.

Rmz: *Actually Ng doesn't recommend this, but imho it's still a good idea if the point is to avoid overfitting, which really isn't so much of an issue if compression is the point :-)*

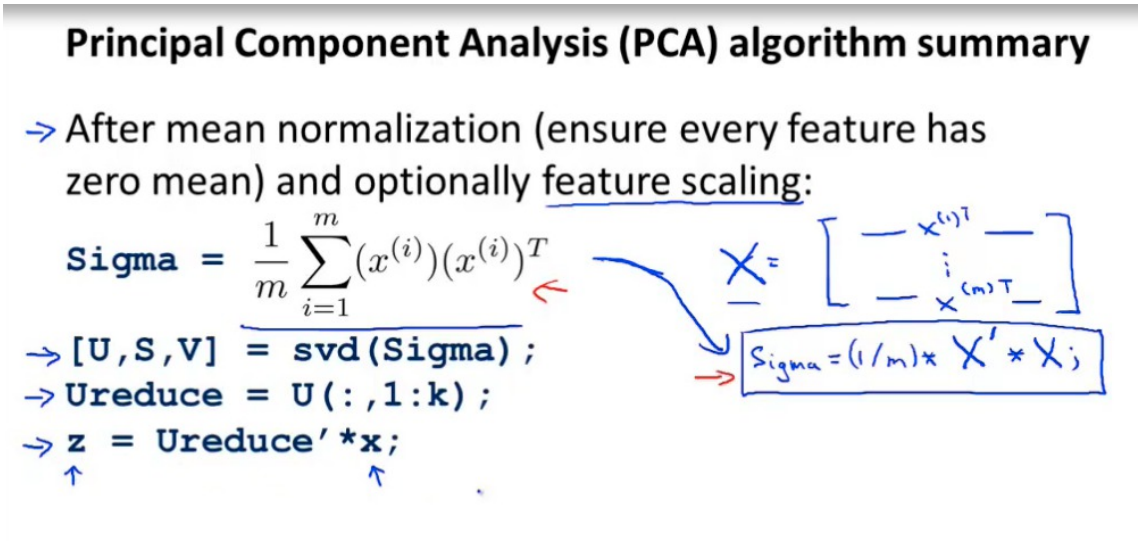


Fig. 10.8 PCA in a single slide. The SVD function in Octave gives us what we need (not optimally efficient but it's ok)

Missing imagefile file choosingk

Fig. 10.9 choosingk

10.3 Reconstruction from compressed representation

How to go back from the compressed data back to the original representation?

Consider a two dimensional dataset we wish to reduce. We do that by projecting:

$$z = U^T \text{reduce} x$$

If we want to go the oposite direction:

$$x_{\text{approx}} = U_{\text{reduce}} z$$

This process is called *reconstruction* of the original data.

10.4 Advice for applying PCA

PCA can be used to speed up a supervised learning algorithm. This is the most common way that Ng uses PCA. Assume that we have a very high (ten thousand or so) feature vectors (e.g. in computer vision):

$$(x^{(1)}, y^{(1)}), (x^{(1)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$$

Assume that we wish to use some algorithm, *SVM*, *neural network* or whatever we can reduce the algorithm to work on lower dimensional data which will make it run more efficient. The method to use is illustrated in figure **??**. Just remember to apply the U_{reduce} found in the training set on the cross validataion and test sets. It is often possible to reduce the number of dimension five or ten times without reducing the classification performance. When using PCA for learning algorithms we are using a variance retention metric and use some high k.

For visualization we often use k=2 or 3.

Misuse

There is a frequent misuse of PCA: To prevent overfitting. The number of dimensions (features) in the learning algorithm, fewer data will prevent overfitting. This is **bad**. Use regularization instead :-) PCA throws away data without regards to the y value and will often perform worse than regularization, which knows something of the value of y.

Missing imagefile file pcaplanwithflaw

Fig. 10.10 pcaplanwithflaw

Before committing to using PCA, one should also consider doing the whole thing without PCA: PCA adds complication and if the project is feasible without PCA, then that's perhaps the best way of doing it.

Only use PCA if the algorithm runs too slow, or the memory footprint is too large.

11 Anomaly detection

AD has some features of unsupervised learning but have som aspects of supervised learning. Assume we are a manefacturer of aircraft engines. We measure features (heat, vibration etc.) You now have a data set of many features. The anomaly detection problem is to to check if any engine is anomalous in any way, should it undergo further testing or something. Anomalies should be examined more before it sent to the customers.

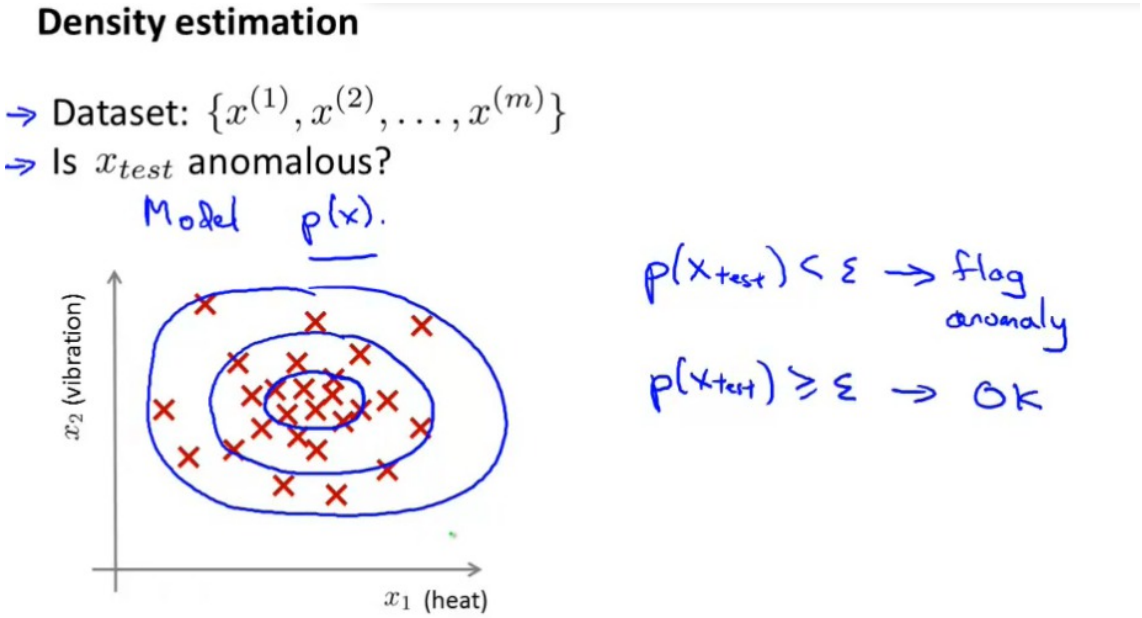


Fig. 11.1 Using Gaussians to estimtate the density of a set of data points

Formally, if we see a sample that has a very small probability of being part of the model that defines the normal samples, then we should flag an anomaly. Some examples of anomaly detection is

Fraud detection (user activities, with a model of user activities). Identify unusual users by checking which have $p(x) < \epsilon$. (typing speed, etc.).

This flags users that behave strangely, not just fraudsters.

Another example is manufacturing.

Yet one is montiing computers in a data center (memory use, numer of disk acccesses, CPU load, load /network traffic, etc.)

11.1 The Gaussian distribution

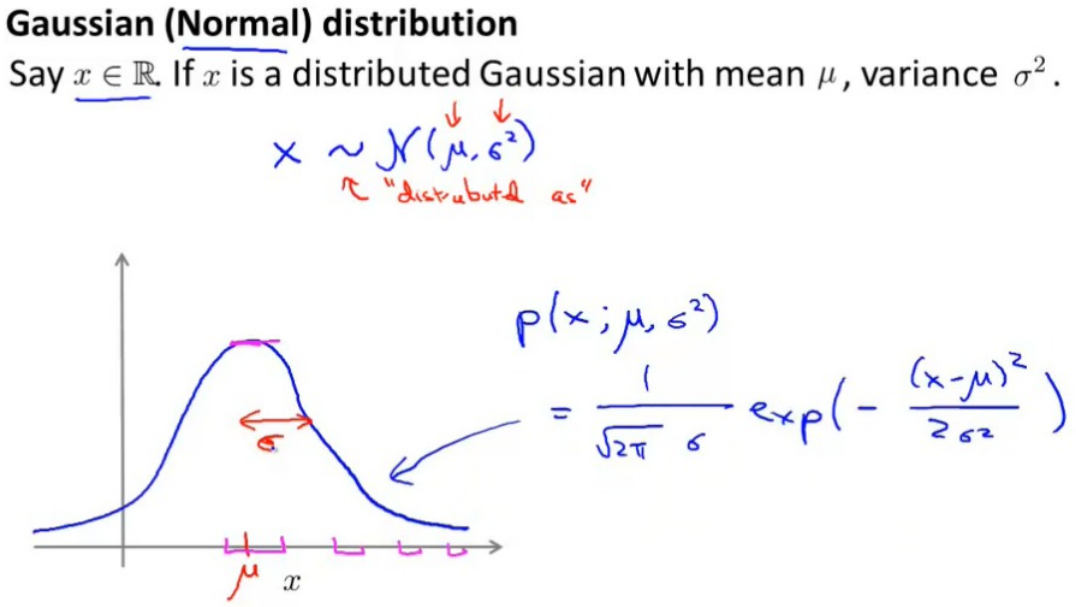


Fig. 11.2 The Gaussian distribution and the standar parameters used to describe it.

$$x \sim \mathcal{N}(\mu, \sigma^2)$$

The Gaussian probability distribution is parametrized has mean μ and with defined by σ.

$$p(x; \mu, sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{x\sigma^2}\right)$$

σ is called the *standard deviation* and σ^2 is called the *variance*.

The *parameter estimation* is the problem of fitting a data set to a Gaussian. How to find the parameters σ^2 and $\mu$for the Gaussian that best fits the given data? The standard formulas for making these estimates are:

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ \sigma^2 &= \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)} - \mu\right)^2 \end{aligned}$$

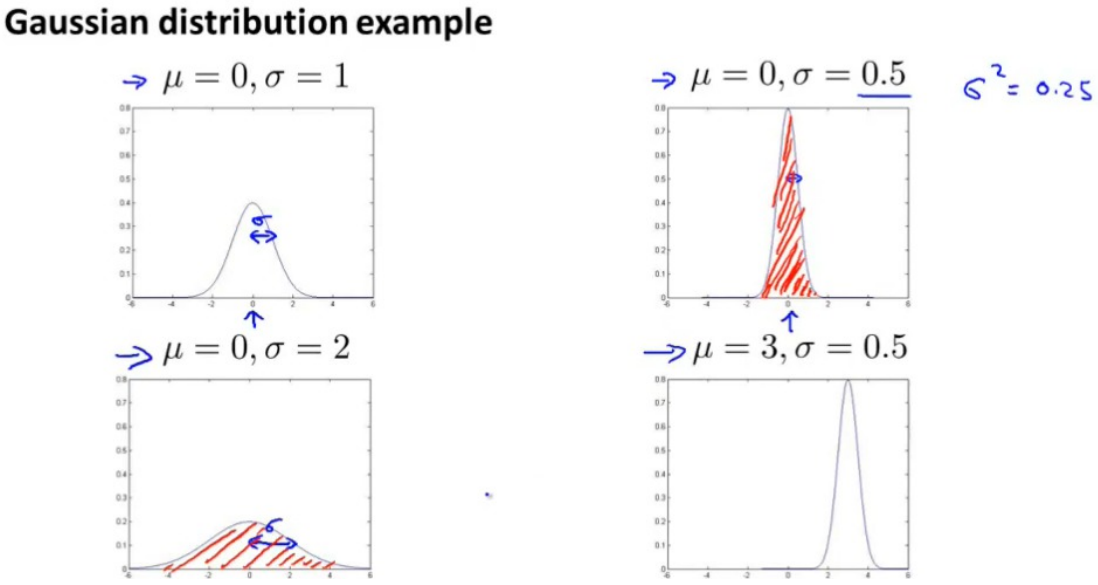


Fig. 11.3 Some examples (wide and narrow) of Gaussian distributions and their parameters.

These parameters are actually the *maximum likelyhood estimators* for the normal distribution. Note (some versions of this formula use 1/(m-1)).

11.2 An anomaly detection algorithm

Assume we have a training set $x^{(i)}, \ldots, x^{(m)}$, each sample is a real number. We'll model this as a product of Gaussian distributed random variables, so that $x_i \sim mathcalN(\mu, \sigma^2)$.

$$p(x) = \prod_{j}^{n} p(x_j; \mu_j, \sigma_j^2)$$

This equation correspond to an independence assumption on the features, but it usually works fine even if the terms are not entirely independent. This problem is also called the problem of *density estimation*.

An algorithm to detect anomalies can then be:

- Choose features x_i that might be indictive of anomalous examples (e.g. temperature, vibration, speed of typing etc.)

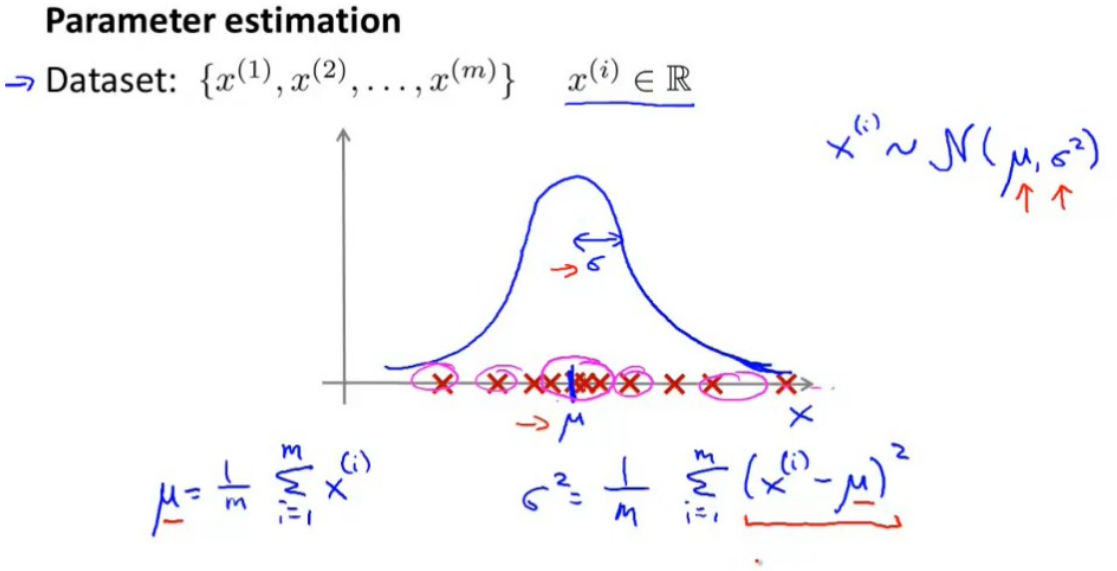


Fig. 11.4 Fitting a data set to a Gaussian is easy, just find the average and the variance and you're done.

- Fit the parameters $(\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots \sigma_n^2)$ using the max likelyhood estimator:

$$\begin{aligned} \mu &= \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \\ \sigma^2 &= \frac{1}{m} \sum_{i=1}^{m} \left(x^{(i)} - \mu\right)^2 \end{aligned}$$

- Given a new example x, compute $p(x)$ using:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x - \mu_j)^2}{x\sigma_j^2}\right)$$

  Assume that you're looking at an anomaly if $p(x) < \epsilon$.

The algorithm can be vectorized, e.g. using this trick:

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{bmatrix} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

Anomaly detection algorithm

1. Choose features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

3. Given new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Anomaly if $p(x) < \varepsilon$

Fig. 11.5 An algorithm to detect anomalities: Find the multidimensional Gaussian, see how well a single datum is described by this Gaussian, and establish a threshold to define anomalities.

11.3 Developing an evaluating an anomaly detection system

It's important to have real numbers that can be used to evaluate the learning algorithm, this is called *real-number evaluation*. We can use this to include/exclude features etc.c We will assume that we have som labelled data, assuming $y = 0$ if normal and $y = 1$ as anomalous. We then take a training example which is a large example of normal samples, however it's also ok to have a few anomalous cases too. We then have a cross validation set and a test set.

Assume 10000 good (normal) engines. We have 20 flawed engines in this training set. Usually there are much more good than bad samples. One way of splitting is to use 6000 known good as the unlabled training set (y=0), test is 2000 engines with ten known anomalities and 2000 (also with ten known anomalities) is used for crossvalidation (6+2+2). Using the same items in the crossvalidation and test sets is not usually a good idea.

We then evaluate the algorithm like this:

- Fit a model on the training set (max likelyhood estimation of the Gaussians).

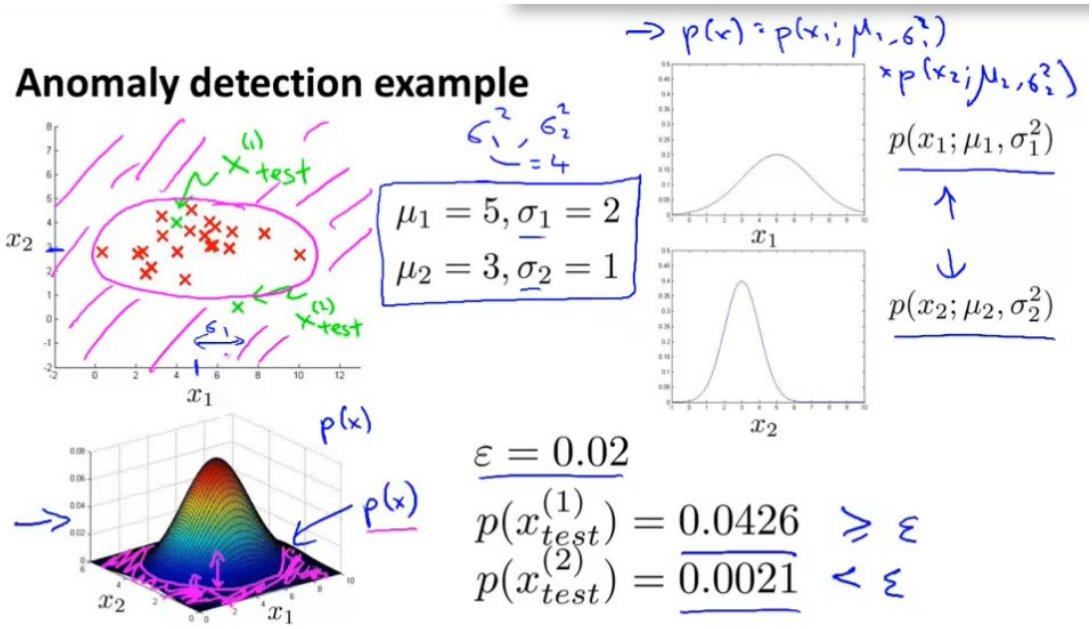


Fig. 11.6 Example of an anomaly detector

- On the cross-validation test examples x, predict:

$$y = \begin{cases} 1 & \text{if } p(x) < \epsilon \text{ anomaly} \\ 0 & \text{if } p(x) \geq \epsilon \text{ anomaly} \end{cases}$$

- Possible evaluation metrics:
  - True positives, false positives, false negatives, true negatives
  - Precision/recall
  - F_1-score (a way to summarize preciesion/recall things)
- We can also use cross validation to choose the parameter ϵ. Just try many and pick one that maximizes F_1 or something.

This is actually not that dissimilar to supervised learning. The labels will be very *skewed* since there are many more of the normals than the anomalies. Classification accuracy is not very accurate.

We can then use the set of parameters and evaluate the parameter use on the cross validation set.

11.4 Anomaly detection v.s. supervised learning

If we have labelled data, why don't we use a supervised learning algorithm? You should use anomaly detection when very small number of positive examples and a large number of negative examples. We'll save the negative examples for test and cross validation. If you have lots of both positive and negative examples so then you can use supervised learning.

Another way is to detect many different ways things things can go wrong. It's hard to learn from the small set of things that went bad, in particular the future anomalies may look nothing like the anomalous examples seen so far.

On the other hand, given enough examples of both classes it's possible to get a good sense of what future bogusness will look at.

All of this is just another way of saying that anomaly detection is usually better when there are very *skewed samples*.

Some applications of anomaly detection: Fraud detection, manufacturing QA, Monitoring machines in data centers. If you have a lot of examples, sometimes supervised learning can be used.

Supervised learning (spam classification, weather prediction, cancer classification (plenty of samples).

In many types of settings there are actually zero samples of negative samples, and in those cases AD is usually used.

11.5 Multivariate Gaussian distribution

MGD has some advantages over the earlier algorithm since it makes use of covariance structures in the data being used. The previous univariate method did not make use of these structures. In the data center example (see **??**), the univariate algorithm will not flag an anomaly even though one should have been flagged.

The modified version will not model the individual parameters directly, but we'll model $p(x)$ for $x \in \mathbf{R}^n$ modelling the same thing in one go. The input parameters is the multivariate average μ and the *covariance matrix* $\Sigma \in \mathbf{R}^{n \times n}$.

```
% Bogus mathaccents, but why?
\[
p(x;\mu, \Sigma) = \frac{1}{(2\pi)^\frac{n}{2}
  |\Sigma|^\frac{1}{2}}\exp\parens{-\frac{1}{2}\parens{x - \mu}^T \Sigma^{-1}(x-\u
\]
```

$|\Sigma|$ is the *determinant* of Σ, can be computed using the octave command `det(Sigma)`. Some examples of multivariate Guassian distributions can be admired in figure **??**.

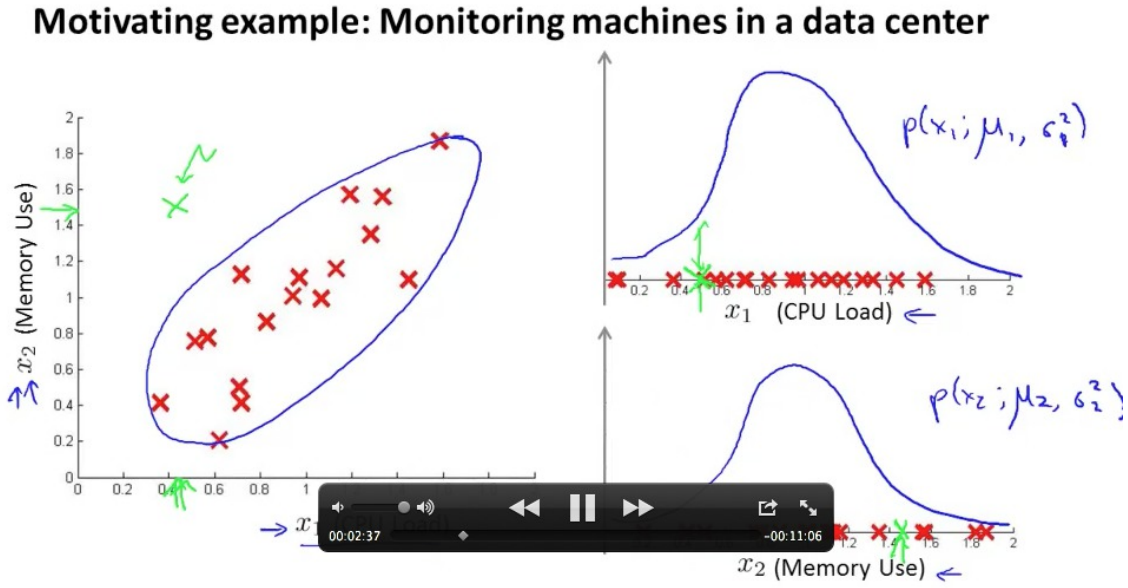


Fig. 11.7 Memory use v.s. machine load in a data center: What is an anomalous situation?

smaller Σ gives sharper bump. Increased sigma gives blunter bump :-) In fact, we can get rotations giving correspondences that are off-diagonal. This means that we can capture both negative and positive correlations.

By varying the mean (the μ parameter) we can translate the peak of the distribution to various points (see ??).

The key advantage of using the multivariate Gaussian disribution is that it actually captures these covariations and makes "sharper" filters.

11.6 Anomaly detection using the multivariate gaussian distribution

An algorithm to check how well a data point fits with a multidimensional Gaussian

We see that $\Sigma = \frac{1}{m} X^T X$ after subtracting the mean.

The algorithm is

1. Fit the model $p(x)$ by setting:

$$\begin{aligned} \mu &= \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \\ \Sigma &= \frac{1}{m} \sum_{i=1}^{m} (x^{(i)-\mu}) \cdot (x^{(i)-\mu})^T \end{aligned}$$

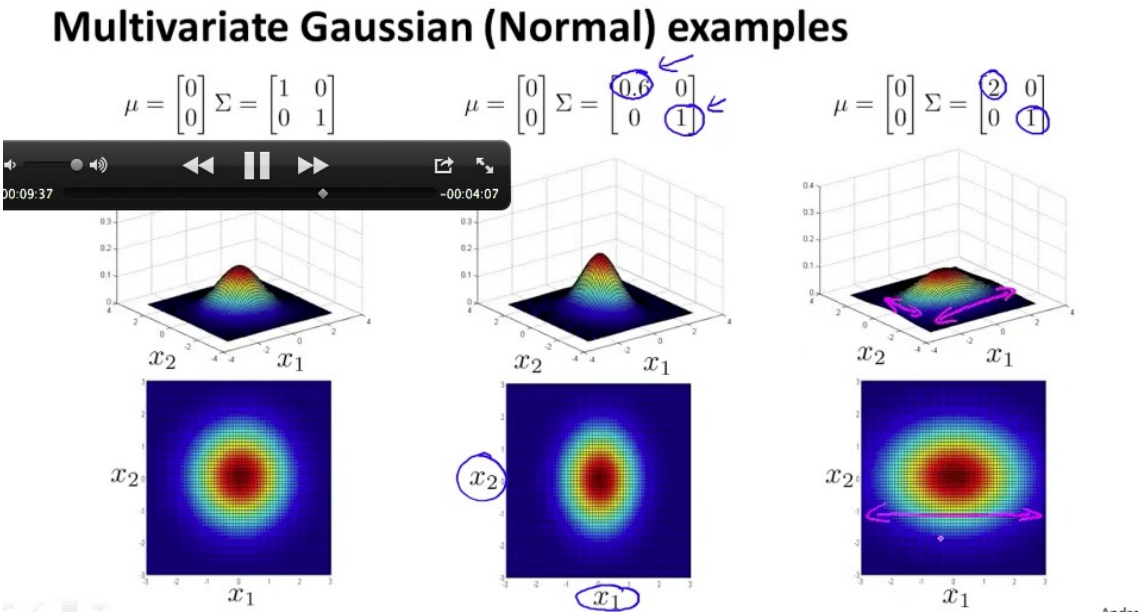


Fig. 11.8 Examples of multivariate Gaussians with parameters.

2. Given a new example x compute:

$$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{n}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T\sigma^{-1}(x-\mu)\right)$$

flag anomaly if $p(x) < \epsilon$.

11.6.1 Relationship with the original model

The original model are actually a special case of multivariate gaussians where the contours of the Gaussians are always *axis aligned*, never at an angle. The defining property for the original model is that has only zero elements on the off-diagonal elements of the covariance matrix.

When would you use these models?

The original model is used more often, and the multivariate gaussian is less often used.

In the original model you can create extra features (such as x_1/x_2 features to capture unusual combinations. The multivariate version captures these automatically.

The original model is computationally cheaper, so it scales to high n (tens or hudred thousands). The multivariate version needs to invert a matrix, which scales

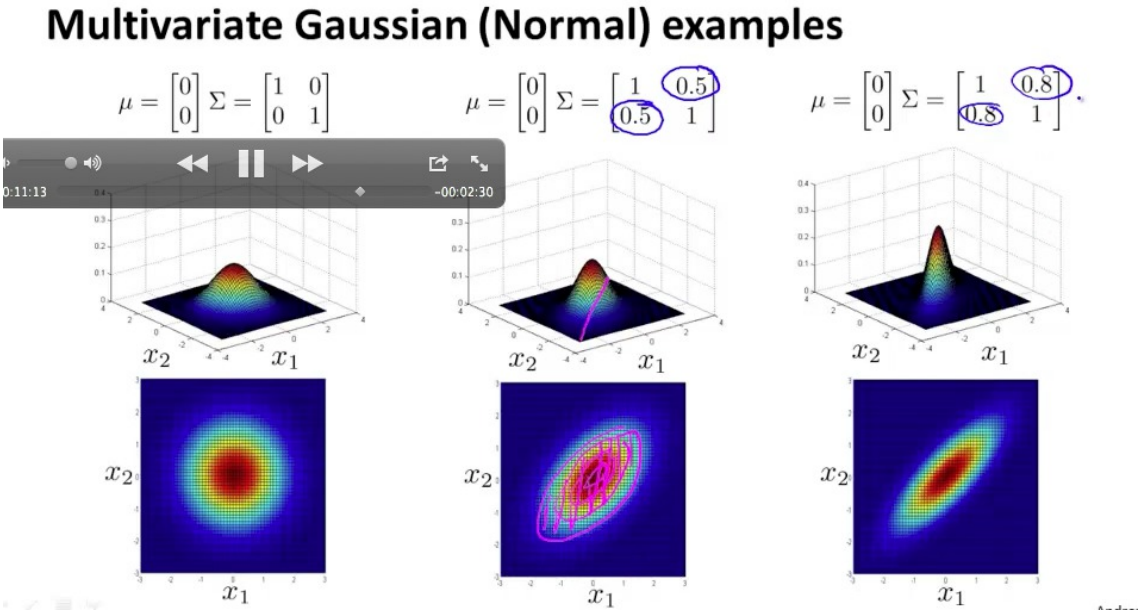


Fig. 11.9 Gaussians where the variation is not along the horizontal or vertical axes.

much less well.

The original version works well with a small training set. The multivariate version needs $m > n$ otherwise Σ will be non-invertible (singular). (some changes can be used :-)

Rule of thumb, use it only if m is much larger than n (ten times or so). There is a lot of parameters ($\Sigma \sim \frac{n^2}{2}$).

One technical property of the multivariate gaussian method is that the Σ may be *singular*, there is usually two reasons for this: Either $n < m$ or redundant features (like $x_1 = x_2$). First check the m and n values then for redundant features. (*linearly depenent features*).

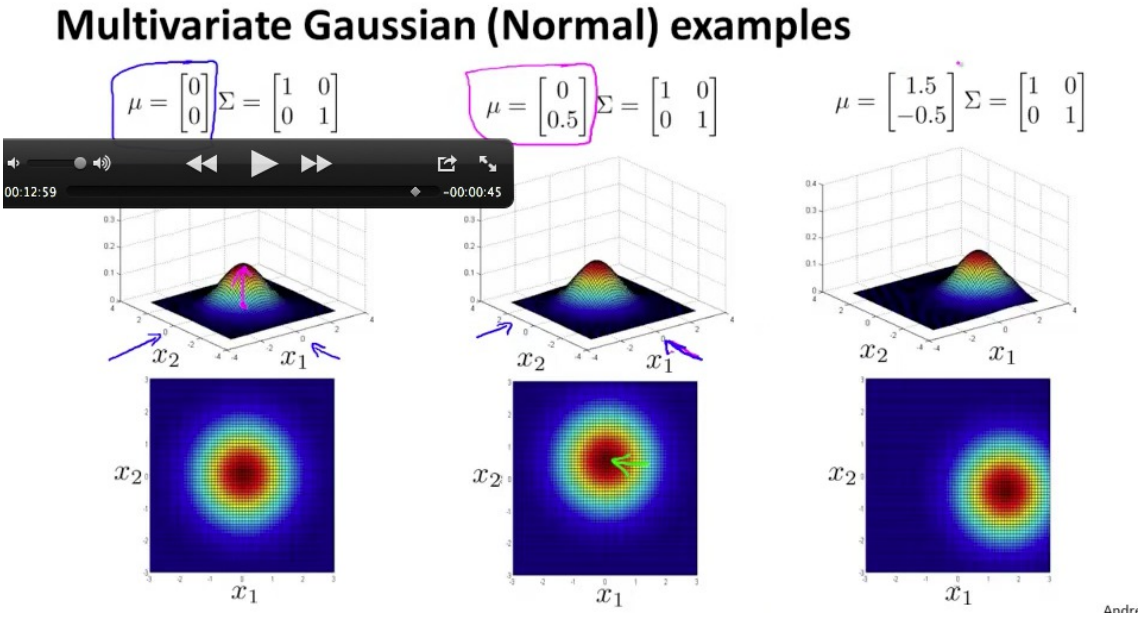


Fig. 11.10 A Gaussian where the center is translated

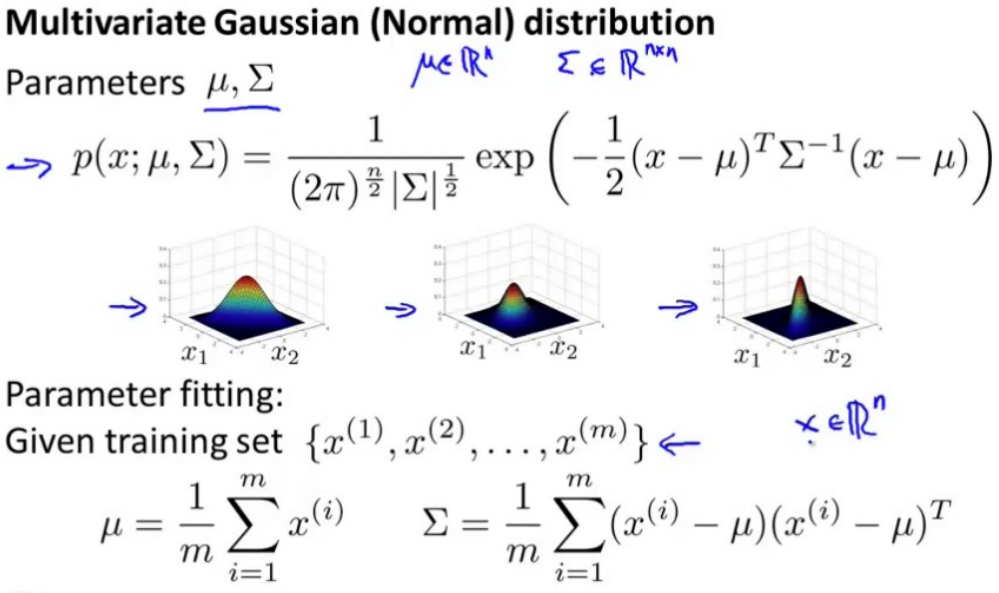


Fig. 11.11 Fitting parameters to a multivariate Gaussian.

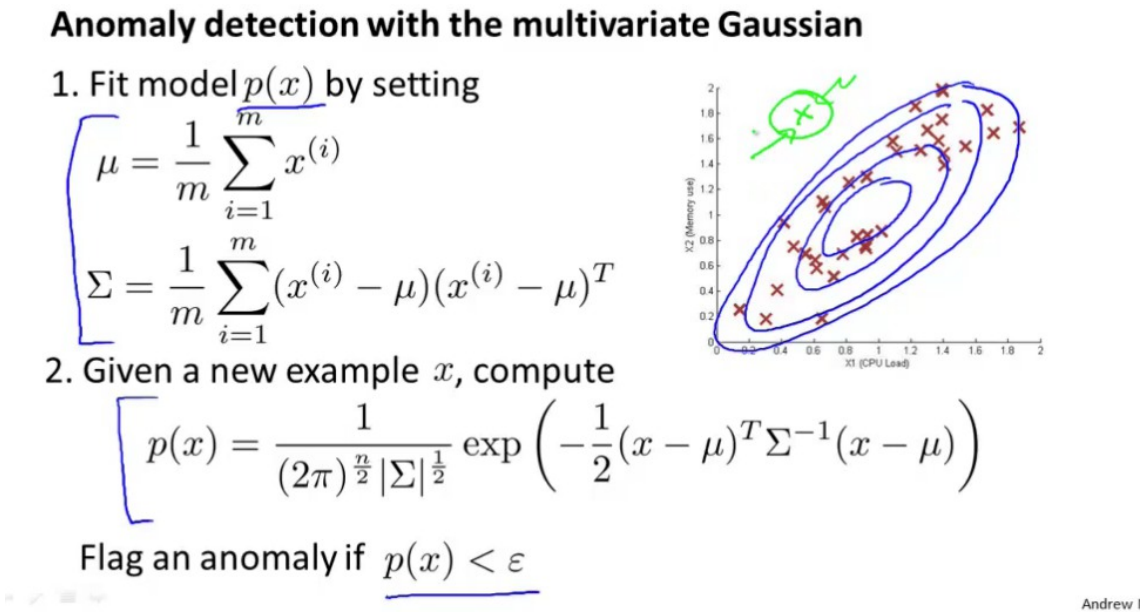


Fig. 11.12 multiparamalgorithm

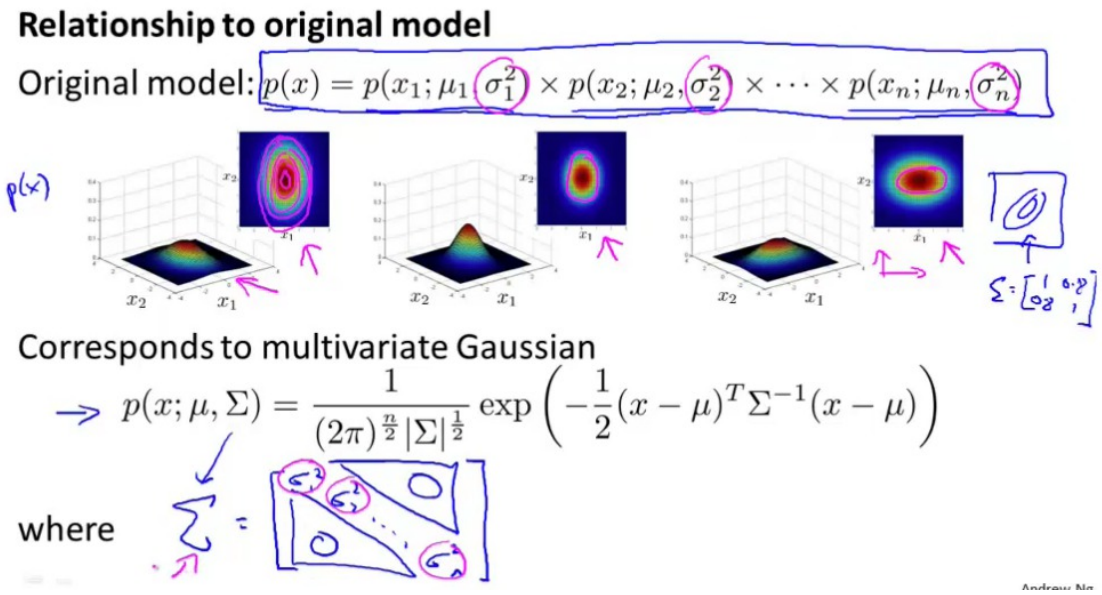


Fig. 11.13 The connection between a multivariate Gaussian and univatriate Gaussians (at least that's what I believe this plot is all about)

12 Recommender systems

Recommender systems is an important application of machine learning. It's a hot topic in Silicon Valley right now :) (netflix, amazon etc.) Performance on recommender systems give immediate feedback to the bottom line.

Reommender systems is not very big part of what happens in the machine learning seen from academia. Another thing is that the features that you choose. For some types of prolems there are algorithms that is pretty good at choose which features should be used.

To motivate the problem we will be using the problem of predicting movie rating. Movies can be rated from one to five stars (zero is ok in this case since that lets the makes the math easier)

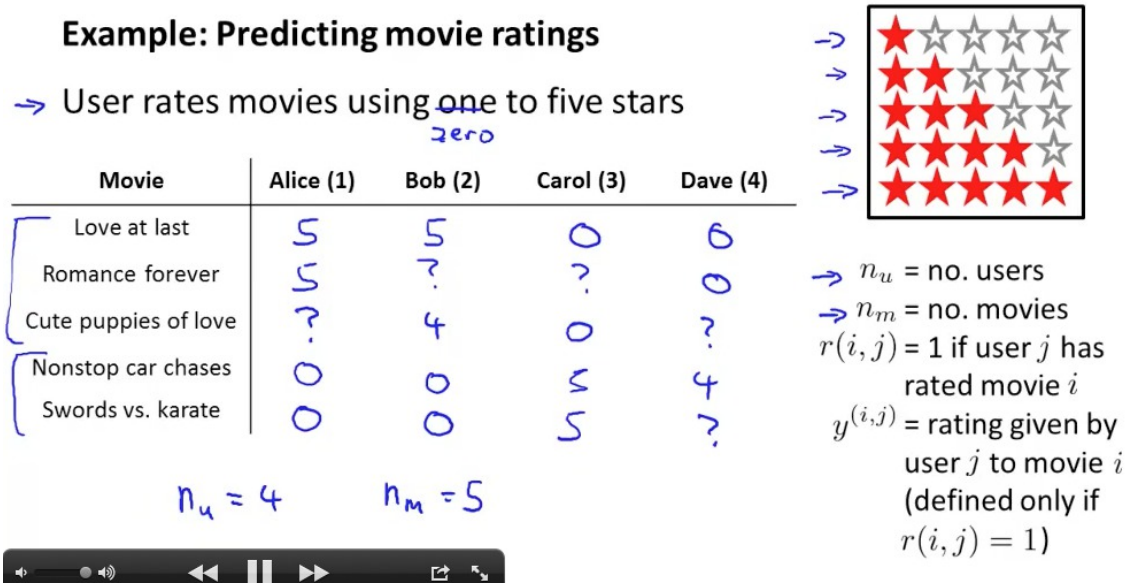


Fig. 12.1 A small table of movies and individual users' preferences

Alice and Bob seems to like the same type of movies, and so does Carol and Dave. We get rating info, but that rating isn't complete, so we have two arrays to represent both the presence and the value of ratings.

Rmz: *Should add the content of the figure as tex here*

The job in recommender systems is to figure out how to find interesting things for an user.

12.1 Content based recommendations

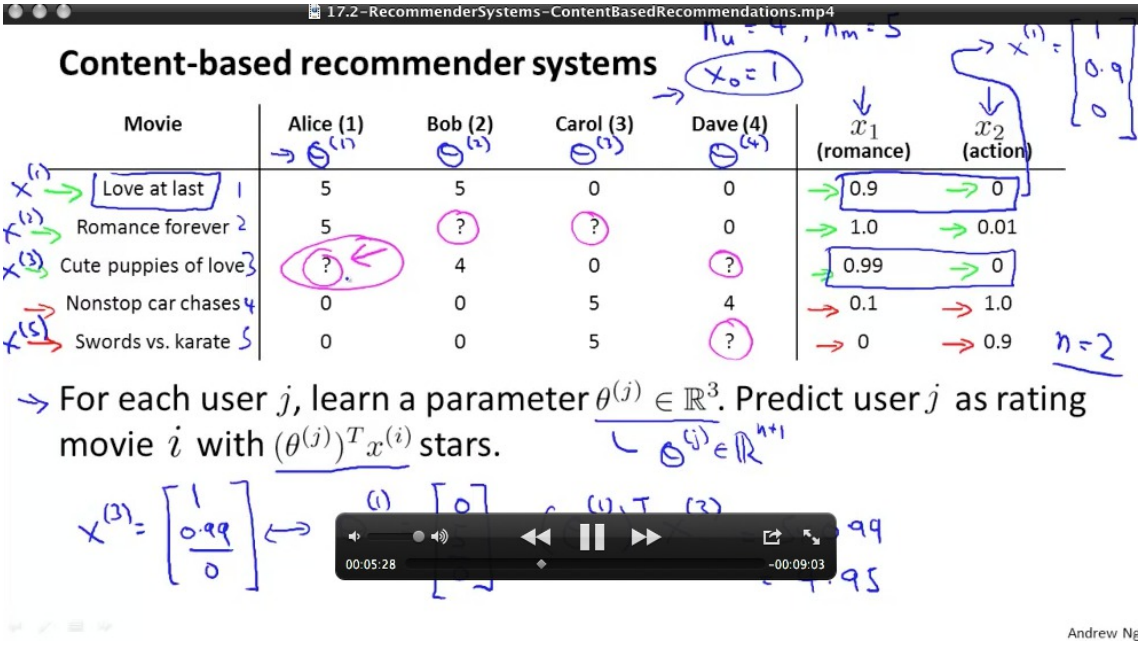


Fig. 12.2 A Content based recommender

In conent-based recommendations we add a bunch of features, in fig ?? we add "romance" and "action". With features like this, the movies can be represented as feature vectors. We add a feature always equals to one $x_0 = 1$ as an *intercept term*.

$n = 2$ since we have two features, nd we don't count the intercept term. We could treat the star-prediction problem as a linear regression problem and let it rip.

One formulation is:

- $r(i, j) = 1$ if user j has rated movie i, (0 otherwise).
- $y^{((i,j)}$ = rating by user j on movie i (if defined)
- $\theta^{(j)}$ parameter vector for user j.
- $x^{(i)}$ feature vector for movie i
- For user j, movie i, predict rating: $(\theta^{(j)})^T(x^{(i)})$

- $m^{(j)}$ no of movies rated by user j
- to learn $\theta^{(j)}$ we can solve this using linear regression. Use gradient descent or the normal equations in order to minimize:

$$\min \theta^{(j)} \sum_{i:r(i,j)=\frac{1}{2m^{(j)}}1} \left((\theta^{(j)})^T(x^{(i)}) - y^{(i,j)}\right)^2 + \frac{\lambda}{2m^{(j)}} \sum_{k=1}^{n} \left(\theta_k^{(j)}\right)^2$$

  We can add a *regularization* term after the sum (done in the equation above). We don't regularize over the bias term (as usual).

In the subsequent math we'll get rid of the constant term $m^{(j)}$ since it will not change the optimization result.

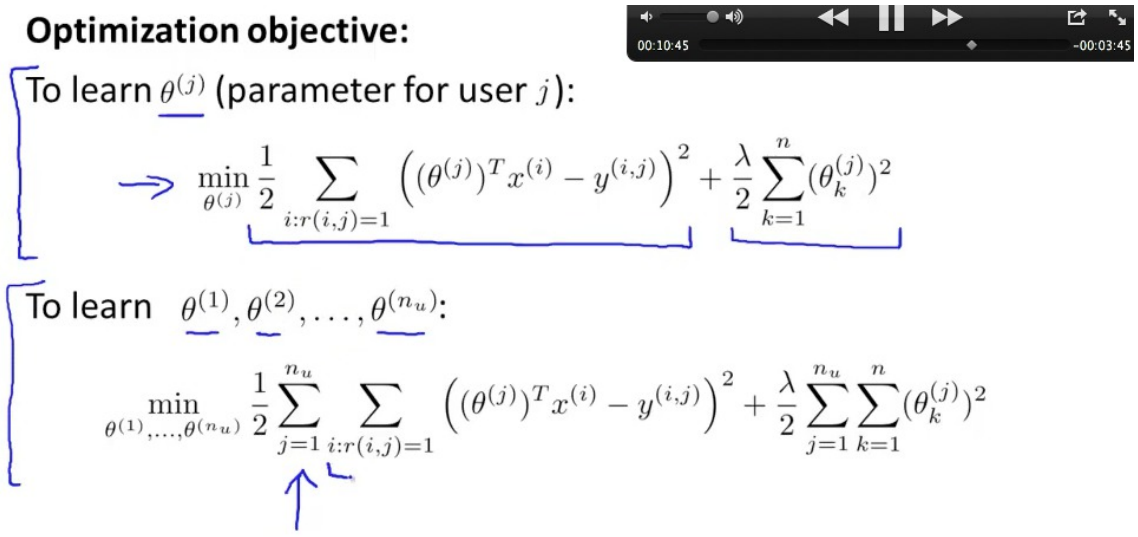


Fig. 12.3 An objective for a recommender system: Predict what the user is most likely to do next, and then suggest that

In figure **??** we calculate all of the objectives for all the users. We can put this together in a gradient descent update algorithm as depicted in figure **??**. The only difference wrt to linear regression is the 1/m term that is missing here. This objective can be plugged into *conjugate gradient descent* or *LBFJS* instead.

Content based recommendations require a content feature vector. For many types of movies we don't have those types of vectors, so we can use other, non-content based recommencations.

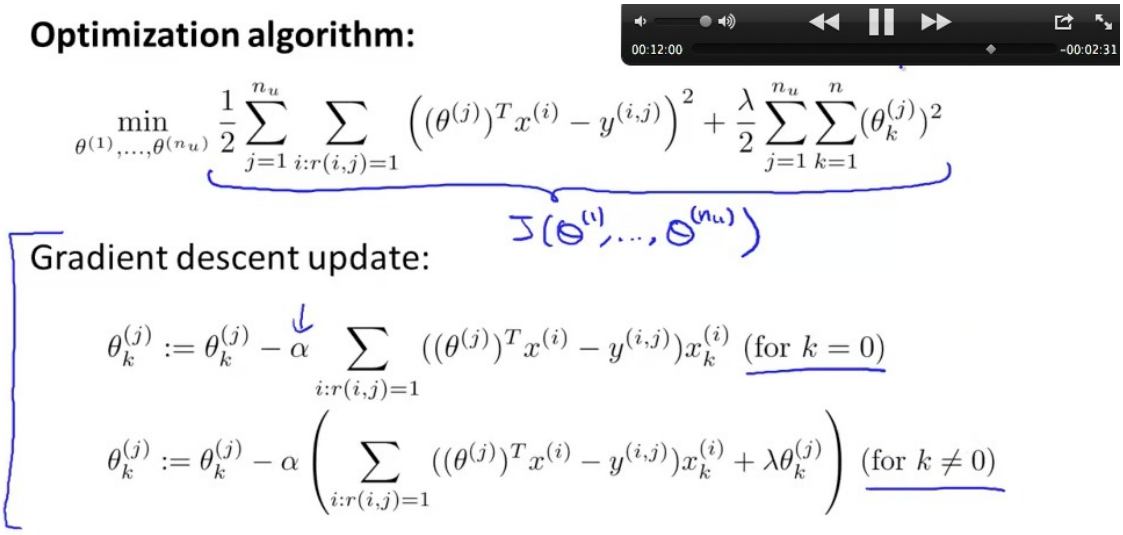


Fig. 12.4 Gradient descent for a recommender system

12.2 Collaborative filtering

Has feature learning as one of its atributes. Often it's hard to discover the features. The basic assumption is that the users has given us a vector that tells us what $\theta^{((j)}$ then we can infer what the feature vectors are for any given movie. How can we find the $\theta^{((j)}$s from the data set?

One algorithm is to randomly guess a θ to learn features for the movies, and then use that to bootstrap the algorithm, and then iterate iterate.

Rmz: *Isn't this an eigenvector finding method?*

The idea is that every user is helping the algorithm a little bit by letting the system learn features better.

There is an efficient way that doesn't need to go back between x and θ and solve it all in one go using the optimization criterion on the bottom of figure **??**. Basically the bnottom criterion is the two on top put into a single criterion and optimized for all the variables. The first sum on the bottom has a term that says $(i,j) : r(i,j) = 1$ which means that the inner sum is over all the pairs of (i,j) such that $r(i,j)$ is equal to one, which means that user i has made a rating of movie j. It's just a quantifier hooked up to a summation mechanism.

The two regularization terms for θ-s and x-es.

There is no intercept term in this system either. This is why we are learning all the features, because we if we need a feature that is always equal one, the algorithm will find one for itself.

The collaborative filtering algorithm:

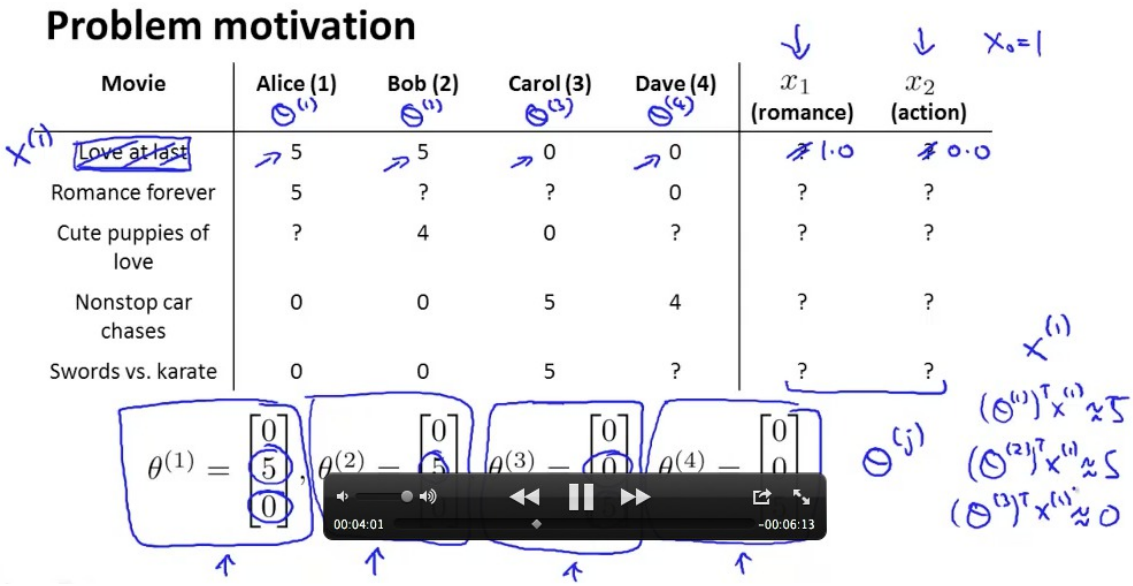


Fig. 12.5 Detecting features from user preferences

- Initialize $x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}$ to small random values (much like we did for neural networks).
- Minimize $J((x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$ using gradient descent, e.g. for every $j = 1, \ldots, n_u, i = 1, \ldots, n_m$: do

$$\begin{aligned} x_k^{(i)} &:= x_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^{(j)} + \lambda x_k^{(i)} \right) \\ \theta_k^{(i)} &:= \theta_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(j)} + \lambda \theta_k^{(i)} \right) \end{aligned}$$

  All of the parameters are regularized. The intercept term is not present so there is no θ_0 to treat speicially.
- For an user with parameter *theta* and a movie with learned features x predict a star rating of $\theta^T x$.

This is actually pretty decent algorithm both for modelling user behavior and for extracting parameters describing the movies.

12.3 Low rank matrix factorization

To vectorize collaborative filtering we need some tricks we can use. We can use the matrix Y in figure ??. This vectorized implemntation is called the *low rank matrix*

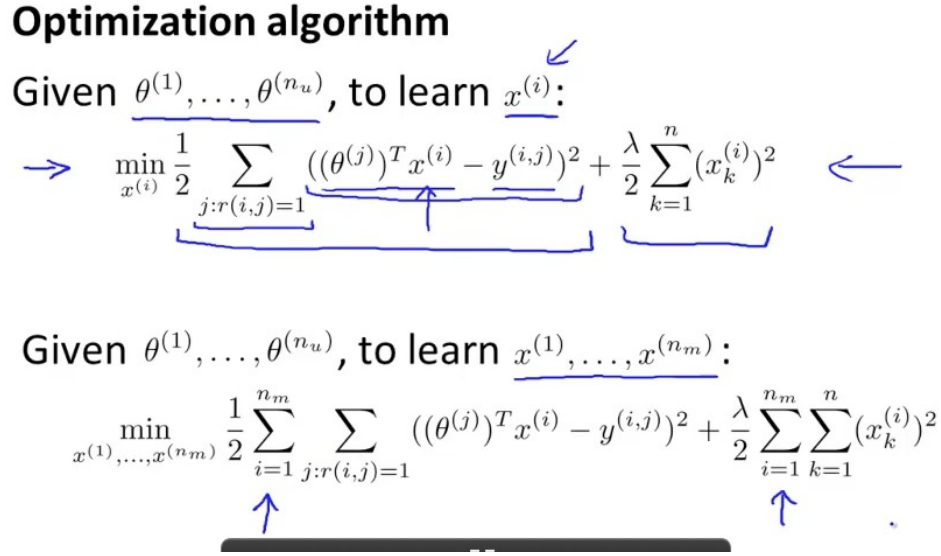


Fig. 12.6 Collaborative optimization

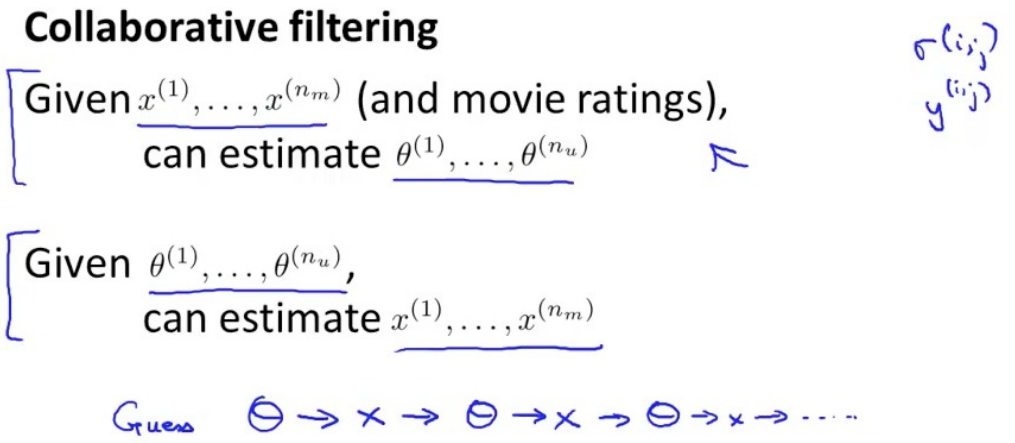


Fig. 12.7 A simple collaborative filter algorithm

factorization due to the fact that the matrix $X\theta$ is a low rank matrix.

Finally, having run the collaborative filter we can find related movies. It is often hard to figure out what the features exactly are, but usually the features that are captured will often find the most salient features fr figuring out why people like those movies. We can use this to measure how similar two movies are, we try to minimize some distance between the movies, we can use this to find e.g. the five movies that are the most similar to the one the user is seeing right now.

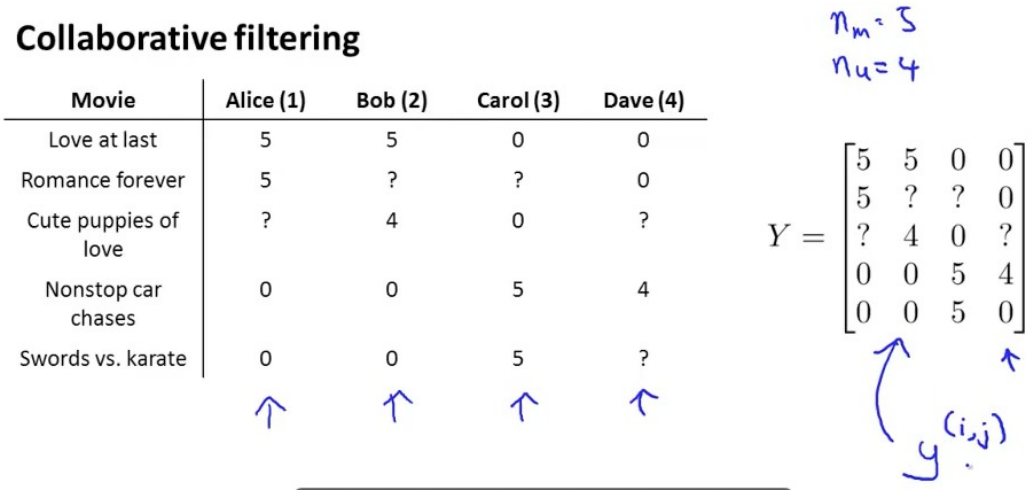


Fig. 12.8 Lower rank matrix? What does this mean?

12.3.1 Mean normalization

The mean normalization will usually let the algorithm run a little bit better. If we have an user without any ratings, if we include the user's θ in the expression to minimize, the regularization term will force all the terms to become zero and that will impose no penalty so that is what we get. We will then assume that the user with no ratings don't like anything. It seems that we are actually overfitting to the data we have, so perhaps some sort of smoothing will help us? That is where the idea of mean normalization comes into play.

First we calulate the averages of all the rows by first summing up all the nonzero but defined values, and the divide by the number of nonzero but defined values. Then we subract this value from all the values in the row. This we do for all the rows, then we have the "mean" of the rows subtracted from the row. We subtract for each row the average rating for the row. The question marks stay question marks.

The new mean normalized ratings is then used by the learning algorithm. This means that we need to add back the mean before returning the result. The results actually makes sense.

If we have no movies with no ratings, we can play with version of the algorithm to let columns get some mean values, but that may not be such a good idea :-)

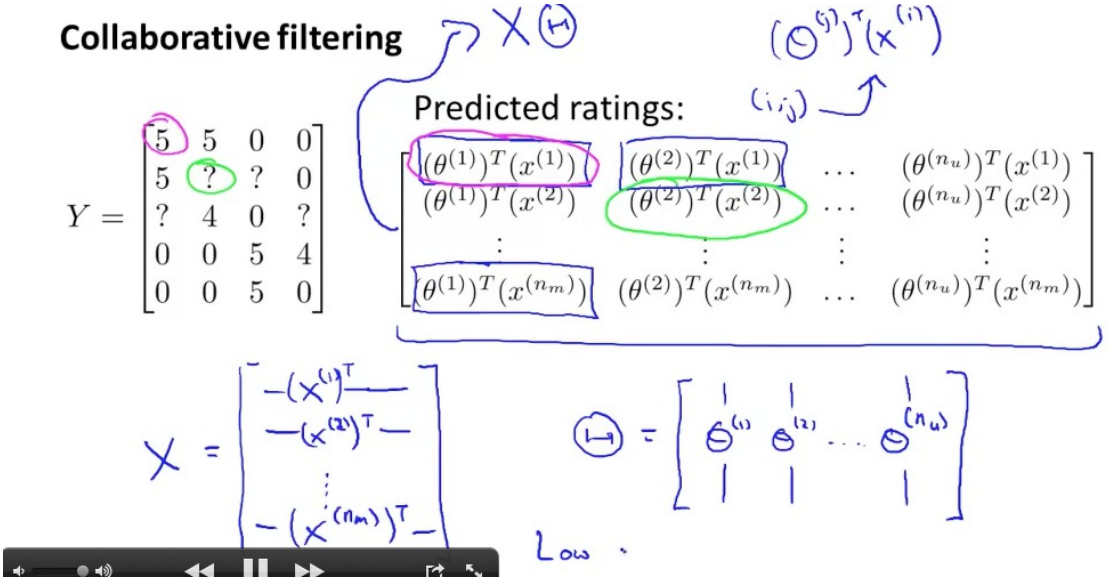


Fig. 12.9 Predicting rating from incomplete data

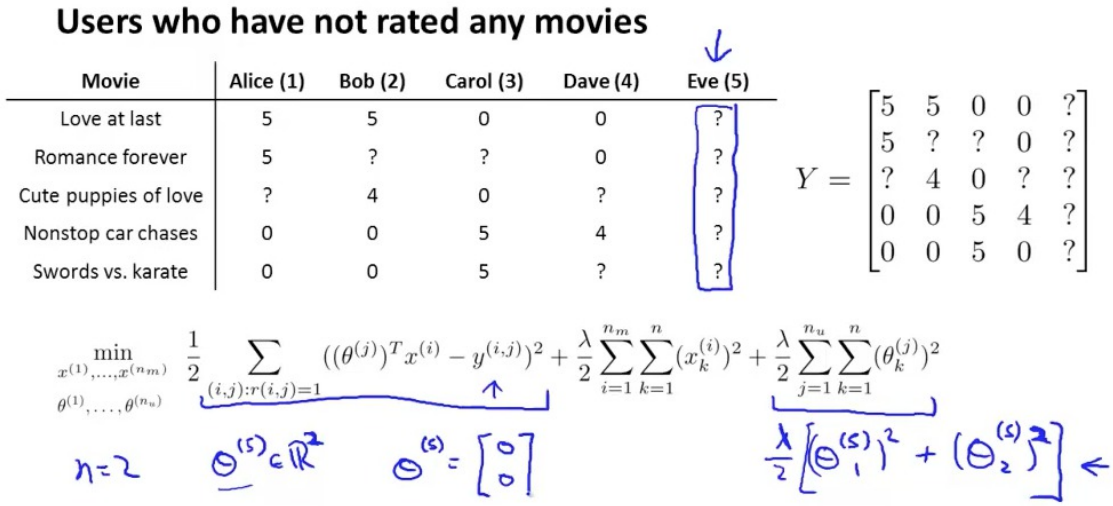


Fig. 12.10 How to handle users without any ratings

Missing imagefile file meannormalization

Fig. 12.11 Implementing mean normalization

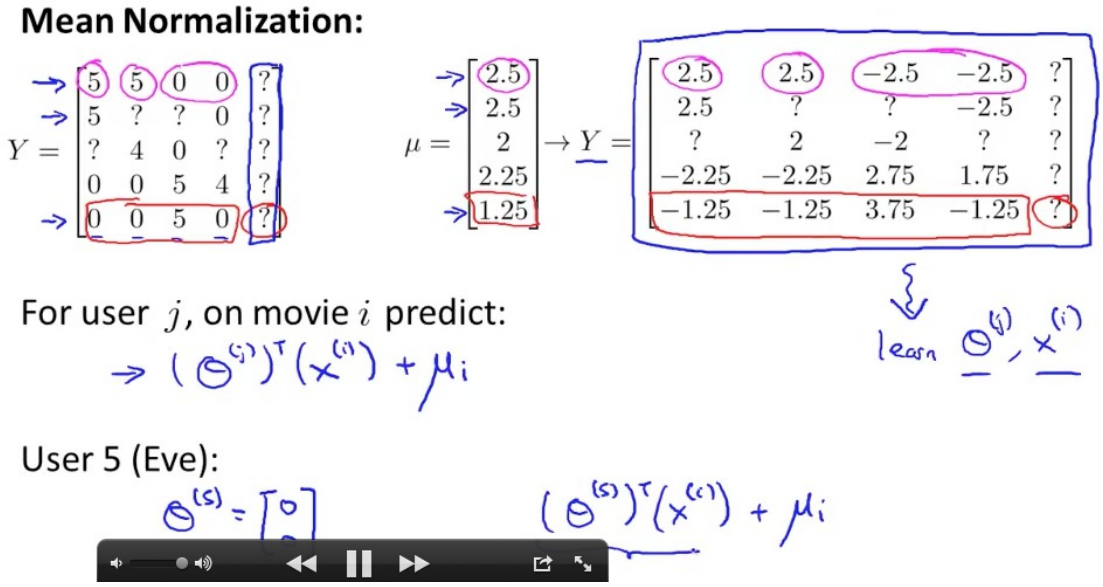


Fig. 12.12 meannormalizedresult

13 Large scale machine learning

Learning with big datasets. One of the reasons algorithms works better today than five or ten years ago is that we have massive datsets, now we will learn how to work with that. One of the best method of getting a good algorithm is to use a low-bias algorithm an a huge learning set. "It'w not who has the best algorithm, but who has the most data that wins".

Learning with large datasets comes with computationa cost. Working with hundreds of millions of records is very realistic. If you wish to use a gradient descent algorithm on a dataset like this, we need to need to compute a summation over a hundred million entries just to compute one step of gradient descent. We'll now learn both methods for replacing the algorithm, and more efficient ways to compute derivatives. We'll learn how to fit neural networks and regression models.

First we must ask us why not use a subset, e.g. a thousand samples? This is a good thing to check. The way to check this is to make plot the *learning curve*. If we have a high variance problem and it may be a good idea to add more samples. If we have a high bias case then it is unlikely that increasing the learning set will help us.

However, i a high bias method adding more features or adding more hidden nodes in a neural networks. We'll consider two algorithms that can be used to handle really big datasets, one is *stochastic gradient descent* and the other is *map reduce*.

13.1 Stochastic gradient descent

For many learning algorithms we have a cost function and then use gradient descent to compute minima. When we have very many variables gradient descent becomes a computationally expensive algorithm. We will now discuss a variants of gradient descent that scales much better.

The cost function of J is a bowl-shaped function (convex). Stochastic gradient descent can be used both on linear regression and neural networks etc.

Gradient descent will follow the gradient to the minimum. If N is large, computing the derivative is costly.

The algorithm we have been using so far is called *Batch gradient descent*. If we use this dataset for really large datasets, we have to stream all the data through

memory for each iteration. There just isn't enough room in memory to keep it all there, and having done it once, we then have to do it again for the next iteration. It's fairly costly.

The summarized batch gradient descent algorithm is based on the error function:

$$J_{\text{train}} = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

And then to iterate this step:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum \left(h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)} \right)$$

for every $j \in \{0, \ldots, n\}$.

In contrast the *stochastic gradient descent* algorithm uses these steps:

$$\begin{array}{lcl} \text{cost}(\theta, (x^{(i)}, y^{(i)})) & = & \frac{1}{2} h_\theta(x^{(i)}) - y^{(i)})^2 \\ J_{\text{train}} & = & \frac{1}{m} \sum_{i=1}^{m} \text{cost}(\theta, (x^{(i)}, y^{(i)})) \end{array}$$

The cost of an error for a training set air is half the square of the error produced by the hypothesis. The cost function measures how the hypothesis is doing on a single sample. We can now define the stochastic descent algorithm.

1. Randomly shuffle the dataset (standard reprocessing step).
2. repeat for all the training examples for $i \in [1, \ldots, m]$:

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

for $j \in [1, \ldots, n]$. This thing makes use of the fact that:

$$\left(h_\theta(x^{(i)} - y^{(i)} \right) = \frac{\partial}{\partial \theta_j} \text{cost} \left(\theta, (x^{(i)}), j^{(i)} \right)$$

The algorithm is scanning through the training examples, and then it will take a small "gradient descent" step and take another little step through the parameter step and so on for each training example.

This view of stochastic descent motivates why we want to shuffle the data set, since we want it to be random :) This will speed up the convergence a bit.

Missing imagefile file stochasticgradientdescent

Fig. 13.1 Convergence pattern for a stochastic gradient descent algorithm

Another different is that we make progress in fitting tot he parameters before we can make a little progress towards a global minimum, we get progress for each training example we look at.

Batch gradient descent takes a reasonably straight line towards the maximum, but stochastic gradient descent will have a somewhat more erratic path towards the minimum. It wil not necessarily have a monotonous path towards the global minimum. The path will look more random. Furthermore the algorithm doesn't really converge to the global minimum and stay there, but this isn't a problem since it will still end up in a region pretty close to the global minimum. As a practical matter, the result will be useful in most cases. The inner loop may have to be iterated one to ten times. For really large datasets it may be sufficient to have a single dataset through the dataset.

Contrast this to batch gradient desscent, where we would have to make a lot of steps. Implementing this algorithm will let us scale the algorithm to much bigger datasets and thus get better performance.

13.2 Mini batch gradient descent

Missing imagefile file minibatchdescent

Fig. 13.2 minibatchdescent

The *Mini batch gradient descent* is somewhere in between the *stochastic gradient descent algorithm* and the *batch gradient descent algorithm*. In the mini-batch gradient descent we use b samples in each iteration instead of just one as in stochastic GD.

Typical choices for b is 10, and other typical choices may be anywhere in the range $2 - -100$. The idea is that we get (for instance) ten examples $x^{(i)}, y^{(i)}, \ldots, x^{(i+9)}, y^{(i+9)}$. Then we perform a gradient descent update (assuming b=10):

```
%There is a double superscript below, but
% what should it -really- be?
\[
   \theta_j = \theta_j - \alpha\frac{1}{10}  \sum_{k=i}^{}^{i+9}  \parens{h(x^
\]
```

The gradient term over the ten examples (the batch size) .

3. Iterate $i := i + 1$.

Again we don't need to scan through the entire training set before making progress in modifying the parameters.

How about mini-batch gradient descent and not stochastic gradient descent? And the answer is *vectorization*. By using appropriate vectorization we can partially parallellize the gradient computation and that can give a noticable improvement. One problem is the variable b that we now need to keep track of.

13.3 Convergene of stochastic gradient descent

We'll need to figure out how to manage convergence, and how to manage the learning rate α. In batch gradient descent we could observe that the error was decreasing. We can't do that for stocastic gradient descent since we need to scan through the entire training set to compute the cost function.

So for stochastic gradient descent we instead compute:

$$\text{cost}(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} h_\theta(x^{(i)}) - y^{(i)})^2$$

before updating θ using $\left(x^{(i)}, y^{(i)}\right)$. As the algorithm is scanning through the training set, we compute how well the hypothesis is doing on *t*he training example we are working on.

Then every 1000 iterations (or so) plot the $\text{cost}(\theta, (x^{(i)}, y^{(i)}))$ averaged over the 1000 examples. This doesn't cost much and we can keep track of how the algorithm is doing. See some examples in ??. With smaller learning rates we may be able to get better results. Smaller learning rates gives smaller oscillations, sometimes it doesn't matter much ;) By increasing the number of iterations to smooth over the curve will get smoother. Sometimes it doesn't look like the algorithm is learning. It may be that averaging over larger samples will then show a trend indicating that the algorithm is learning after all. However, it may also be flat, and that is an indication that the algorithm isn't working. If you se that that the error rate is learning, then you are observing that the algorithm is *diverging* and you should use a smaller learning rate α.

The learning rate is usually held constant, but if you want the stochastic gradient descent to actually converge on the global minmum, you can slowly decrease the α over time to help with convergence, e,g.:

$$\alpha = \frac{\text{const1}}{\text{iterationNumber} + \text{const2}}$$

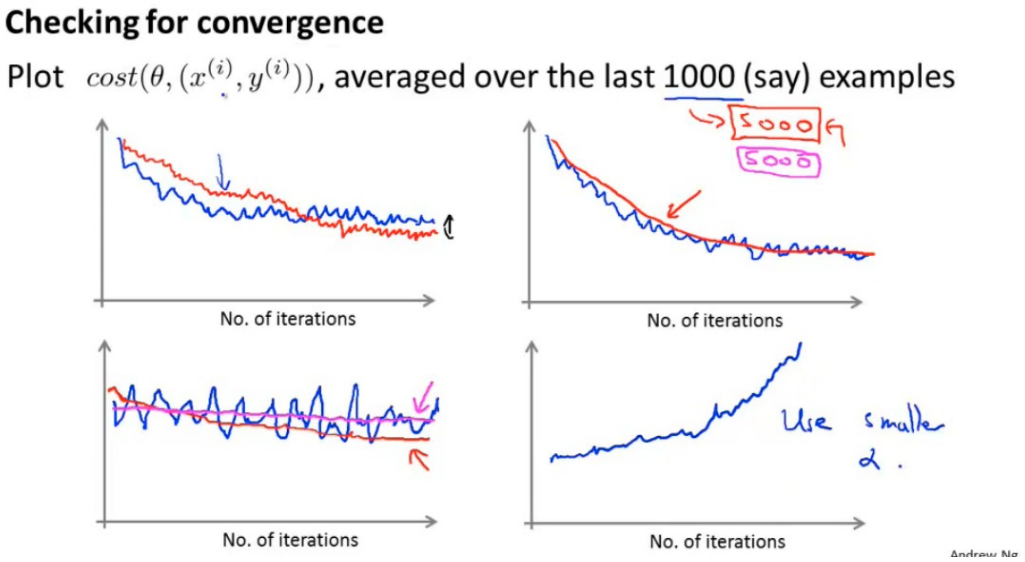


Fig. 13.3 gradientdescentconvergence

The constantants are parameters you may have to play a bit with. They may help, but they also increase the number of parameters you have to mess with so it makes life harder. This process is usually not used.

13.4 Online learning

Continous stream of data and we wish to have an algorithm to learn from this. Many large internet companies use this type of techniques, often called *online learning algorithm*s to analyze data.

Assume a case where you are running a site that sells packages. Sometimes the uses wish to use a shipping service (y=1) and sometimes not (y=0). We wish to learn $p(y = 1|x; \theta)$ to optimize prize. The x includes the price we ask for. We can use logistic regression (or any of the other algorithms, but let's use lr now).

We will then repeat forever, get an (x, y) pair from the user. The algorithm then update the parameters θ using (x, y):

$$\theta_j := \theta_j - \alpha(h_\theta(x) - y) \cdot x_j$$

for $j = 0, \ldots, n$).

We are actually discarding the notion of having a learning set, we're just processing the samples as they come in. If you are running a really high volume website this type of algorithm is really useful. Data is essentially free, so we don't have to

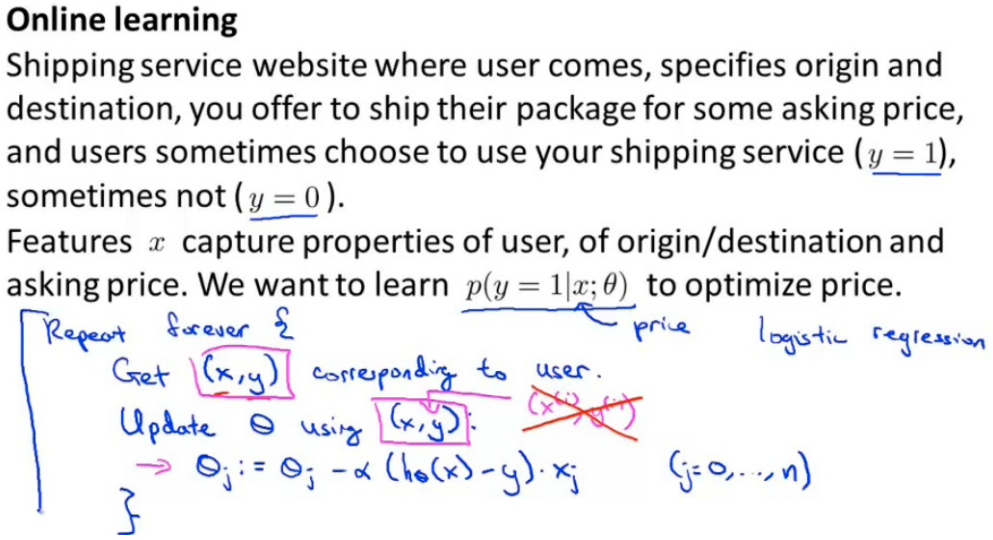


Fig. 13.4 Onlne learning. Update using every data point as they arrive.

look for training examples. If we have fewer users it may be smarter to tuck the dat away in a data set and run it.

The online approach has the advantage of being able to adopt to changing user preferences. The algorithm is following trends as they happen.

Another appliction of online learnign is in product search. Assume we sell mobile phones. We might get 100 phones that matches the search "android phone with 1080p camera", but we will return ten results. We could have an algorithm to figure out which those ten phones should be. We may let clickthrough for users represent an $y = 1$ and otherwise $y = 0$. We can then use this data to estimate $p(y = 1|x; \theta)$. This is the problem of predicting ther *click through rate* (*CTR*). Since we can estimate the CTR, we can now just select the ten phones with the highest estimated CTR.

Every time we give the user ten choices, we actually get ten learning examples. We get them, run ten steps of gradient descent, and then throw the dat away.

There are other things we can do instead of estimating phones, we can choose special offers to show the user, customized selection of news articles, product recommendations etc.

Any of these problems could be formulated as a standard machine learning problem with a training set, but if you get too much data, there really isn't any point in saving away the training set, it's better just to use the data on the fly.

The algorithm used in *online learning* is really very similar to the *stochastic gradient descent* algoritm

13.5 Map reduce

Some machine learning problems are just too big to run on a single computer. The map/reduce approach is very important even if we won't spend much time on it than stochastic gradient descent. The reason less time is use is that because it's easier to explain.

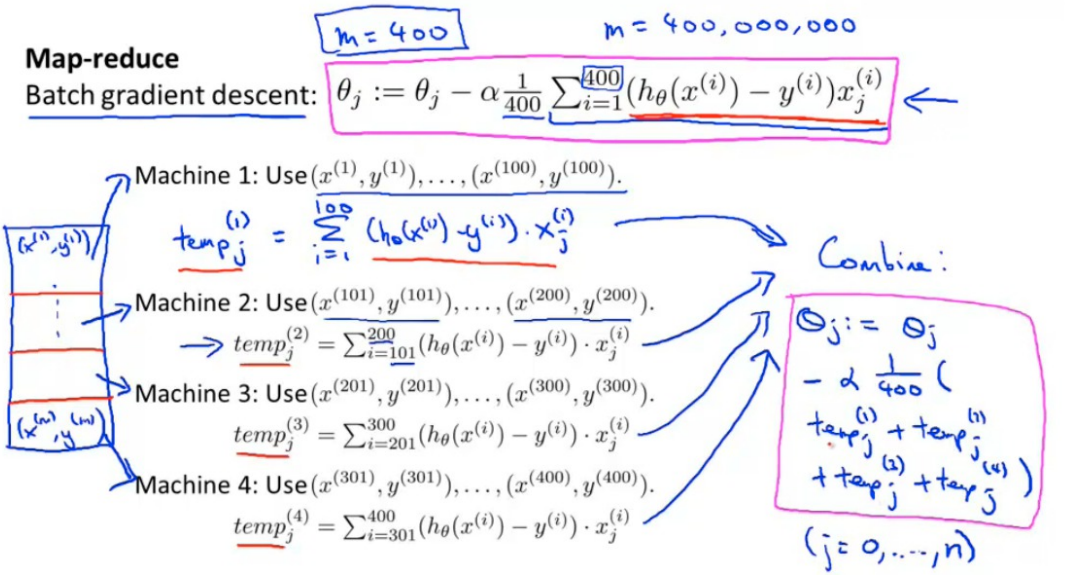


Fig. 13.5 Map/Reduce: Divide and conquer :-)

The map/reduce idea is to split the computational task into different partitions. Assume that we have four computers, we will split our training set into four pieces. The first machine will just use the first quarter of the training set, and similarly for the rest of them. If we're working on a *batch gradient descent* algorithm, then our basic step is:

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}$$

When we split this up we get temporary variables that summarize parts of the big sum:

$$\text{temp}_j^{(1)} := \sum_{i=1}^{100} (h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}$$

and then to do a similar things for the three other partitions. Then we collect the temporary variables and update like this:

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{4} \text{temp}_j^{(i)}$$

Then we do this separately for all the $j = 0, \ldots, n$.

This is exactly identical to batch gradient descent, but it's parallell.

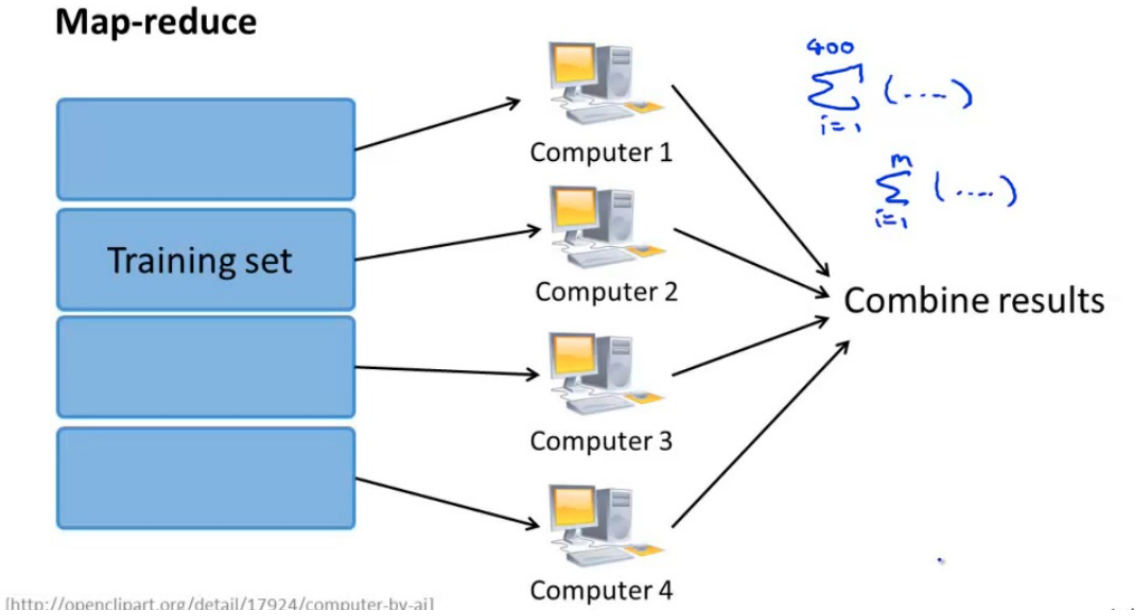


Fig. 13.6 Dividing the learning task between multiple processors

In practice we we get less than an n times speedup (latencies, etc.)

Many learning algorithms can be expressed as computing sums over functions of the training set, and when they do they can be implemented using map/reduce.

Assume that we wish to use an advanced optimization function (LB.. etc.) calculating the cost function is expensive. We can then split the calculation of the error functions and partial derivatives over many machines.

A single computer with multi-core CPUs can also be a nice place to use map/reduce. We can then split the job on diffent cores within the same computer. The advantage of thinking about map/reduce this way is that we don't have to think much about network latency.

One last caveat on using multi-core machines. Some libraries can automatically paralleize over multiple cores, sometimes you can just implement your algorithm in

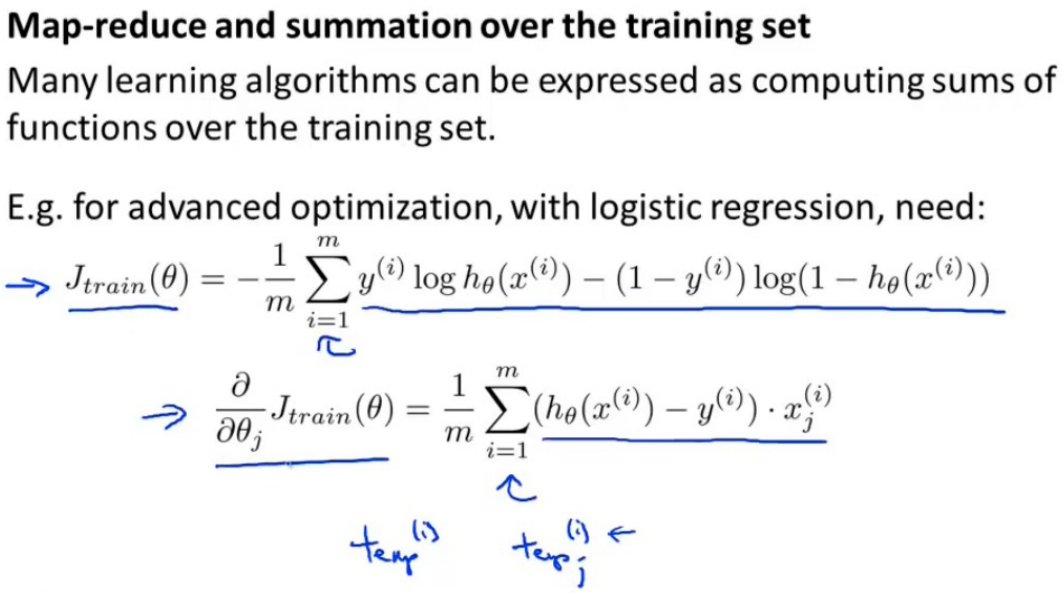


Fig. 13.7 Map/Reduce requires some reformulations in terms of partitioning of the data and operations (e.g. sums) over those partitions.

a vectorized fashion and the library will take care of some of this automatically.

Hadoop is a nice system for parallelizing algorithms to make them run on really large datasets.

14 Application example: Photo OCR

Fig. 14.1 Recognizing text in images is a hard task

Big example. Architecture and software engineering. Computer vision and artificial data synthesis. Photo *OCR* or *Optical Character Recognition* lets a computer look for characters in pictures to identify characters. It dos so, and does it in several steps. First to identify where the characters are, and then to scan the texts and come up with good interpretations. While OCR for documents is fairly well established, Photo OCR is still considered a hard machine learning problem.

The pipeline is:

- Character recognition.
- Character segmentation.
- Character classification.

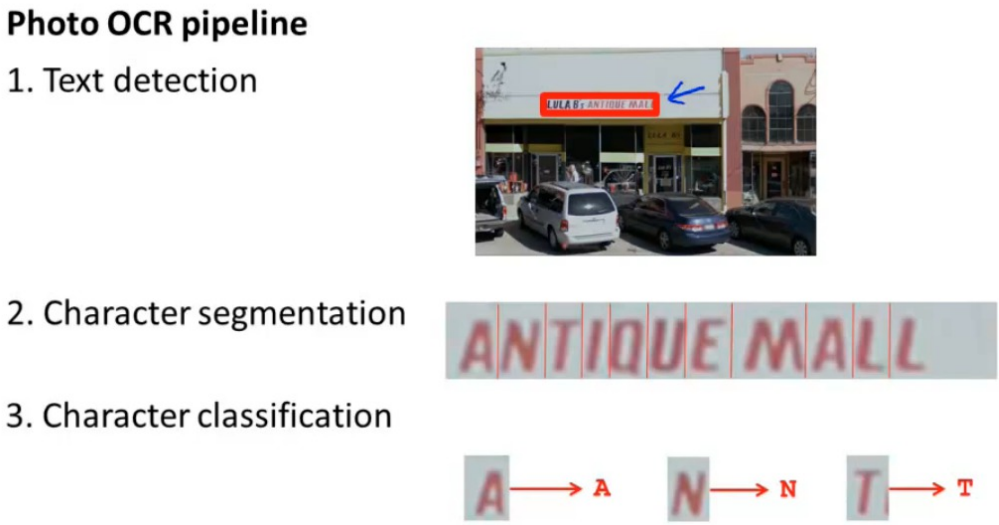


Fig. 14.2 A processing pipeline for processing text in images

Missing imagefile file ocrpipelinegraphics

Fig. 14.3 ocrpipelinegraphics

Some pipelines also has spelling correction, but we won't do much work on that now. A system like this is a *machine learning pipeline*. Pipelines like this are common in machine learning.

When designing a machine learning system designing the pipeline is ofthen one of the most important decision that will be made. If you have a team of engineers working on this it can easily be one to five engineers working on each of the subtasks.

In complex machine systems the concept of a pipeline is pretty pervasive.

14.0.1 Sliding windows

The first component in the OCR pipeline we will look at a *sliding window classifier*. Detecting text is difficult among other things since the text slots have different aspect ratios. We'll start loking at a simpler case, *pedestrian detection* and then apply the techniques we develop there in the text detction case.

In pedestrian detection we wish to find the individual pedestrians. The aspect ratio of most pedestrians is about the same.

To build a pedestrian detector, we can choose a bunch of positive and negative examples and a standard image size. Several thousand examples will probably be

Fig. 14.4 Text detection v.s. pedestarian dectection. Detecting pedestarians is a much simpler task since they by and large all have the same orientation and aspect ration, whereas text usually does not.

good. Some neural network, e.g. a neural network can then be used to classify an image patch to detect if it contains a pedestrian or not. The algorithm is to extract a rectangle of the correct aspect ratio and run it across the image to search through. The distance one steps is called *stepsize* or *stride*, a stepsize of one gives the best result, but is computaionally expensive. A stride of eight or even higher will be faster but have worse error rates. Then then run larger image patches, resizing it into the standard size (say 82x 36) and then run that through the classifier. ANd then do it at an even larger image. This will detct squares with pedestrians in a window.

FInding text regions is harder, but it starts the same way. Start with a bunch of patches of images that contains text (and not) and apply that to a classifier algorithm. As we run the sliding window over the target window, we detect regions that may contain texts. An input image is shown in fig. ?? and the corresponding output of the classifier is depicted in figure ??. The white patches on the result is to indicate that the clasifiers thinks it might have found text.

We are not quite done yet, since we want to draw rectangles around the texts. To do this we first apply an *expansion operator* on the output from the classifier and get an image like the one in figure **??**. The expansion option basically smear it a bit by using a rule that will color an expanded image white if it has a white pixel nearer than five pixels. We can then look at all the white regions and apply a heuristic to expand rectangles with reasonable aspect regions and then we're

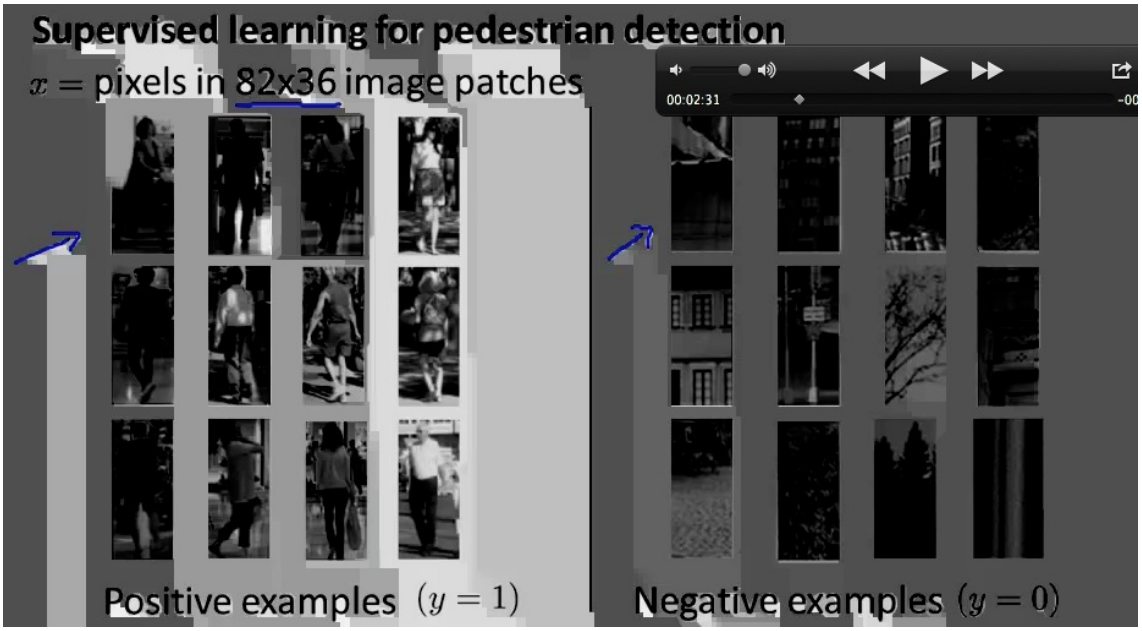


Fig. 14.5 Some examples of pedestarians, used as input for machine learning.

done. This classifier actualy misses some text that is hard to read (written against a transparent window).

We can no use later stages in the pipeline to do character segmentation. We can again use a supervised learning alorithm. The first thing we should do is to lok for splits between characters. We

so we train a classifier to detect positive and negative examples, we can then run this on the text detection system and figure out what are individual characters and what is not. We only need to slide the classifier over a single row so it will be a one dimensional sliding window. This will give us a set of locations where we should split the image into subimages containing single characters.

Finally we do character classification, and we know how to do that :-) Standard a standard supervised learning algorithm to classify which characters the images represent.

That is the photo OCR pipeline.

14.1 Getting lots of data: Artificial data synthesis

A low bias algorithm with a whole lot of training data is a really good way of getting a good learning algorithm. But where do we get all the training data? If we have a lot of raw data that is good, but if we don't we can cheat and make

Fig. 14.6 Image to search for letters in using a sliding windows algorithm.

them ourselves using *artificial data synthesis*. It can't be applied to every problem, but if it applies to the problem at hand, it is an easy way to get a huge data set. There are two basic techniques that are used, crating new data from scratch, and to use a small training set and amplify that into a larger training set. We'll look at both of those ideas.

Consider the *artificial data synthesis idea*. If we go out into the world and collect a bunch of images we can use that. Modern computers often has a huge font library stored in it. The are font libraries that can be used to generate training examples by taking characters from fonts and past it onto random backgrounds we may get something that looks a lot like the set of images to the right in figure ??. It is a bit of work to generate this data, but it can certainly be done. Use litt blurring, affine (shearing/rotation/scaling) operations, and you get a training set. If you do a sloppy work when you make the artificial data it won't work very well. Using this technique you have an essentially unlimited supply of labeled data. We just generate new data from scratch.

The other approach is to synthesize data by introducing distortions. In figure ?? this is done by taking a real image, and then warping it using sixteen different "warpings" :-) i.e. distorting filters. Again, in order to do this for a particular

Fig. 14.7 The output from the sliding windows character detector run on the picture in fig ??

application this takes work. The warpings must be reasonable. In speech recognition something entirely different will be necessary :-)

A word of warning: The distortions that are chosen must be ones that are meaningful for the domain in question and ones you might expect to see in the test set. Just adding random noise will usually not be very helpful.

The process of artificial data synthesis is a bit of an art :-)

As always, make sure you have a low bias classifier before spending the effort (plot learning curves). Keep increasing the number fo features until you actually have a low bias classifier, and only then you should start working with increasing the training set.

A question that is often asked is "How much work would it be to get ten times as much data as we currently have?" It's a very good question to ask. Often the answer is "it's not that hard", and often if you can get ten times as much data that is often a good way to make the algorithm better. Artificial data synthesis is one way (both from scratch and distorting existing data). Another way is to collect/label data yourself. It is useful to do the math on how hard it is to get and label the examples yourself. If it takes ten seconds classify a datum, and you have say a thousand items then it will take a ten thousand seconds, which is a bit about

Fig. 14.8 A summary slide for the text detection using a sliding window and a expander algorithm

three hours of work. It is often surprising to see how little it can be to get a lot more data and give the learning algorithm a huge boost in performance.

Finally we can "crowdsource". We can even hire folks (e.g. through the *Amazon Mechanical Turk*) to do the labelling for us. It can be done. It is often quite a bit of work to get high quality labelling.

Remember to check the algorithm with learning curves, and figure out how hard it is to get ten times as much learning data.

14.2 Ceiling analysis

Ceiling analysis: How to pick the part of the pipeline to work on next. Consider the process in fig ??, where should you allocate resources? Each of the boxes in figure ?? represents somewhere we can add resources. Again the idea of a *single row number evaluation metric*. Assume that we have the character recognition metric for the OCR example. If the system has 72 percent accuracy. We will now consider each module and for every test example we'll provide the correct output. We'll just manually tell it where the results are. We will simulate that we have a system with perfect accuracy. It's easy, instead of letting the algorithm do the work we do it for it. We then do the same type of thing for the other modules. We'll give it both perfect text detection an character detection. All the time we

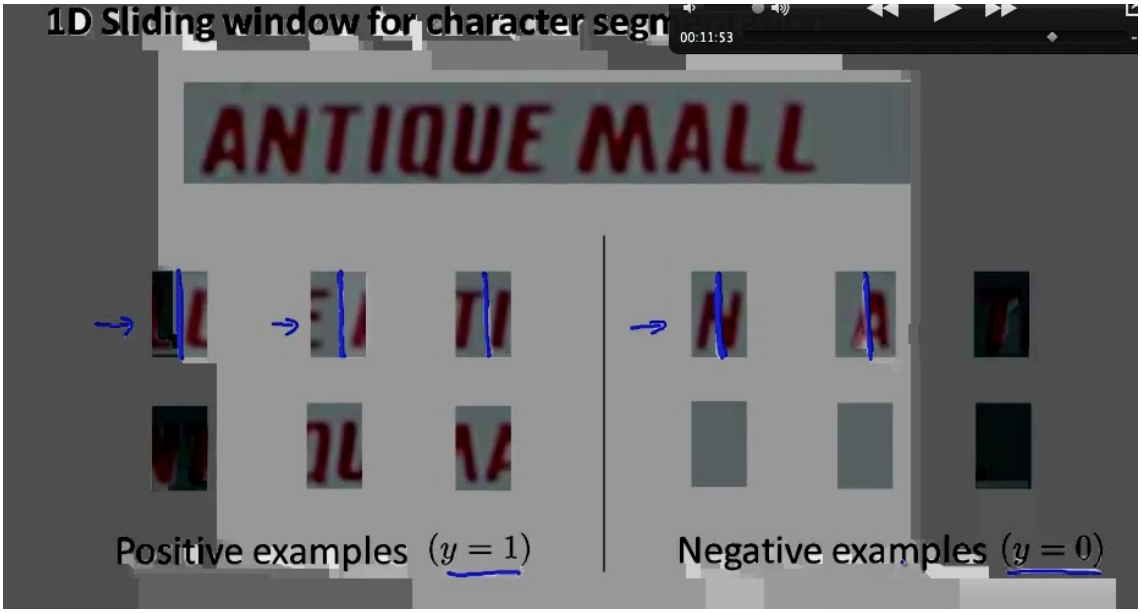


Fig. 14.9 Character segmentation using machine learning

are looking at the accuracy for the entire system.

We can now see which parts gives us the best upside. In the example in figure ?? the text detection system has a potential upside of seventeen percent, the other parts much less. This indicates that working on text detection (in this example) is the place where we can get the most bang for the buck.

We find the "ceiling" or the upper boundary for how much gain we can get by making a module into a perfectly functioning component.

Let's consider another example, in this case *face recognition from images*. The example is artificial in the sense that this isn't how face recognition is done in practice.

We have a pipline consisting of a background remover, face detector, eye segmentation, nose segmentation, mouth segmentation. We then feed all of this into a logistic regression classifier and that gives us a label for the person. This pipeline is probably too complicated for a real face recognizer, but it's ok for a ceiling analysis example. e can break down the system.

Cautionary story: There were a company that let two engineers spend a year and half to remove backgrounds, but it didn't make much difference to the overall performance. Doing a ceiling analysis beforehand they could have prioiritized otherwise.

Our time as deveopers is very important. It's important to focus our time on the component where we can make the most impact (gradient descent:-). Prof. Ng's

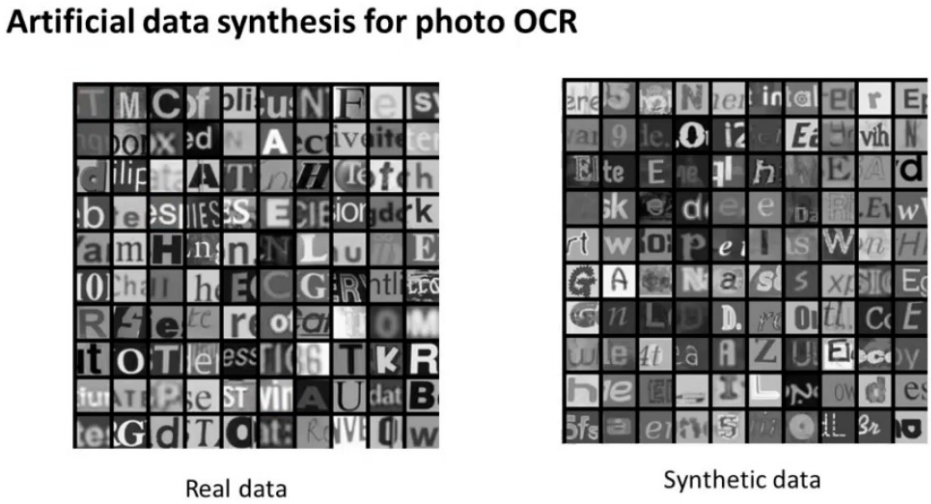


Fig. 14.10 Real learning data v.s. synthetic learning data

experience has shown him that he shouldn't trust his gut feeling very much. It's better to do a ceiling analysis to focus the effort.

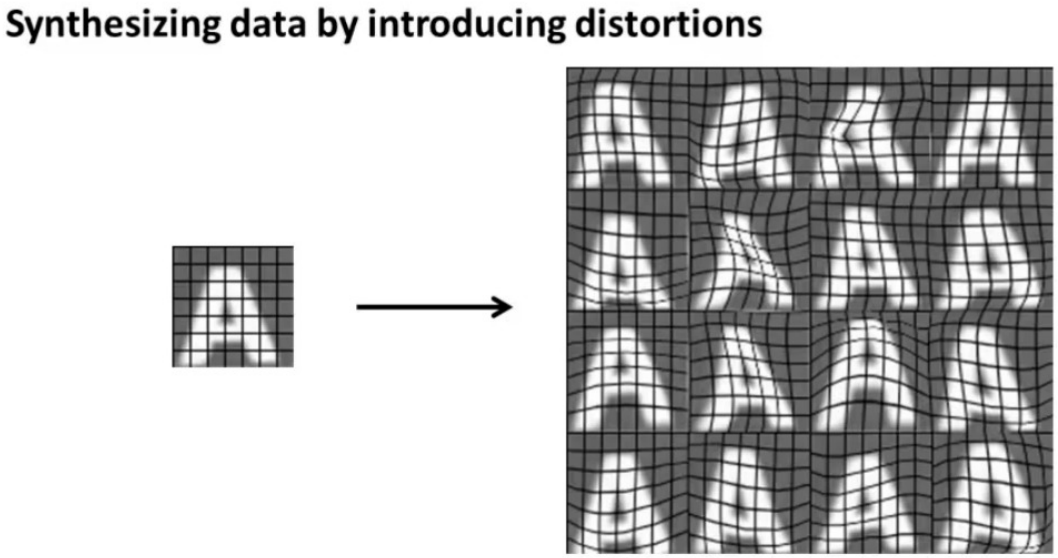


Fig. 14.11 Warping characters to generate more learning input

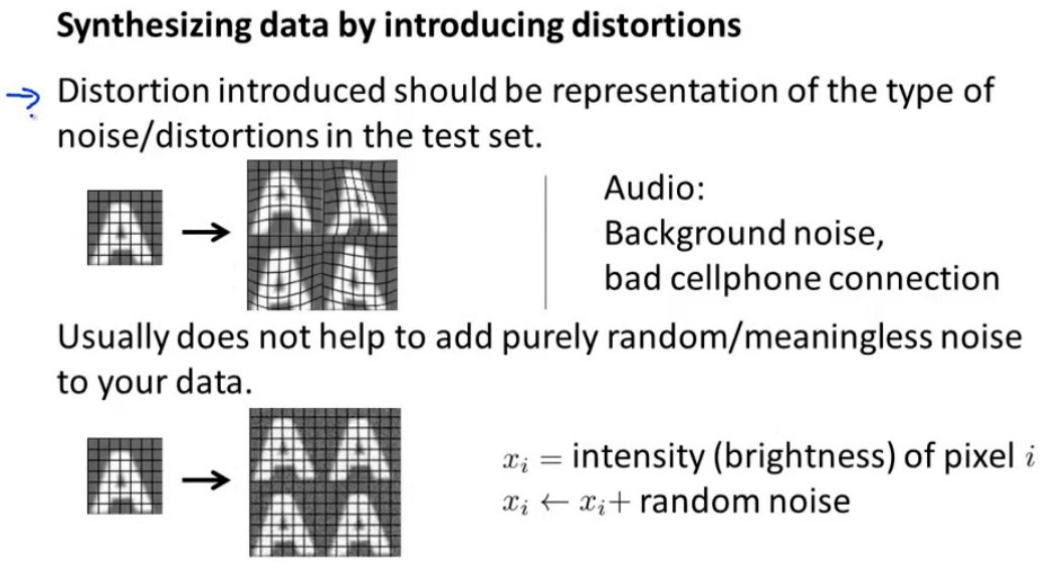


Fig. 14.12 How to choose appropriate distortions. Random noise is usually not a good choice.

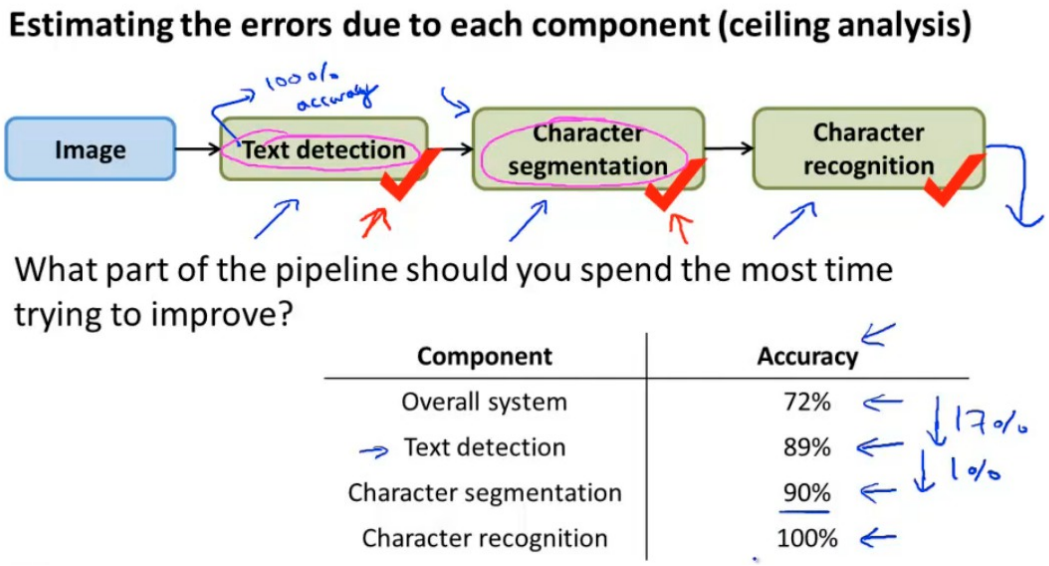


Fig. 14.13 Ceiling analysis: Figuring out where to get the most value for the effort expended on improving the processing pipeline.

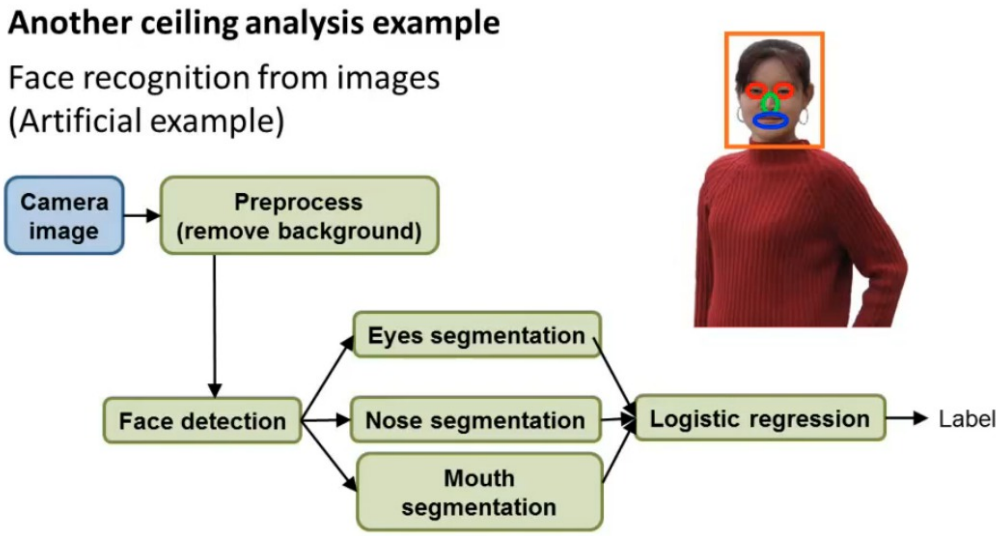


Fig. 14.14 A face recognition pipeline

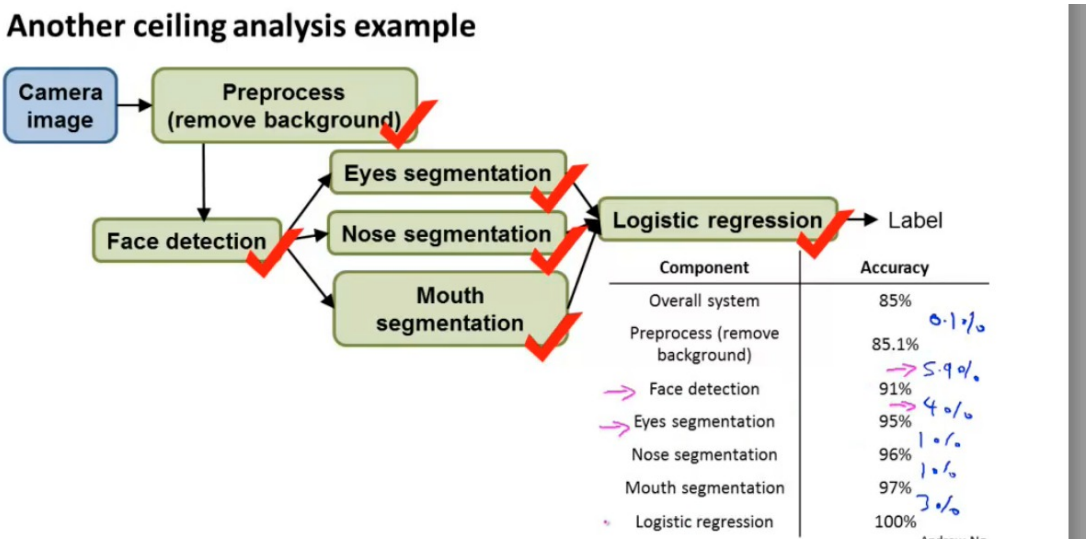


Fig. 14.15 A ceiling analysis on the face recognition pipeline.

Index