04: Linear Regression with Multiple Variables

Previous Next Index

Linear regression with multiple features

New version of linear regression with multiple features

- Multiple variables = multiple features
- In original version we had
  - X = house size, use this to predict
  - y = house price
- If in a new scheme we have more variables (such as number of bedrooms, number floors, age of the home)
  - x_1, x_2, x_3, x_4 are the four features
    - x_1 - size (feet squared)
    - x_2 - Number of bedrooms
    - x_3 - Number of floors
    - x_4 - Age of home (years)
  - y is the output variable (price)
- More notation
  - **n**
    - number of features (n = 4)
  - **m**
    - number of examples (i.e. number of rows in a table)
  - $\mathbf{x^i}$
    - vector of the input for an example (so a vector of the four parameters for the i^{th} input example)
    - i is an index into the training set
    - So
      - x is an n-dimensional feature vector
      - x^3 is, for example, the 3rd house, and contains the four features associated with that house
  - $\mathbf{x_j^i}$
    - The value of feature j in the ith training example
    - So
      - x_2^3 is, for example, the number of bedrooms in the third house
- Now we have multiple features
  - What is the form of our hypothesis?
  - Previously our hypothesis took the form;
    - $h_\theta(x) = \theta_0 + \theta_1 x$
      - Here we have two parameters (theta 1 and theta 2) determined by our cost function
      - One variable x
  - Now we have multiple features
    - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$
  - For example
    - $h_\theta(x) = 80 + 0.1 x_1 + 0.01 x_2 + 3 x_3 - 2 x_4$
      - An example of a hypothesis which is trying to predict the price of a house
      - Parameters are still determined through a cost function
  - For convenience of notation, $x_0 = 1$

        - For every example i you have an additional 0th feature for each example
        - So now your **feature vector** is n + 1 dimensional feature vector indexed from 0
            - This is a column vector called x
            - Each example has a column vector associated with it
            - So let's say we have a new example called "X"
        - **Parameters** are also in a 0 indexed n+1 dimensional vector
            - This is also a column vector called θ
            - This vector is the same for each example
    - Considering this, hypothesis can be written
        - $h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$
    - If we do
        - $h_\theta(x) = \theta^T X$
            - θ^T is an [1 x n+1] matrix
            - In other words, because θ is a column vector, the transposition operation transforms it into a row vector
            - So before
                - θ was a matrix [n + 1 x 1]
            - Now
                - θ^T is a matrix [1 x n+1]
            - Which means the inner dimensions of θ^T and X match, so they can be multiplied together as
                - [1 x n+1] * [n+1 x 1]
                - = $h_\theta(x)$
                - So, in other words, the transpose of our parameter vector * an input example X gives you a predicted hypothesis which is [1 x 1] dimensions (i.e. a single value)
        - This $x_0 = 1$ lets us write this like this
    - This is an example of multivariate linear regression

Gradient descent for multiple variables

- Fitting parameters for the hypothesis with gradient descent
    - Parameters are θ_0 to θ_n
    - Instead of thinking about this as n separate values, think about the parameters as a single vector (θ)
        - Where θ is n+1 dimensional
- Our cost function is

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

- Similarly, instead of thinking of J as a function of the n+1 numbers, J() is just a function of the parameter vector
    - J(θ)
- **Gradient descent**

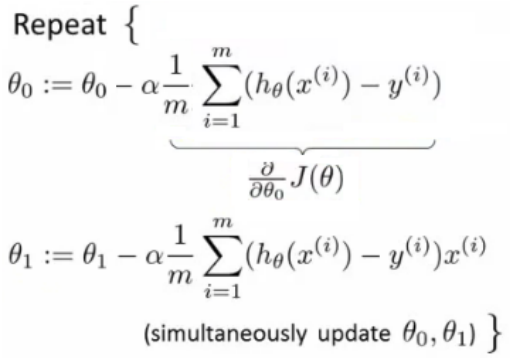
Repeat {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \ldots, \theta_n)$$

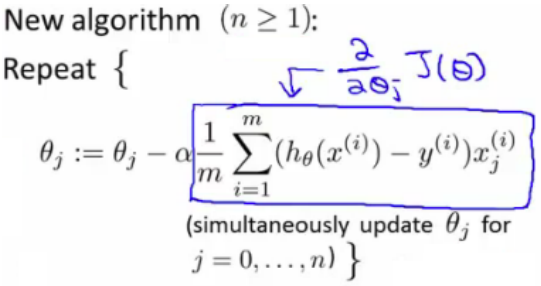
}

(simultaneously update for every $j = 0, \ldots, n$)

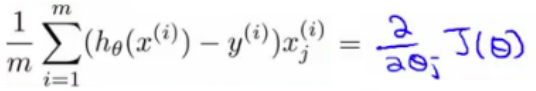
- Once again, this is
  - $\theta_j = \theta_j$ - learning rate (α) times the partial derivative of $J(\theta)$ with respect to $\theta_{J(...)}$
  - We do this through a **simultaneous update** of every θ_j value
- Implementing this algorithm
  - When n = 1



- Above, we have slightly different update rules for θ_0 and θ_1
  - Actually they're the same, except the end has a previously undefined $x_0^{(i)}$ as 1, so wasn't shown
- We now have an almost identical rule for multivariate gradient descent



- What's going on here?
  - We're doing this for each j (0 until n) as a simultaneous update (like when n = 1)
  - So, we re-set θ_j to
    - θ_j minus the learning rate (α) times the partial derivative of of the θ vector with respect to θ_j
    - In non-calculus words, this means that we do
      - Learning rate
      - Times 1/m (makes the maths easier)
      - Times the sum of
        - The hypothesis taking in the variable vector, minus the actual value, times the j-th value in that variable vector for EACH example
  - It's important to remember that

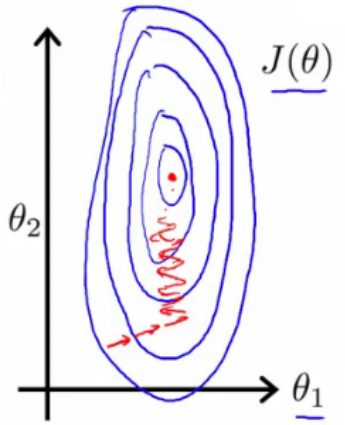


- These algorithm are highly similar

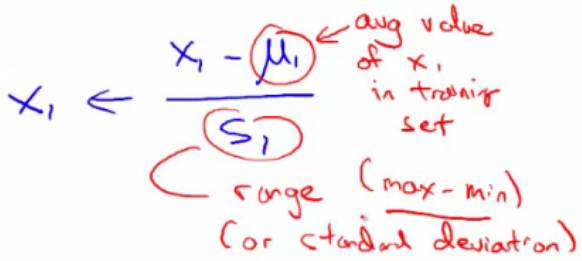
Gradient Decent in practice: 1 Feature Scaling

- Having covered the theory, we now move on to learn about some of the practical tricks
- Feature scaling

- If you have a problem with multiple features
- You should make sure those features have a similar scale
  - Means gradient descent will converge more quickly
- e.g.
  - x1 = size (0 - 2000 feet)
  - x2 = number of bedrooms (1-5)
  - Means the contours generated if we plot θ_1 vs. θ_2 give a very tall and thin shape due to the huge range difference
- Running gradient descent on this kind of cost function can take a long time to find the global minimum



- Pathological input to gradient descent
  - So we need to rescale this input so it's more effective
  - So, if you define each value from x1 and x2 by dividing by the max for each feature
  - Contours become more like circles (as scaled between 0 and 1)
- May want to get everything into -1 to +1 range (approximately)
  - Want to avoid large ranges, small ranges or very different ranges from one another
  - Rule a thumb regarding acceptable ranges
    - -3 to +3 is generally fine - any bigger bad
    - -1/3 to +1/3 is ok - any smaller bad
- Can do **mean normalization**
  - Take a feature x_i
    - Replace it by (x_i - mean)/max
    - So your values all have an average of about 0



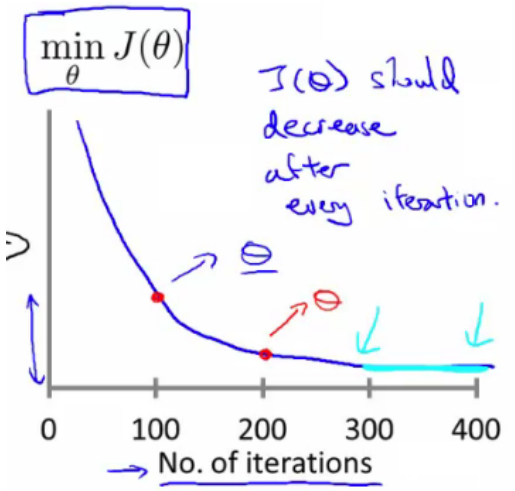
- Instead of max can also use standard deviation

Learning Rate α

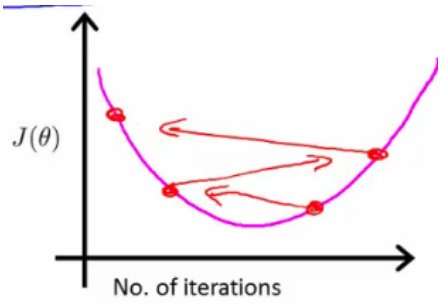
- Focus on the learning rate (α)
- Topics
  - Update rule
  - Debugging
  - How to chose α

Make sure gradient descent is working

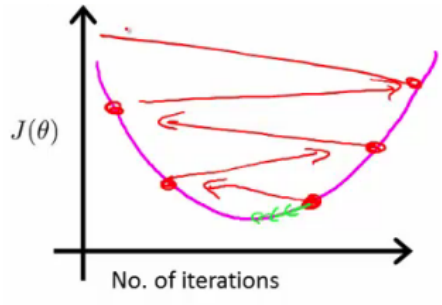
- Plot min J(θ) vs. no of iterations
  - (i.e. plotting J(θ) over the course of gradient descent
- If gradient descent is working then J(θ) should decrease after every iteration
- Can also show if you're not making huge gains after a certain number
  - Can apply heuristics to reduce number of iterations if need be
  - If, for example, after 1000 iterations you reduce the parameters by nearly nothing you could chose to only run 1000 iterations in the future
  - Make sure you don't accidentally hard-code thresholds like this in and then forget about why they're their though!



  - Number of iterations varies a lot
    - 30 iterations
    - 3000 iterations
    - 3000 000 iterations
    - Very hard to tel in advance how many iterations will be needed
    - Can often make a guess based a plot like this after the first 100 or so iterations
  - Automatic convergence tests
    - Check if J(θ) changes by a small threshold or less
      - Choosing this threshold is hard
      - So often easier to check for a straight line
        - Why? - Because we're seeing the straightness in the context of the whole algorithm
        - Could you design an automatic checker which calculates a threshold based on the systems preceding progress?
  - Checking its working
    - If you plot J(θ) vs iterations and see the value is increasing - means you probably need a smaller α
      - Cause is because your minimizing a function which looks like this



- But you overshoot, so reduce learning rate so you actually reach the minimum (green line)



- So, use a smaller α

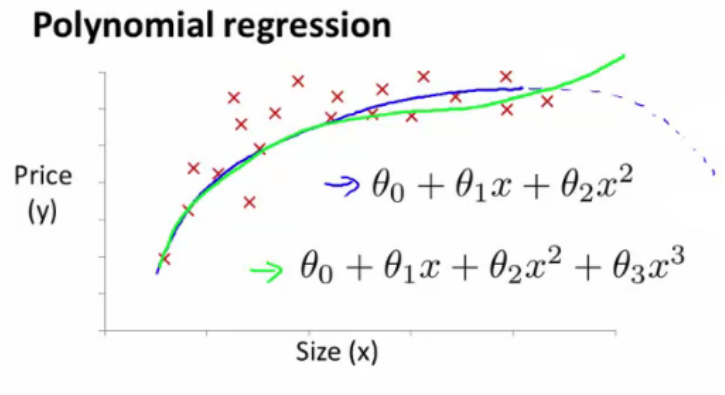
- Another problem might be if J(θ) looks like a series of waves
  - Here again, you need a smaller α
- However
  - If α is small enough, J(θ) will decrease on every iteration
  - BUT, if α is too small then rate is too slow
    - A less steep incline is indicative of a slow convergence, because we're decreasing by less on each iteration than a steeper slope
- Typically
  - Try a range of alpha values
  - Plot J(θ) vs number of iterations for each version of alpha
  - Go for roughly threefold increases
    - 0.001, 0.003, 0.01, 0.03. 0.1, 0.3

Features and polynomial regression

- Choice of features and how you can get different learning algorithms by choosing appropriate features
- Polynomial regression for non-linear function

- Example
  - House price prediction
    - Two features
      - Frontage - width of the plot of land along road (x_1)
      - Depth - depth away from road (x_2)
  - You don't have to use just two features
    - **Can create new features**
  - Might decide that an important feature is the land area
    - So, create a new feature = frontage * depth (x_3)
    - $h(x) = \theta_0 + \theta_1 x_3$
      - Area is a better indicator
  - Often, by defining new features you may get a better model

- Polynomial regression
  - May fit the data better
  - $\theta_0 + \theta_1 x + \theta_2 x^2$ e.g. here we have a quadratic function
  - For housing data could use a quadratic function
    - But may not fit the data so well - inflection point means housing prices decrease when size gets really big
    - So instead must use a cubic function



- How do we fit the model to this data
  - To map our old linear hypothesis and cost functions to these polynomial descriptions the easy thing to do is set
    - $x_1 = x$
    - $x_2 = x^2$
    - $x_3 = x^3$
  - By selecting the features like this and applying the linear regression algorithms you can do polynomial linear regression
  - Remember, feature scaling becomes even more important here
- Instead of a conventional polynomial you could do variable ^(1/something) - i.e. square root, cubed root etc
- Lots of features - later look at developing an algorithm to chose the best features

Normal equation

- For some linear regression problems the normal equation provides a better solution
- So far we've been using gradient descent
  - Iterative algorithm which takes steps to converse
- Normal equation solves θ analytically
  - Solve for the optimum value of theta
- Has some advantages and disadvantages

How does it work?

- Simplified cost function
  - $J(\theta) = a\theta^2 + b\theta + c$
    - θ is just a real number, not a vector
  - Cost function is a quadratic function
  - How do you minimize this?
    - Do

- $\frac{d}{d\theta} J(\theta) =$
  - Take derivative of J(θ) with respect to θ
  - Set that derivative equal to 0
  - Allows you to solve for the value of θ which minimizes J(θ)
- In our more complex problems;
  - Here θ is an n+1 dimensional vector of real numbers
  - Cost function is a function of the vector value
    - How do we minimize this function
      - Take the partial derivative of J(θ) with respect θ_j and set to 0 for every j
      - Do that and solve for θ_0 to θ_n
      - This would give the values of θ which minimize J(θ)
  - If you work through the calculus and the solution, the derivation is pretty complex
    - Not going to go through here
    - Instead, what do you need to know to implement this process

Example of normal equation

Size (feet²)	Number of bedrooms	Number of floors	Age of home (years)	Price ($1000)
x_1	x_2	x_3	x_4	y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178

- Here
  - m = 4
  - n = 4
- To implement the normal equation
  - Take examples
  - Add an extra column (x_0 feature)
  - Construct a matrix (X - **the design matrix**) which contains all the training data features in an [m x n+1] matrix
  - Do something similar for y
    - Construct a column vector y vector [m x 1] matrix
  - Using the following equation (X transpose * X) inverse times X transpose y

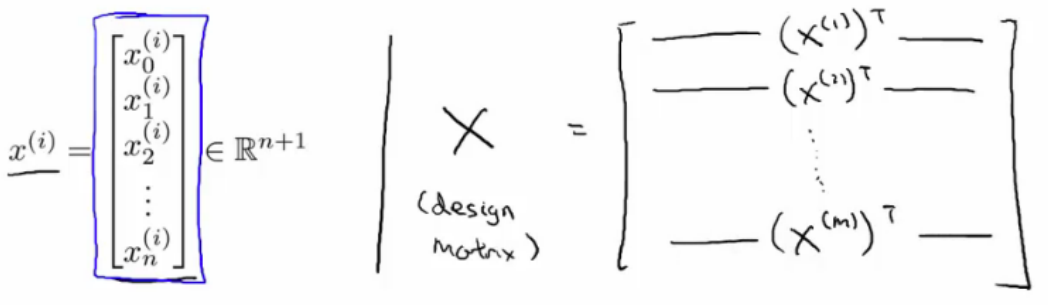
$$\theta = (X^T X)^{-1} X^T y$$

$$\left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} X \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \right)^{-1} X \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} X \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

- If you compute this, you get the value of theta which minimize the cost function

General case

- Have m training examples and n features
  - The **design matrix** (X)
    - Each training example is a n+1 dimensional feature column vector
    - X is constructed by taking each training example, determining its transpose (i.e. column -> row) and using it for a row in the design A
    - This creates an [m x (n+1)] matrix



  - **Vector y**
    - Used by taking all the y values into a column vector

$$\theta = (X^T X)^{-1} X^T y$$

- What is this equation?!
  - $(X^T * X)^{-1}$
    - What is this --> the inverse of the matrix $(X^T * X)$
      - i.e. $A = X^T X$
      - $A^{-1} = (X^T X)^{-1}$
- In octave and MATLAB you could do;

```
pinv(X'*x)*x'*y
```

    - X' is the notation for X transpose
    - pinv is a function for the inverse of a matrix
- In a previous lecture discussed feature scaling
  - If you're using the normal equation then no need for feature scaling

When should you use gradient descent and when should you use feature scaling?

  - ***Gradient descent***
    - Need to chose learning rate
    - Needs many iterations - could make it slower
    - Works well even when *n* is massive (millions)
      - Better suited to big data

        - What is a big *n* though
            - 100 or even a 1000 is still (relativity) small
            - If n is 10 000 then look at using gradient descent
    - ***Normal equation***
        - No need to chose a learning rate
        - No need to iterate, check for convergence etc.
        - Normal equation needs to compute $(X^T X)^{-1}$
            - This is the inverse of an n x n matrix
            - With most implementations computing a matrix inverse grows by $O(n^3)$
                - So not great
        - Slow of *n* is large
            - Can be much slower

Normal equation and non-invertibility

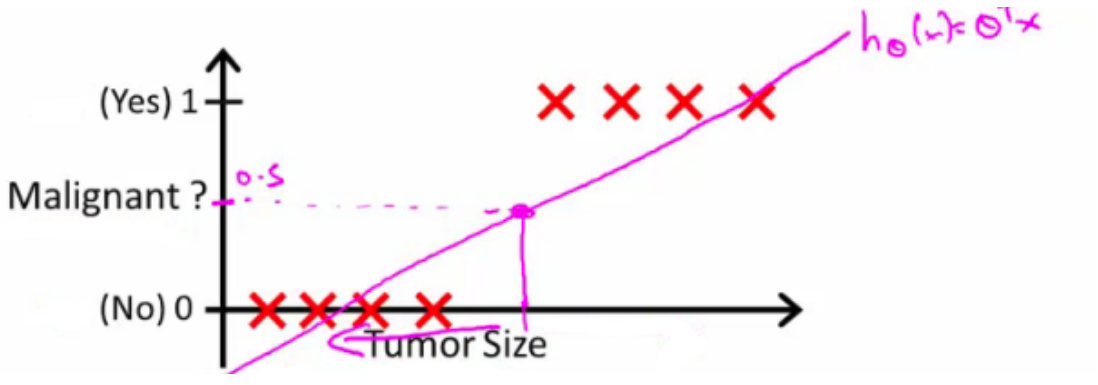
- Advanced concept
    - Often asked about, but quite advanced, perhaps optional material
    - Phenomenon worth understanding, but not probably necessary
- When computing $(X^T X)^{-1} * X^T * y)$
    - What if $(X^T X)$ is non-invertible (singular/degenerate)
        - Only some matrices are invertible
        - This should be quite a rare problem
            - Octave can invert matrices using
                - pinv (pseudo inverse)
                    - This gets the right value even if $(X^T X)$ is non-invertible
                - inv (inverse)
    - What does it mean for $(X^T X)$ to be non-invertible
        - Normally two common causes
            - **Redundant features** in learning model
                - e.g.
                    - x_1 = size in feet
                    - x_2 = size in meters squared
            - **Too many features**
                - e.g. m <= n (m is much larger than n)
                    - m = 10
                    - n = 100
                - Trying to fit 101 parameters from 10 training examples
                - Sometimes work, but not always a good idea
                - Not enough data
                - Later look at *why* this may be too little data
                - To solve this we
                    - Delete features
                    - Use **regularization** (let's you use lots of features for a small training set)
    - If you find $(X^T X)$ to be non-invertible
        - Look at features --> are features linearly dependent?
            - So just delete one, will solve problem

06: Logistic Regression

Previous Next Index

Classification

- Where y is a discrete value
  - Develop the logistic regression algorithm to determine what class a new input should fall into
- Classification problems
  - Email -> spam/not spam?
  - Online transactions -> fraudulent?
  - Tumor -> Malignant/benign
- Variable in these problems is Y
  - Y is either 0 or 1
    - 0 = negative class (absence of something)
    - 1 = positive class (presence of something)
- Start with **binary class problems**
  - Later look at multiclass classification problem, although this is just an extension of binary classification
- How do we develop a classification algorithm?
  - Tumour size vs malignancy (0 or 1)
  - We *could* use linear regression
    - Then threshold the classifier output (i.e. anything over some value is yes, else no)
    - In our example below linear regression with thresholding seems to work



- We can see above this does a reasonable job of stratifying the data points into one of two classes

    - But what if we had a single Yes with a very small tumour
    - This would lead to classifying all the existing yeses as nos
- Another issues with linear regression
    - We know Y is 0 or 1
    - Hypothesis can give values large than 1 or less than 0
- So, logistic regression generates a value where is always either 0 or 1

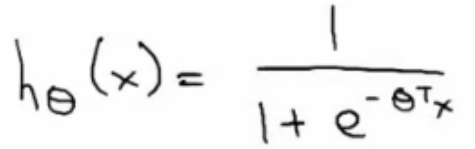
    - Logistic regression is a **classification algorithm** - don't be confused

Hypothesis representation

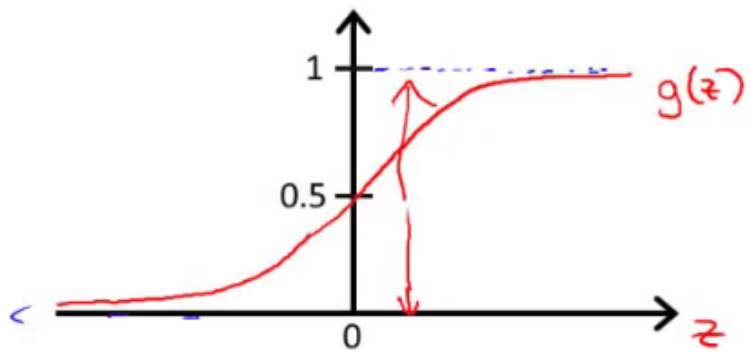
- What function is used to represent our hypothesis in classification
- We want our classifier to output values between 0 and 1

    - When using linear regression we did $h_\theta(x) = (\theta^T x)$
    - For classification hypothesis representation we do $h_\theta(x) = g((\theta^T x))$

        - Where we define g(z)
            - z is a real number
        - $g(z) = 1/(1 + e^{-z})$
            - This is the **sigmoid function**, or the **logistic function**
        - If we combine these equations we can write out the hypothesis as



- What does the sigmoid function look like
- Crosses 0.5 at the origin, then flattens out]
    - Asymptotes at 0 and 1



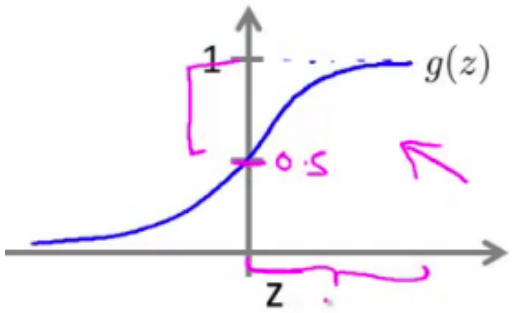
- Given this we need to fit θ to our data

Interpreting hypothesis output

- When our hypothesis ($h_\theta(x)$) outputs a number, we treat that value as the estimated probability that y=1 on input x
  - Example
    - If X is a feature vector with $x_0 = 1$ (as always) and x_1 = tumourSize
    - $h_\theta(x) = 0.7$
      - Tells a patient they have a 70% chance of a tumor being malignant
  - We can write this using the following notation
    - $h_\theta(x) = P(y=1|x ; \theta)$
  - What does this mean?
    - Probability that y=1, given x, parameterized by θ
- Since this is a binary classification task we know y = 0 or 1
  - So the following must be true
    - $P(y=1|x ; \theta) + P(y=0|x ; \theta) = 1$
    - $P(y=0|x ; \theta) = 1 - P(y=1|x ; \theta)$

Decision boundary

- Gives a better sense of what the hypothesis function is computing
- Better understand of what the hypothesis function looks like
  - One way of using the sigmoid function is;
    - When the probability of y being 1 is greater than 0.5 then we can predict y = 1
    - Else we predict y = 0
  - When is it exactly that $h_\theta(x)$ is greater than 0.5?
    - Look at sigmoid function
      - g(z) is greater than or equal to 0.5 when z is greater than or equal to 0

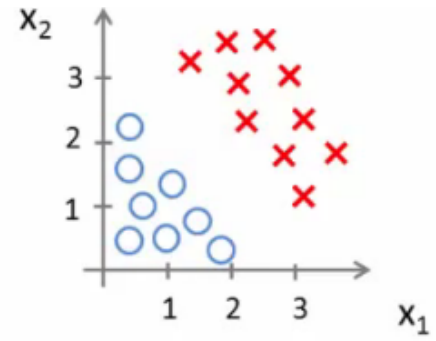


    - So if z is positive, g(z) is greater than 0.5
      - $z = (\theta^T x)$
    - So when

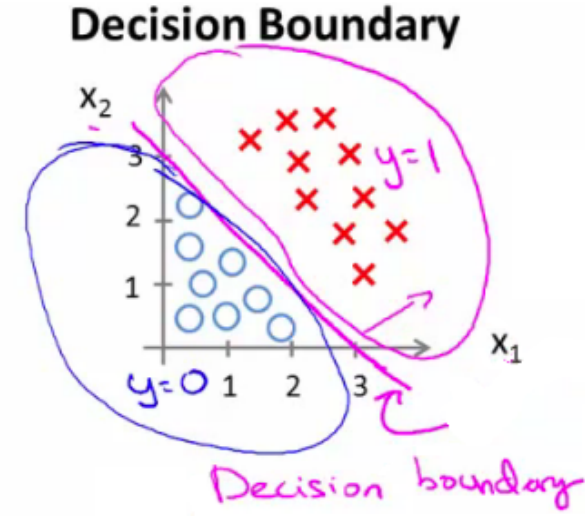
        - θ^T x >= 0
      - Then h_θ >= 0.5
- So what we've shown is that the hypothesis predicts y = 1 when θ^T x >= 0
  - The corollary of that when θ^T x <= 0 then the hypothesis predicts y = 0
  - Let's use this to better understand how the hypothesis makes its predictions

Decision boundary

- $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$



- So, for example
  - $\theta_0 = -3$
  - $\theta_1 = 1$
  - $\theta_2 = 1$
- So our parameter vector is a column vector with the above values
  - So, θ^T is a row vector = [-3,1,1]
- What does this mean?
  - The z here becomes θ^T x
  - We predict "y = 1" if
    - $-3x_0 + 1x_1 + 1x_2 >= 0$
    - $-3 + x_1 + x_2 >= 0$
- We can also re-write this as
  - If ($x_1 + x_2 >= 3$) then we predict y = 1
  - If we plot
    - $x_1 + x_2 = 3$ we graphically plot our **decision boundary**

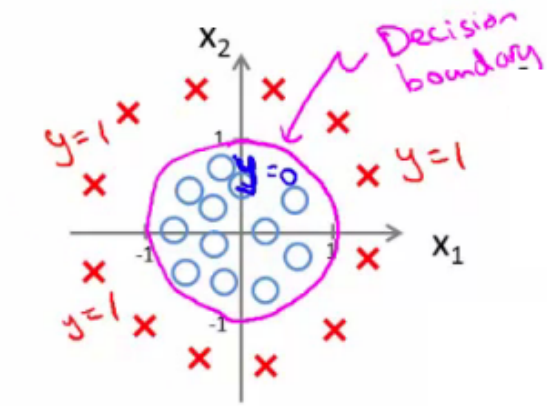


- Means we have these two regions on the graph
  - Blue = false
  - Magenta = true
  - Line = decision boundary
    - Concretely, the straight line is the set of points where $h_\theta(x) = 0.5$ exactly
  - The decision boundary is a property of the hypothesis
    - Means we can create the boundary with the hypothesis and parameters without any data
      - Later, we use the data to determine the parameter values
    - i.e. y = 1 if
      - $5 - x_1 > 0$
      - $5 > x_1$

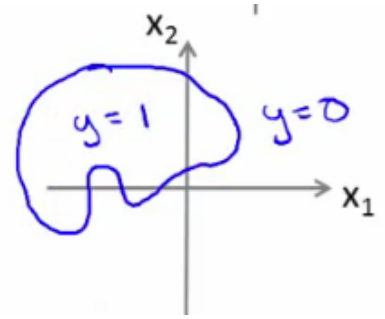
Non-linear decision boundaries

- Get logistic regression to fit a complex non-linear data set
  - Like polynomial regress add higher order terms
  - So say we have
    - $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_3 x_1^2 + \theta_4 x_2^2)$
    - We take the transpose of the θ vector times the input vector
      - Say θ^T was [-1,0,0,1,1] then we say;
      - Predict that "y = 1" *if*

        - $-1 + x_1^2 + x_2^2 >= 0$
          or
        - $x_1^2 + x_2^2 >= 1$
    - If we plot $x_1^2 + x_2^2 = 1$
        - This gives us a circle with a radius of 1 around 0



- Mean we can build more complex decision boundaries by fitting complex parameters to this (relatively) simple hypothesis
- More complex decision boundaries?
    - By using higher order polynomial terms, we can get even more complex decision boundaries



Cost function for logistic regression

- Fit θ parameters
- Define the optimization object for the cost function we use the fit the parameters
    - Training set of *m* training examples
        - Each example has is n+1 length column vector

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

m examples $x \in \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix}$ $x_0 = 1, y \in \{0, 1\}$

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

- This is the situation
  - Set of m training examples
  - Each example is a feature vector which is n+1 dimensional
  - $x_0 = 1$
  - $y \in \{0,1\}$
  - Hypothesis is based on parameters (θ)
    - Given the training set how to we chose/fit θ?
- Linear regression uses the following function to determine θ

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

- Instead of writing the squared error term, we can write
  - If we define "cost()" as;
    - $\text{cost}(h_\theta(x^i), y) = 1/2(h_\theta(x^i) - y^i)^2$
    - Which evaluates to the cost for an individual example using the same measure as used in linear regression
  - We can **redefine J(θ) as**

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

    - Which, appropriately, is the sum of all the individual costs over the training data (i.e. the same as linear regression)
- To further simplify it we can get rid of the superscripts
  - So

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(x), y)$$

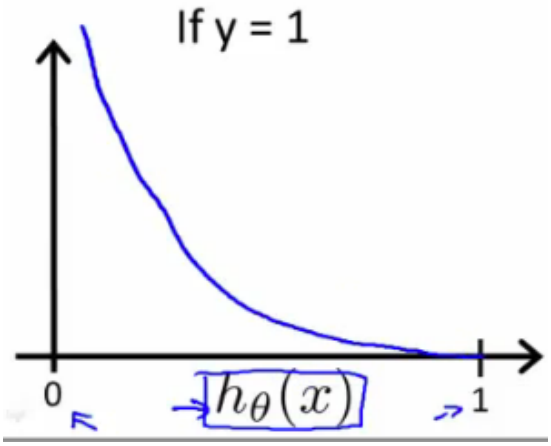
- What does this actually mean?
  - This is the cost you want the learning algorithm to pay if the outcome is $h_\theta(x)$ and the actual outcome is y
  - If we use this function for logistic regression this is a **non-convex function** for parameter optimization
    - Could work....
- What do we mean by non convex?
  - We have some function - J(θ) - for determining the parameters
  - Our hypothesis function has a non-linearity (sigmoid function of $h_\theta(x)$)
    - This is a complicated non-linear function
  - If you take $h_\theta(x)$ and plug it into the Cost() function, and them plug the Cost() function into J(θ) and plot J(θ) we find many local optimum -> *non convex function*
  - Why is this a problem
    - Lots of local minima mean gradient descent may not find the global optimum - may get stuck in a global minimum
  - We would like a convex function so if you run gradient descent you converge to a global minimum

A convex logistic regression cost function

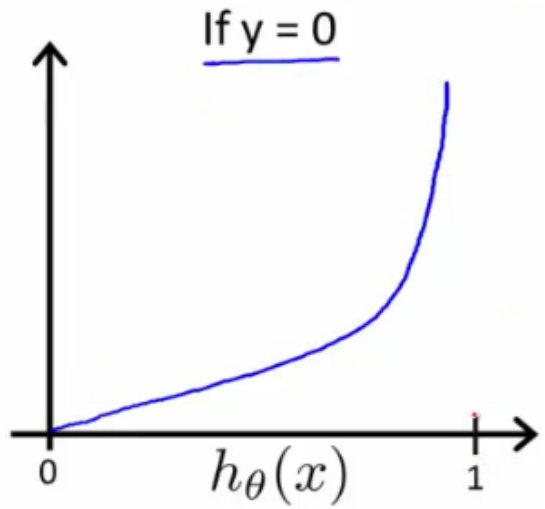
- To get around this we need a different, convex Cost() function which means we can apply gradient descent

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

- **This is our logistic regression cost function**
  - This is the penalty the algorithm pays
  - Plot the function
- Plot y = 1
  - So $h_\theta(x)$ evaluates as $-\log(h_\theta(x))$



- So when we're right, cost function is 0
  - Else it slowly increases cost function as we become "more" wrong
  - X axis is what we predict
  - Y axis is the cost associated with that prediction
- This cost functions has some interesting properties
  - If y = 1 and $h_\theta(x) = 1$
    - If hypothesis predicts exactly 1 and thats exactly correct then that corresponds to 0 (exactly, not nearly 0)
  - As $h_\theta(x)$ goes to 0
    - Cost goes to infinity
    - This captures the intuition that if $h_\theta(x) = 0$ (predict P (y=1|x; θ) = 0) but y = 1 this will penalize the learning algorithm with a massive cost
- What about if y = 0
- then cost is evaluated as -log(1- h_θ(x))
  - Just get inverse of the other function



- Now it goes to plus infinity as $h_\theta(x)$ goes to 1
- With our particular cost functions $J(\theta)$ is going to be convex and avoid local minimum

Simplified cost function and gradient descent

- Define a simpler way to write the cost function and apply gradient descent to the logistic regression
  - By the end should be able to implement a fully functional logistic regression function
- Logistic regression cost function is as follows

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

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

Note: $y = 0$ or 1 always

- This is the cost for a single example
  - For binary classification problems y is always 0 or 1
    - Because of this, we can have a simpler way to write the cost function
      - Rather than writing cost function on two lines/two cases
      - Can compress them into one equation - more efficient
  - Can write cost function is
    - **$\text{cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y)\log(1 - h_\theta(x))$**
      - This equation is a more compact of the two cases above
  - We know that there are only two possible cases
    - y = 1
      - Then our equation simplifies to
        - $-\log(h_\theta(x)) - (0)\log(1 - h_\theta(x))$
          - $-\log(h_\theta(x))$
          - Which is what we had before when y = 1
    - y = 0

        - Then our equation simplifies to
            - -(0)log($h_\theta(x)$) - (1)log(1 - $h_\theta(x)$)
            - = -log(1- $h_\theta(x)$)
            - Which is what we had before when y = 0
    - Clever!
- So, in summary, our cost function for the θ parameters can be defined as

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

- Why do we chose this function when other cost functions exist?
    - This cost function can be derived from statistics using the principle of **maximum likelihood estimation**
        - Note this does mean there's an underlying Gaussian assumption relating to the distribution of features
    - Also has the nice property that it's convex
- To fit parameters θ:
    - Find parameters θ which minimize J(θ)
    - This means we have a set of parameters to use in our model for future predictions
- Then, if we're given some new example with set of features x, we can take the θ which we generated, and output our prediction using

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

    - This result is
        - p(y=1 | x ; θ)
            - Probability y = 1, given x, parameterized by θ

How to minimize the logistic regression cost function

- Now we need to figure out how to minimize J(θ)
    - Use gradient descent as before
    - Repeatedly update each parameter using a learning rate

Repeat $\{$

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

(simultaneously update all θ_j)

$\}$

- If you had *n* features, you would have an n+1 column vector for θ
- This equation is the same as the linear regression rule
  - The only difference is that our definition for the hypothesis has changed
- Previously, we spoke about how to monitor gradient descent to check it's working
  - Can do the same thing here for logistic regression
- When implementing logistic regression with gradient descent, we have to update all the θ values (θ_0 to θ_n) simultaneously
  - Could use a for loop
  - Better would be a vectorized implementation
- Feature scaling for gradient descent for logistic regression also applies here

Advanced optimization

- Previously we looked at gradient descent for minimizing the cost function
- Here look at advanced concepts for minimizing the cost function for logistic regression
  - Good for large machine learning problems (e.g. huge feature set)
- *What is gradient descent actually doing?*
  - We have some cost function J(θ), and we want to minimize it
  - We need to write code which can take θ as input and compute the following
    - J(θ)
    - Partial derivative if J(θ) with respect to j (where j=0 to j = n)

$$J(\theta)$$

$$\frac{\partial}{\partial \theta_j} J(\theta) \quad \text{(for } j = 0, 1, \ldots, n \text{)}$$

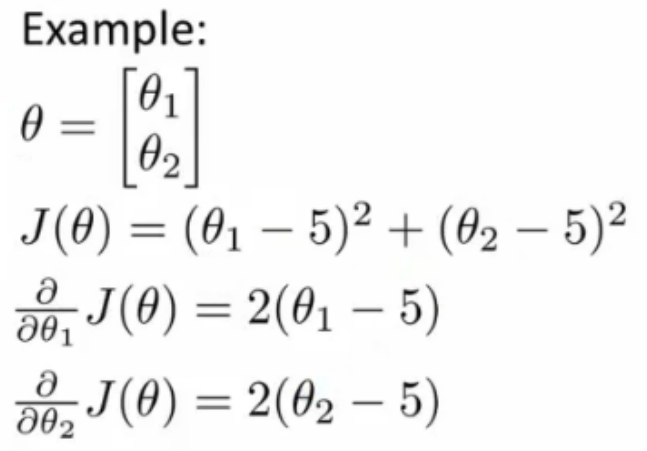
- Given code that can do these two things
  - Gradient descent repeatedly does the following update

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

- So update each j in θ sequentially
- So, we must;
  - Supply code to compute J(θ) and the derivatives
  - Then plug these values into gradient descent
- Alternatively, instead of gradient descent to minimize the cost function we could use
  - **Conjugate gradient**
  - **BFGS** (Broyden-Fletcher-Goldfarb-Shanno)
  - **L-BFGS** (Limited memory - BFGS)
- These are more optimized algorithms which take that same input and minimize the cost function
- These are *very* complicated algorithms
- Some properties
  - **Advantages**
    - No need to manually pick alpha (learning rate)
      - Have a clever inner loop (line search algorithm) which tries a bunch of alpha values and picks a good one
    - Often faster than gradient descent
      - Do more than just pick a good learning rate
    - Can be used successfully without understanding their complexity
  - **Disadvantages**
    - Could make debugging more difficult
    - Should not be implemented themselves
    - Different libraries may use different implementations - may hit performance

Using advanced cost minimization algorithms

- How to use algorithms
  - Say we have the following example



- Example above
  - θ_1 and θ_2 (two parameters)
  - Cost function here is $J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$
  - The derivatives of the $J(\theta)$ with respect to either θ_1 and θ_2 turns out to be the $2(\theta_i - 5)$
- First we need to define our cost function, which should have the following signature

```
function [jval, gradent] = costFunction(THETA)
```

- Input for the cost function is **THETA**, which is a vector of the θ parameters
- Two return values from **costFunction** are
  - **jval**
    - How we compute the cost function θ (the underived cost function)
      - In this case = $(\theta_1 - 5)^2 + (\theta_2 - 5)^2$
  - **gradient**
    - 2 by 1 vector
    - 2 elements are the two partial derivative terms
    - i.e. this is an n-dimensional vector
      - Each indexed value gives the partial derivatives for the partial derivative of $J(\theta)$ with respect to θ_i
      - Where i is the index position in the **gradient** vector
- With the cost function implemented, we can call the advanced algorithm using

```
options= optimset('GradObj', 'on', 'MaxIter', '100'); % define the
options data structure
initialTheta= zeros(2,1); # set the initial dimensions for theta %
initialize the theta values
```

```
[optTheta, funtionVal, exitFlag]= fminunc(@costFunction,
initialTheta, options); % run the algorithm
```

- Here
  - **options** is a data structure giving options for the algorithm
  - **fminunc**
    - function minimize the cost function (**f**ind **min**imum of **unc**onstrained multivariable function)
  - **@costFunction** is a pointer to the costFunction function to be used
- For the octave implementation
  - **initialTheta** must be a matrix of at least two dimensions
- How do we apply this to logistic regression?
  - Here we have a vector

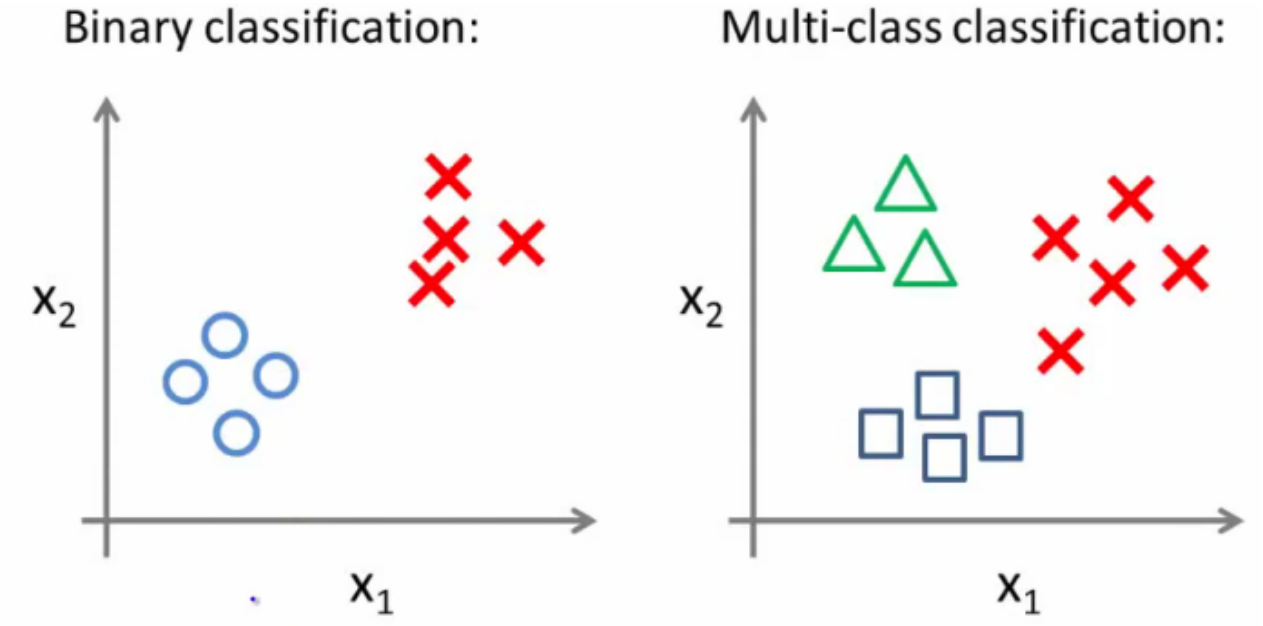
$$\text{theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

```
function [jVal, gradient] = costFunction(theta)
    jVal = [code to compute J(θ)];
    gradient(1) = [code to compute ∂/∂θ0 J(θ)];
    gradient(2) = [code to compute ∂/∂θ1 J(θ)];
    ⋮
    gradient(n+1) = [code to compute ∂/∂θn J(θ)];
```

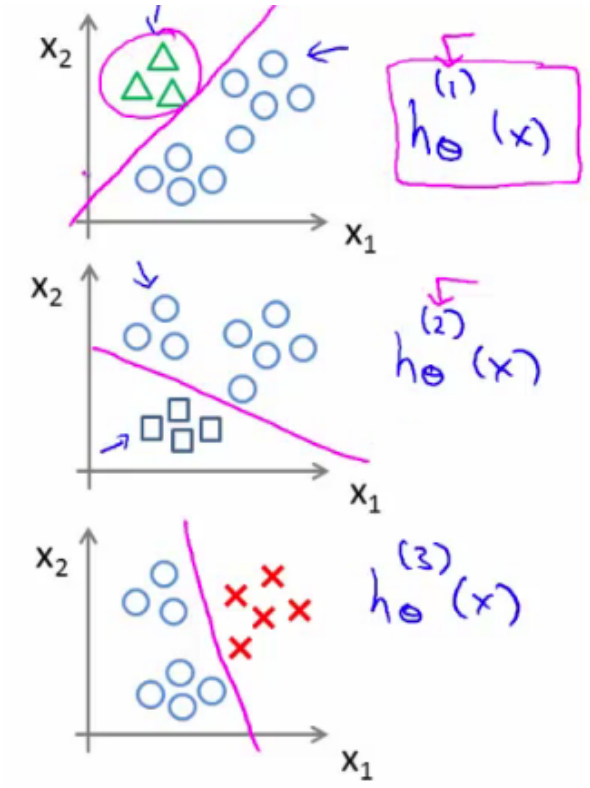
- Here
  - theta is a n+1 dimensional column vector
  - Octave indexes from 1, not 0
- Write a cost function which captures the cost function for logistic regression

Multiclass classification problems

- Getting logistic regression for multiclass classification using **one vs. all**
- Multiclass - more than yes or no (1 or 0)
    - Classification with multiple classes for assignment



- Given a dataset with three classes, how do we get a learning algorithm to work?
    - Use one vs. all classification make binary classification work for multiclass classification
- **One vs. all classification**
    - Split the training set into three separate binary classification problems
        - i.e. create a new fake training set
            - Triangle (1) vs crosses and squares (0) $h_\theta^1(x)$
                - $P(y=1 \mid x_1; \theta)$
            - Crosses (1) vs triangle and square (0) $h_\theta^2(x)$
                - $P(y=1 \mid x_2; \theta)$
            - Square (1) vs crosses and square (0) $h_\theta^3(x)$
                - $P(y=1 \mid x_3; \theta)$



- **Overall**
  - Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class i to predict the probability that y = i
  - On a new input, *x* to make a prediction, pick the class *i* that maximizes the probability that $h_\theta^{(i)}(x) = 1$

07: Regularization

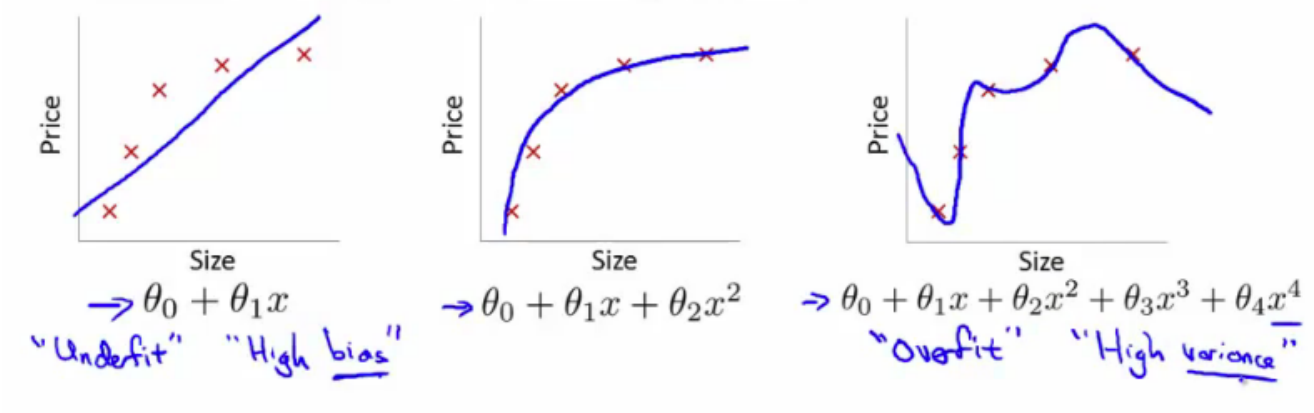
Previous Next Index

The problem of overfitting

- So far we've seen a few algorithms - work well for many applications, but can suffer from the problem of overfitting
- What is overfitting?
- What is regularization and how does it help

Overfitting with linear regression

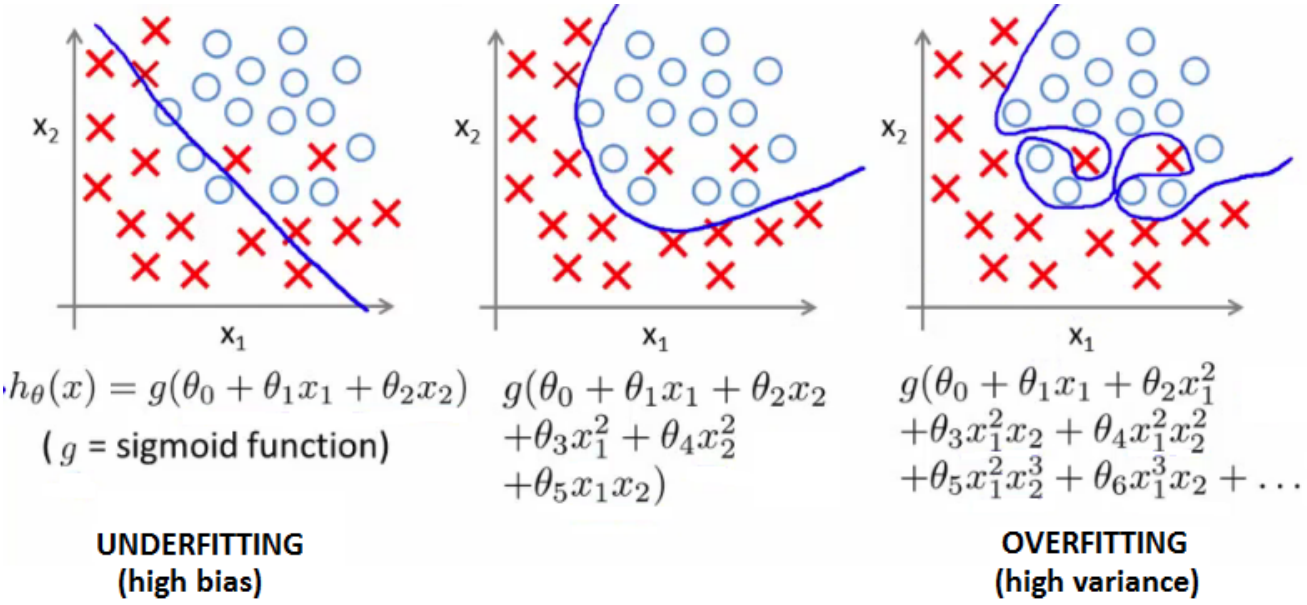
- Using our house pricing example again
  - Fit a linear function to the data - not a great model
    - This is **underfitting** - also known as **high bias**
    - Bias is a historic/technical one - if we're fitting a straight line to the data we have a strong preconception that there should be a linear fit
      - In this case, this is not correct, but a straight line can't help being straight!
  - Fit a quadratic function
    - Works well
  - Fit a 4th order polynomial
    - Now curve fit's through all five examples
      - Seems to do a good job fitting the training set
      - But, despite fitting the data we've provided very well, this is actually not such a good model
    - This is **overfitting** - also known as **high variance**
  - Algorithm has high variance
    - High variance - if fitting high order polynomial then the hypothesis can basically fit any data
    - Space of hypothesis is too large



- To recap, if we have too many features then the learned hypothesis may give a cost function of exactly zero
  - But this tries too hard to fit the training set
  - Fails to provide a *general* solution - **unable to generalize** (apply to new examples)

Overfitting with logistic regression

- Same thing can happen to logistic regression
  - Sigmoidal function is an underfit
  - But a high order polynomial gives and overfitting (high variance hypothesis)



Addressing overfitting

- Later we'll look at identifying when overfitting and underfitting is occurring

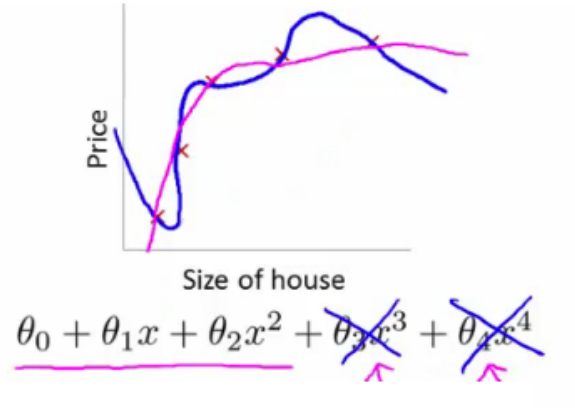
- Earlier we just plotted a higher order function - saw that it looks "too curvy"
  - Plotting hypothesis is one way to decide, but doesn't always work
  - Often have lots of a features - here it's not just a case of selecting a degree polynomial, but also harder to plot the data and visualize to decide what features to keep and which to drop
  - If you have lots of features and little data - overfitting can be a problem
- How do we deal with this?
  - 1) **Reduce number of features**
    - Manually select which features to keep
    - Model selection algorithms are discussed later (good for reducing number of features)
    - But, in reducing the number of features we lose some information
      - Ideally select those features which minimize data loss, but even so, some info is lost
  - 2) **Regularization**
    - Keep all features, but reduce magnitude of parameters θ
    - Works well when we have a lot of features, each of which contributes a bit to predicting y

Cost function optimization for regularization

- Penalize and make some of the θ parameters really small
  - e.g. here θ_3 and θ_4

$$\min_\theta \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\,\theta_3^2 + 1000\,\theta_4^2$$

- The addition in blue is a modification of our cost function to help penalize θ_3 and θ_4
  - So here we end up with θ_3 and θ_4 being close to zero (because the constants are massive)
  - So we're basically left with a quadratic function



- In this example, we penalized two of the parameter values
  - More generally, regularization is as follows
- Regularization
  - Small values for parameters corresponds to a simpler hypothesis (you effectively get rid of some of the terms)
  - A simpler hypothesis is less prone to overfitting
- Another example
  - Have 100 features x_1, x_2, ..., x_{100}
  - Unlike the polynomial example, we don't know what are the high order terms
    - How do we pick the ones to pick to shrink?
  - With regularization, take cost function and modify it to shrink all the parameters
    - Add a term at the end
      - This regularization term shrinks every parameter
      - By convention you don't penalize θ_0 - minimization is from θ_1 onwards

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

$\theta_1, \theta_2, \theta_3, \ldots, \theta_{100}$

- In practice, if you include θ_0 has little impact
- λ is the **regularization parameter**
  - Controls a trade off between our two goals
    - 1) Want to fit the training set well
    - 2) Want to keep parameters small
- With our example, using the **regularized objective** (i.e. the cost function with

the regularization term) you get a much smoother curve which fits the data and gives a much better hypothesis

- If λ is very large we end up penalizing ALL the parameters (θ_1, θ_2 etc.) so all the parameters end up being close to zero
  - If this happens, it's like we got rid of all the terms in the hypothesis
    - This results here is then underfitting
  - So this hypothesis is too biased because of the absence of any parameters (effectively)

- So, λ should be chosen carefully - not too big...
  - We look at some automatic ways to select λ later in the course

Regularized linear regression

- Previously, we looked at two algorithms for linear regression
  - Gradient descent
  - Normal equation
- Our linear regression with regularization is shown below

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

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

- Previously, gradient descent would repeatedly update the parameters θ_j, where j = 0,1,2...n simultaneously
  - Shown below

Repeat $\{$

$$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

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

$$(j = 1, 2, 3, \ldots, n)\ \}$$

- We've got the θ_0 update here shown explicitly
  - This is because for regularization we don't penalize θ_0 so treat it slightly differently

- How do we regularize these two rules?
  - Take the term and add $\lambda/m * \theta_j$
    - Sum for every θ (i.e. j = 0 to n)
  - This gives regularization for gradient descent
- We can show using calculus that the equation given below is the partial derivative of the regularized $J(\theta)$

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

$$(j = 0, 1, 2, 3, \ldots, n)$$

$$\frac{\partial}{\partial \theta_j} J(\theta) \quad \text{regularized}$$

- The update for θ_j
  - θ_j gets updated to
    - θ_j - α * [a big term which also depends on θ_j]
- So if you group the θ_j terms together

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

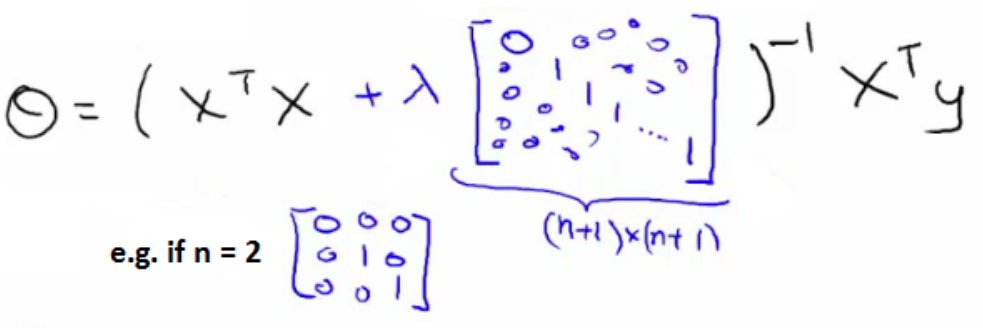
- The term

$$(1 - \alpha \frac{\lambda}{m})$$

  - Is going to be a number less than 1 usually
  - Usually learning rate is small and m is large
    - So this typically evaluates to (1 - a small number)
    - So the term is often around 0.99 to 0.95
- This in effect means θ_j gets multiplied by 0.99
  - Means the squared norm of θ_j a little smaller
  - The second term is exactly the same as the original gradient descent

Regularization with the normal equation

- Normal equation is the other linear regression model
  - Minimize the $J(\theta)$ using the normal equation
  - To use regularization we add a term (+ λ [n+1 x n+1]) to the equation
    - [n+1 x n+1] is the n+1 identity matrix



Regularization for logistic regression

- We saw earlier that logistic regression can be prone to overfitting with lots of features
- Logistic regression cost function is as follows;

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

- To modify it we have to add an extra term

$$+ \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

- This has the effect of penalizing the parameters θ_1, θ_2 up to θ_n
  - Means, like with linear regression, we can get what appears to be a better fitting lower order hypothesis
- How do we implement this?
  - Original logistic regression with gradient descent function was as follows

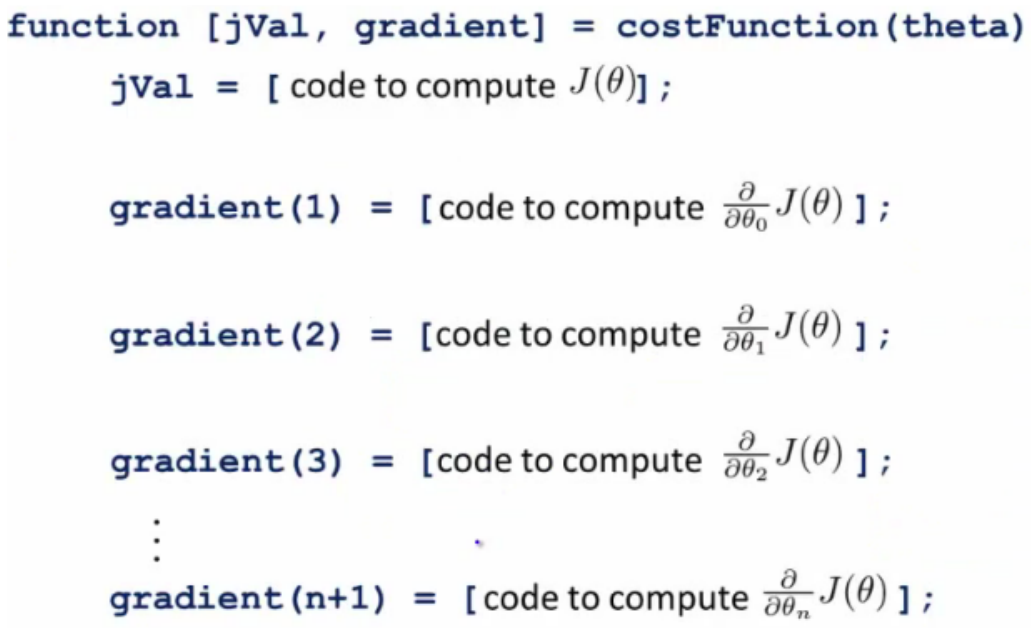
$$\theta_j := \theta_j - \alpha \quad \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$
$$(j = 0, 1, 2, 3, \ldots, n)$$

- Again, to modify the algorithm we simply need to modify the update rule for θ_1, onwards
  - Looks cosmetically the same as linear regression, except obviously the hypothesis is very different

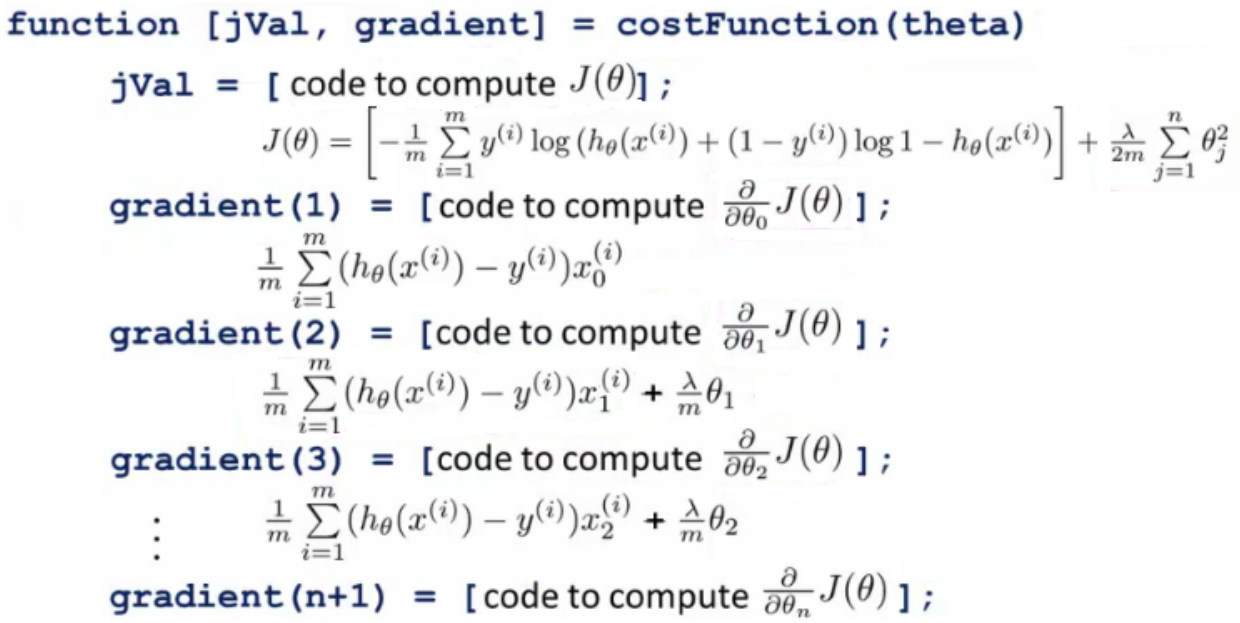
$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Advanced optimization of regularized linear regression

- As before, define a costFunction which takes a θ parameter and gives jVal and gradient back



- use `fminunc`
  - Pass it an `@costfunction` argument
  - Minimizes in an optimized manner using the cost function
- `jVal`
  - Need code to compute J(θ)
    - Need to include regularization term
- Gradient
  - Needs to be the partial derivative of J(θ) with respect to θ_i
  - Adding the appropriate term here is also necessary



- Ensure summation doesn't extend to to the lambda term!
  - It doesn't, but, you know, don't be daft!

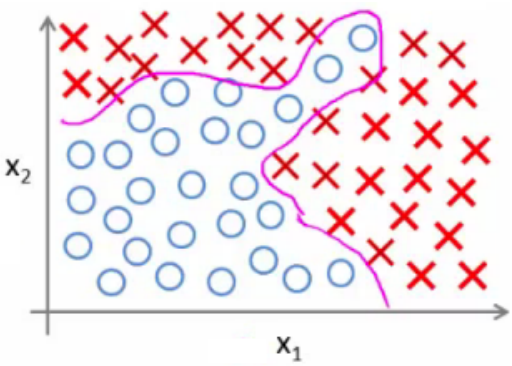
08: Neural Networks - Representation

Previous Next Index

Neural networks - Overview and summary

Why do we need neural networks?

- Say we have a complex supervised learning classification problem
  - Can use logistic regression with many polynomial terms
  - Works well when you have 1-2 features
  - If you have 100 features



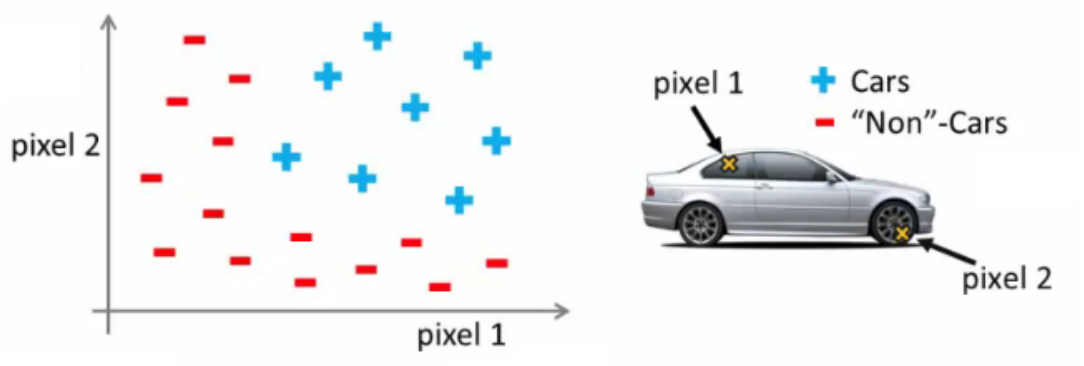
- e.g. our housing example
  - 100 house features, predict odds of a house being sold in the next 6 months
  - Here, if you included all the quadratic terms (second order)
    - There are lots of them (x_1^2 ,x_1x_2, x_1x_4 ..., x_1x_{100})
    - For the case of n = 100, you have about 5000 features
    - Number of features grows $O(n^2)$
    - This would be computationally expensive to work with as a feature set
- A way around this to only include a subset of features
  - However, if you don't have enough features, often a model won't let you fit a complex dataset
- If you include the cubic terms
  - e.g. ($x_1^2x_2$, $x_1x_2x_3$, $x_1x_4x_{23}$ etc)
  - There are even more features grows $O(n^3)$
  - About 170 000 features for n = 100
- Not a good way to build classifiers when n is large

Example: Problems where n is large - computer vision

- Computer vision sees a matrix of pixel intensity values

    - Look at matrix - explain what those numbers represent
- To build a car detector
    - Build a training set of
        - Not cars
        - Cars
    - Then test against a car
- How can we do this

    - Plot two pixels (two pixel locations)
    - Plot car or not car on the graph



- Need a non-linear hypothesis to separate the classes
- Feature space
    - If we used 50 x 50 pixels --> 2500 pixels, so n = 2500
    - If RGB then 7500
    - If 100 x 100 RB then --> 50 000 000 features
- Too big - wayyy too big

    - So - simple logistic regression here is not appropriate for large complex systems
    - Neural networks are much better for a complex nonlinear hypothesis even when feature space is huge

Neurons and the brain

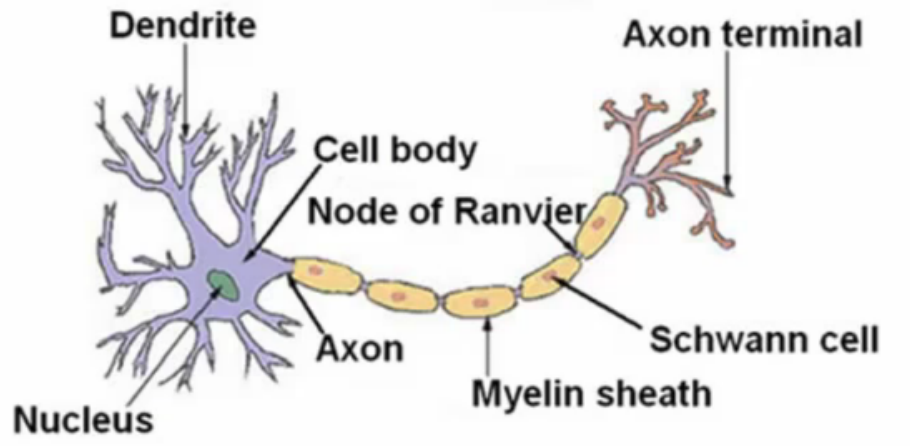
- **Neural networks** (**NNs**) were originally motivated by looking at machines which replicate the brain's functionality
    - Looked at here as a machine learning technique
- Origins

    - To build learning systems, why not mimic the brain?
    - Used a lot in the 80s and 90s
    - Popularity diminished in late 90s
    - Recent major resurgence

        - NNs are computationally expensive, so only recently large scale neural networks became computationally feasible
- Brain
    - Does loads of crazy things
        - Hypothesis is that the brain has a single learning algorithm
    - Evidence for hypothesis
        - Auditory cortex --> takes sound signals
            - If you cut the wiring from the ear to the auditory cortex
            - Re-route optic nerve to the auditory cortex
            - Auditory cortex learns to see
        - Somatosensory context (touch processing)
            - If you rewrite optic nerve to somatosensory cortex then it learns to see
    - With different tissue learning to see, maybe they all learn in the same way
        - Brain learns by itself how to learn
- Other examples
    - Seeing with your tongue
        - Brainport
            - Grayscale camera on head
            - Run wire to array of electrodes on tongue
            - Pulses onto tongue represent image signal
            - Lets people see with their tongue
    - Human echolocation
        - Blind people being trained in schools to interpret sound and echo
        - Lets them move around
    - Haptic belt direction sense
        - Belt which buzzes towards north
        - Gives you a sense of direction
- Brain can process and learn from data from any source

Model representation 1

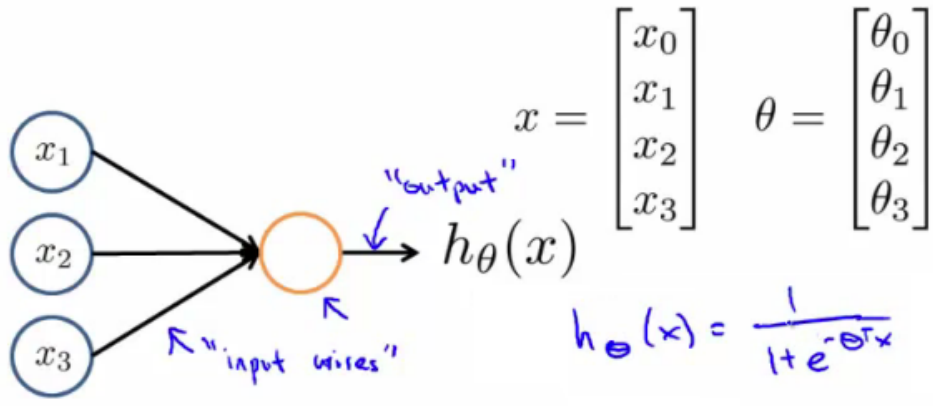
- How do we represent neural networks (NNs)?
    - Neural networks were developed as a way to simulate networks of neurones
- What does a neurone look like



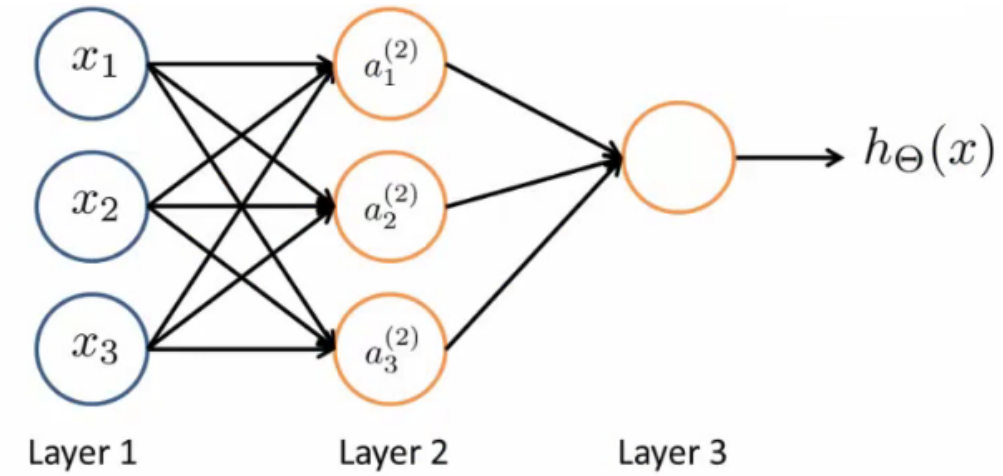
- Three things to notice
  - Cell body
  - Number of input wires (dendrites)
  - Output wire (axon)
- Simple level
  - Neurone gets one or more inputs through dendrites
  - Does processing
  - Sends output down axon
- Neurons communicate through electric spikes
  - Pulse of electricity via axon to another neurone

Artificial neural network - representation of a neurone

- In an artificial neural network, a neurone is a logistic unit
  - Feed input via input wires
  - Logistic unit does computation
  - Sends output down output wires
- That logistic computation is just like our previous logistic regression hypothesis calculation



- Very simple model of a neuron's computation
  - Often good to include an x_0 input - the **bias unit**
    - This is equal to 1
- This is an artificial neurone with a sigmoid (logistic) activation function
  - Θ vector may also be called the **weights** of a model
- The above diagram is a single neurone
  - Below we have a group of neurones strung together

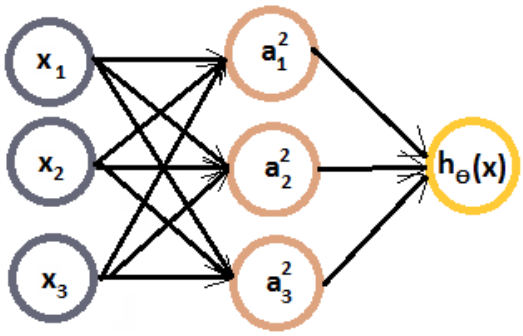


- Here, input is x_1, x_2 and x_3
  - We could also call input activation on the first layer - i.e. (a_1^1, a_2^1 and a_3^1)
  - Three neurones in layer 2 (a_1^2, a_2^2 and a_3^2)
  - Final fourth neurone which produces the output
    - Which again we *could* call a_1^3
- First layer is the **input layer**
- Final layer is the **output layer** - produces value computed by a hypothesis
- Middle layer(s) are called the **hidden layers**
  - You don't observe the values processed in the hidden layer
  - Not a great name
  - Can have many hidden layers

Neural networks - notation

- **$a_i^{(j)}$ - activation of unit *i* in layer *j***
  - So, a_1^2 - is the **activation** of the 1st unit in the second layer
  - By activation, we mean the value which is computed and output by that node
- **$\Theta^{(j)}$ - matrix of parameters controlling the function mapping from layer *j* to layer *j* + 1**
  - Parameters for controlling **mapping** from one layer to the next
  - If network has
    - s_j units in layer *j* and

        - s_{j+1} units in layer j + 1
        - Then Θ^j will be of dimensions $[s_{j+1} \times s_j + 1]$
            - Because
                - s_{j+1} is equal to the number of units in layer (j + 1)
                - is equal to the number of units in layer j, plus an additional unit
    - Looking at the Θ matrix
        - Column length is the number of units in the following layer
        - Row length is the number of units in the current layer + 1 (because we have to map the bias unit)
        - So, if we had two layers - 101 and 21 units in each
            - Then Θ^j would be = [21 x 102]
- What are the computations which occur?
    - We have to calculate the activation for each node
    - That activation depends on
        - The input(s) to the node
        - The parameter associated with that node (from the Θ vector associated with that layer)
- Below we have an example of a network, with the associated calculations for the four nodes below

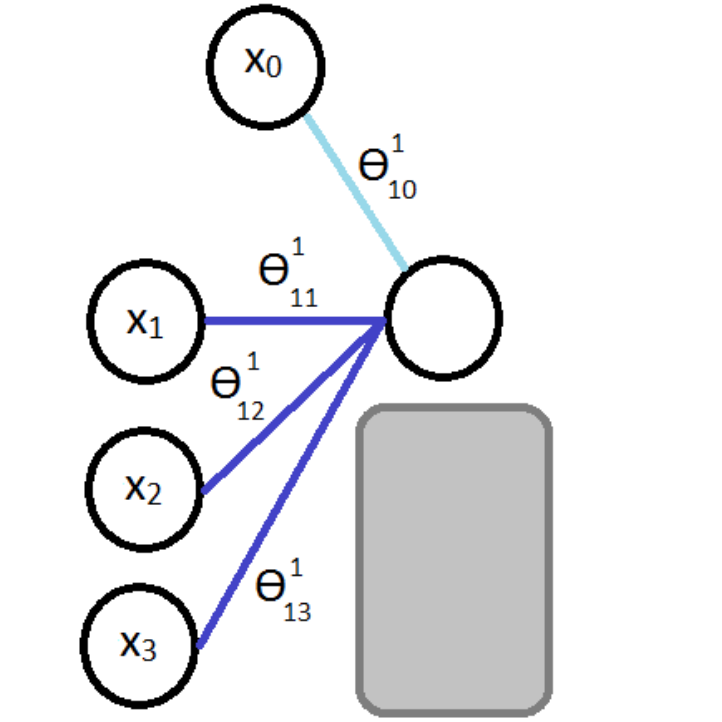


$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

- As you can see
    - We calculate each of the layer-2 activations based on the input values with the bias term (which is equal to 1)
        - i.e. x_0 to x_3
    - We then calculate the final hypothesis (i.e. the single node in layer 3) using exactly the same logic, except in input is not x values, but the activation

values from the preceding layer

- The activation value on each hidden unit (e.g. a_1^2) is equal to the sigmoid function applied to the linear combination of inputs
  - Three input units
    - So $\Theta^{(1)}$ is the matrix of parameters governing the mapping of the input units to hidden units
      - $\Theta^{(1)}$ here is a [3 x 4] dimensional matrix
  - Three hidden units
    - Then $\Theta^{(2)}$ is the matrix of parameters governing the mapping of the hidden layer to the output layer
      - $\Theta^{(2)}$ here is a [1 x 4] dimensional matrix (i.e. a row vector)
  - One output unit
- Something conceptually important (that I hadn't really grasped the first time) is that
  - **Every input/activation goes to every node in following layer**
    - Which means each "layer transition" uses a matrix of parameters with the following significance
      - For the sake of consistency with later nomenclature, we're using j,i and l as our variables here (although later in this section we use j to show the layer we're on)
      - $\mathbf{\Theta}_{ji}^{l}$
        - j (first of two subscript numbers)= ranges from 1 to the number of units in layer l+1
        - i (second of two subscript numbers) = ranges from 0 to the number of units in layer l
        - l is the layer you're moving FROM
      - This is perhaps more clearly shown in my slightly over the top example below



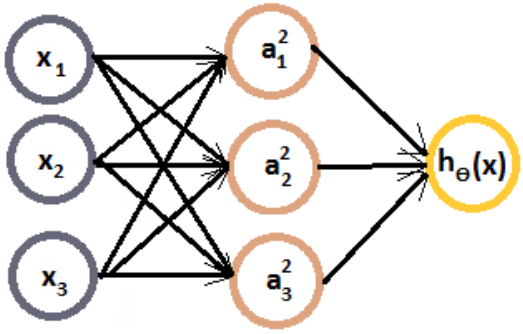
$$g(\Theta_{10}^{1}x_0 + \Theta_{11}^{1}x_1 + \Theta_{12}^{1}x_2 + \Theta_{13}^{1}x_3)$$

- For example
  - Θ_{13}^{1} = means
    - **1** - we're mapping to node 1 in layer l+1
    - **3** - we're mapping from node 3 in layer l
    - **1** - we're mapping from layer 1

Model representation II

Here we'll look at how to carry out the computation efficiently through a vectorized implementation. We'll also consider
why NNs are good and how we can use them to learn complex non-linear things

- Below is our original problem from before
  - Sequence of steps to compute output of hypothesis are the equations below



$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

- Define some additional terms
  - $z_1^2 = \Theta_{10}^1 x_0 + \Theta_{11}^1 x_1 + \Theta_{12}^1 x_2 + \Theta_{13}^1 x3$
  - Which means that
    - $a_1^2 = g(z_1^2)$
  - NB, superscript numbers are the layer associated
- Similarly, we define the others as
  - z_2^2 and z_3^2
  - These values are just a linear combination of the values
- If we look at the block we just redefined
  - We can vectorize the neural network computation
  - So lets define
    - x as the feature vector x
    - z^2 as the vector of z values from the second layer

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

- z^2 is a 3x1 vector
- We can vectorize the computation of the neural network as as follows in two steps
  - $z^2 = \Theta^{(1)}x$

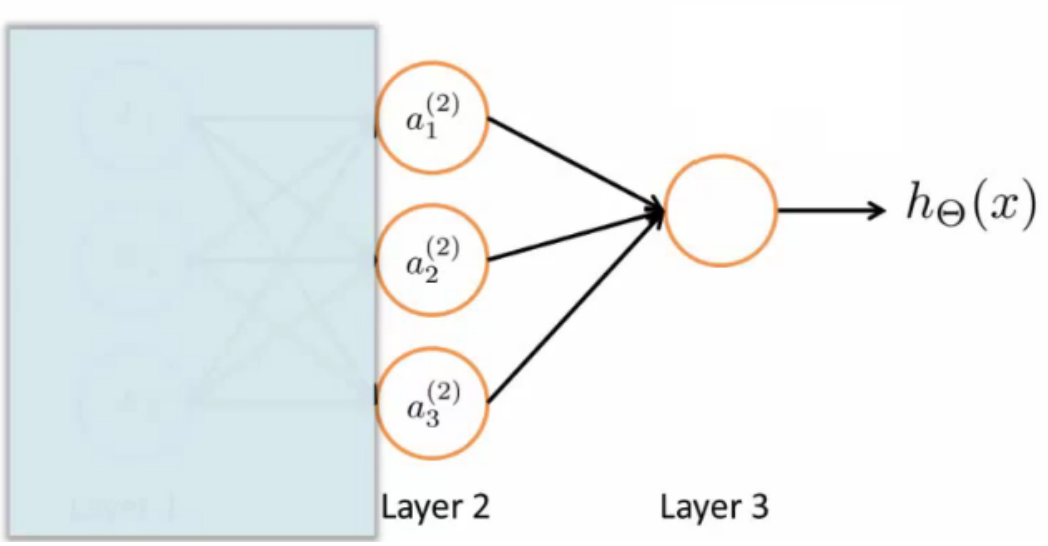
        - i.e. $\Theta^{(1)}$ is the matrix defined above
        - x is the feature vector
    - $a^2 = g(z^{(2)})$
        - To be clear, z^2 is a 3x1 vecor
        - a^2 is also a 3x1 vector
        - g() applies the sigmoid (logistic) function element wise to each member of the z^2 vector
- To make the notation with input layer make sense;
    - $a^1 = x$
        - a^1 is the activations in the input layer
        - Obviously the "activation" for the input layer is just the input!
    - So we define x as a^1 for clarity
        - So
            - a^1 is the vector of inputs
            - a^2 is the vector of values calculated by the $g(z^2)$ function
- Having calculated then z^2 vector, we need to calculate a_0^2 for the final hypothesis calculation

$$h_\Theta(x) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

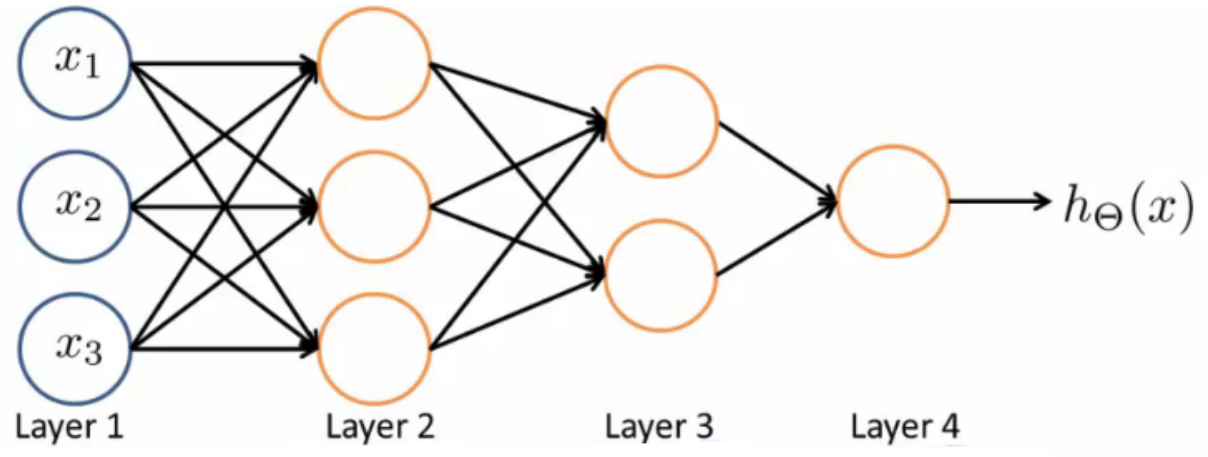
- To take care of the extra bias unit add $a_0^2 = 1$
    - So add a_0^2 to a^2 making it a 4x1 vector
- So,
    - $z^3 = \Theta^2 a^2$
        - This is the inner term of the above equation
    - $h_\Theta(x) = a^3 = g(z^3)$
- This process is also called **forward propagation**
    - Start off with activations of input unit
        - i.e. the x vector as input
    - Forward propagate and calculate the activation of each layer sequentially
    - This is a vectorized version of this implementation

Neural networks learning its own features

- Diagram below looks a lot like logistic regression



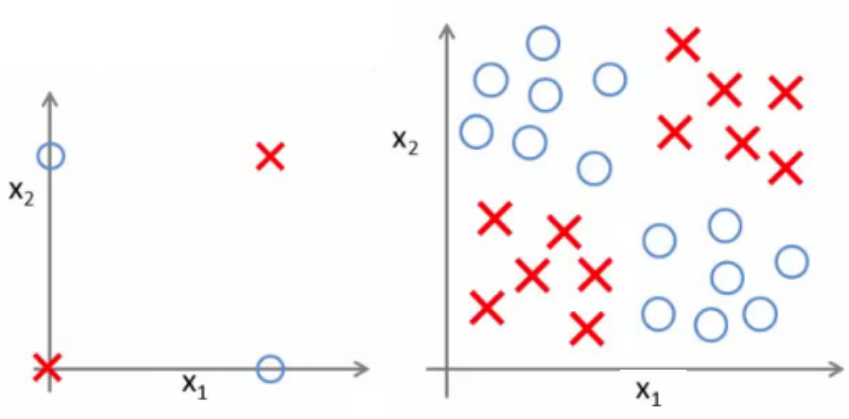
- Layer 3 is a logistic regression node
  - The hypothesis output = $g(\Theta_{10}^2 a_0^2 + \Theta_{11}^2 a_1^2 + \Theta_{12}^2 a_2^2 + \Theta_{13}^2 a_3^2)$
  - This is just logistic regression
    - The only difference is, instead of input a feature vector, the features are just values calculated by the hidden layer
- The features a_1^2, a_2^2, and a_3^2 are calculated/learned - not original features
- So the mapping from layer 1 to layer 2 (i.e. the calculations which generate the a^2 features) is determined by another set of parameters - Θ^1
  - So instead of being constrained by the original input features, a neural network can learn its own features to feed into logistic regression
  - Depending on the Θ^1 parameters you can learn some interesting things
    - Flexibility to learn whatever features it wants to feed into the final logistic regression calculation
      - So, if we compare this to previous logistic regression, you would have to calculate your own exciting features to define the best way to classify or describe something
      - Here, we're letting the hidden layers do that, so we feed the hidden layers our input values, and let them learn whatever gives the best final result to feed into the final output layer
- As well as the networks already seen, other architectures (topology) are possible
  - More/less nodes per layer
  - More layers
  - Once again, layer 2 has three hidden units, layer 3 has 2 hidden units by the time you get to the output layer you get very interesting non-linear hypothesis



- Some of the intuitions here are complicated and hard to understand
  - In the following lectures we're going to go though a detailed example to understand how to do non-linear analysis

Neural network example - computing a complex, nonlinear function of the input

- Non-linear classification: XOR/XNOR
  - x_1, x_2 are binary



- Example on the right shows a simplified version of the more complex problem we're dealing with (on the left)
- We want to learn a non-linear decision boundary to separate the positive and negative examples

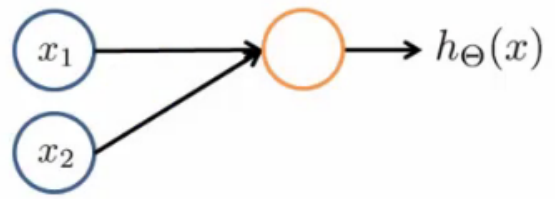
y = x_1 XOR x_2
x_1 XNOR x_2

Where XNOR = NOT (x_1 XOR x_2)

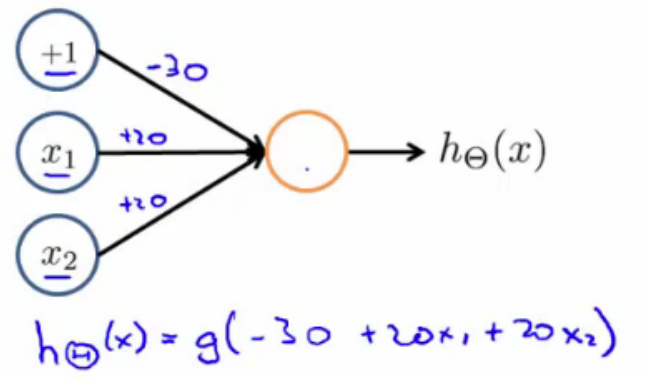
- Positive examples when both are true and both are false
  - Let's start with something a little more straight forward...
  - Don't worry about how we're determining the weights (Θ values) for now - just get a flavor of how NNs work

Neural Network example 1: AND function

- Simple first example

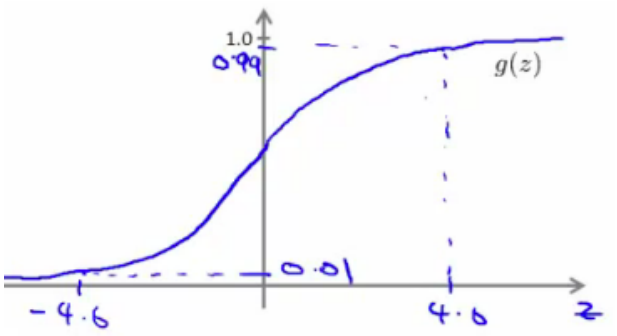


- Can we get a one-unit neural network to compute this logical AND function? (*probably...*)
  - Add a bias unit
  - Add some weights for the networks
    - What are weights?
      - Weights are the parameter values which multiply into the input nodes (i.e. Θ)



- Sometimes it's convenient to add the weights into the diagram
  - These values are in fact just the Θ parameters so
    - $\Theta_{10}^{1} = -30$
    - $\Theta_{11}^{1} = 20$
    - $\Theta_{12}^{1} = 20$

- To use our original notation
- Look at the four input values



x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

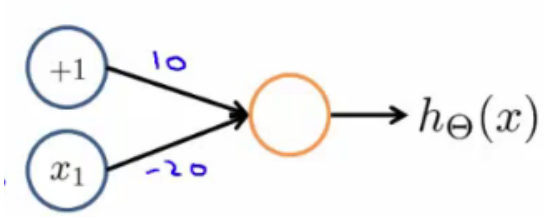
$h_\Theta(x) \approx x_1$ AND x_2

Sigmoid function (reminder)

- So, as we can see, when we evaluate each of the four possible input, only (1,1) gives a positive output

Neural Network example 2: NOT function

- How about negation?



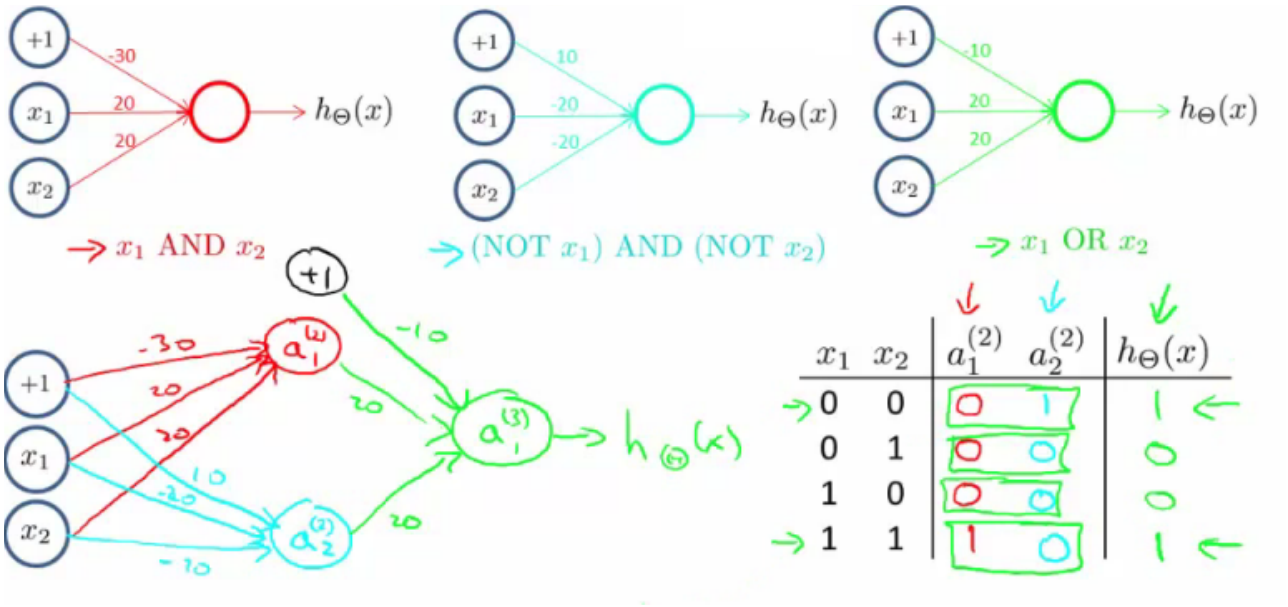
x_1	$h_\Theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

- Negation is achieved by putting a large negative weight in front of the variable you want to negative

Neural Network example 3: XNOR function

- So how do we make the XNOR function work?
  - XNOR is short for NOT XOR
    - i.e. NOT an exclusive or, so either go big (1,1) or go home (0,0)
  - So we want to structure this so the input which produce a positive output are
    - AND (i.e. both true)

      OR
    - Neither (which we can shortcut by saying not only one being true)
- So we combine these into a neural network as shown below;



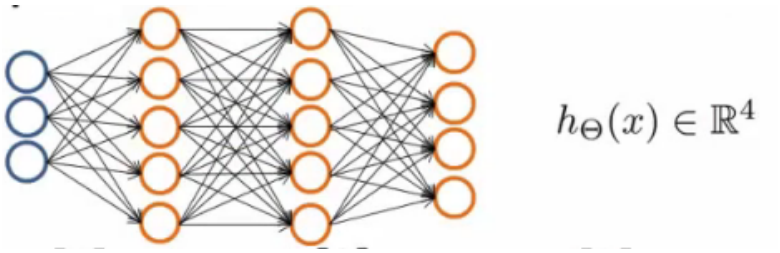
- Simplez!

Neural network intuition - handwritten digit classification

- Yann LeCun = machine learning pioneer
- Early machine learning system was postcode reading
    - Hilarious music, impressive demonstration!

Multiclass classification

- Multiclass classification is, unsurprisingly, when you distinguish between more than two categories (i.e. more than 1 or 0)
- With handwritten digital recognition problem - 10 possible categories (0-9)
    - How do you do that?
    - Done using an extension of one vs. all classification
- Recognizing pedestrian, car, motorbike or truck
    - Build a neural network with four output units
    - Output a vector of four numbers
        - 1 is 0/1 pedestrian
        - 2 is 0/1 car
        - 3 is 0/1 motorcycle
        - 4 is 0/1 truck
    - When image is a pedestrian get [1,0,0,0] and so on
- Just like one vs. all described earlier
    - Here we have four logistic regression classifiers



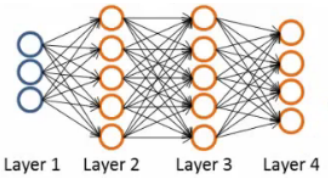
- Training set here is images of our four classifications
  - While previously we'd written y as an integer {1,2,3,4}
  - Now represent y as
- $y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

09: Neural Networks - Learning

Previous Next Index

Neural network cost function

- NNs - one of the most powerful learning algorithms
  - Is a learning algorithm for fitting the derived parameters given a training set
  - Let's have a first look at a neural network cost function
- Focus on application of NNs for classification problems
- Here's the set up
  - Training set is $\{(x^1, y^1), (x^2, y^2), (x^3, y^3) \ldots (x^n, y^m)$
  - L = number of layers in the network
    - In our example below L = 4
  - s_l = number of units (not counting bias unit) in layer l



- So here
  - l = 4
  - s_1 = 3
  - s_2 = 5
  - s_3 = 5
  - s_4 = 4

Types of classification problems with NNs

- Two types of classification, as we've previously seen
- **Binary classification**
  - 1 output (0 or 1)
  - So single output node - value is going to be a real number
  - k = 1
    - NB k is number of units in output layer
  - s_L = 1
- **Multi-class classification**
  - k distinct classifications
  - Typically k is greater than or equal to three
  - If only two just go for binary
  - s_L = k
  - So y is a k-dimensional vector of real numbers

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

Cost function for neural networks

- The (regularized) logistic regression cost function is as follows;

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

- For neural networks our cost function is a generalization of this equation above, so instead of one output we

generate k outputs

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

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

- Our cost function now outputs a k dimensional vector
  - $h_\Theta(x)$ is a k dimensional vector, so $h_\Theta(x)_i$ refers to the ith value in that vector
- Costfunction $J(\Theta)$ is
  - [-1/m] times a sum of a similar term to which we had for logic regression
  - But now this is also a sum from k = 1 through to K (K is number of output nodes)
    - Summation is a sum over the k output units - i.e. for each of the possible classes
    - So if we had 4 output units then the sum is k = 1 to 4 of the logistic regression over each of the four output units in turn
  - This looks really complicated, but it's not so difficult
    - We don't sum over the bias terms (hence starting at 1 for the summation)
      - Even if you do and end up regularizing the bias term this is not a big problem
    - Is just summation over the terms

Woah there - lets take a second to try and understand this!

- There are basically two halves to the neural network logistic regression cost function

First half

$$-\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)}) \log(1-(h_\Theta(x^{(i)}))_k)\right]$$

- This is just saying
  - For each training data example (i.e. 1 to m - the first summation)
    - Sum for each position in the output vector
- This is an average sum of logistic regression

Second half

$$\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

- This is a massive regularization summation term, which I'm not going to walk through, but it's a fairly straightforward triple nested summation
- This is also called a **weight decay** term
- As before, the lambda value determines the important of the two halves

- The regularization term is similar to that in logistic regression
- So, we have a cost function, but *how* do we minimize this bad boy?!

Summary of what's about to go down

The following section is, I think, the most complicated thing in the course, so I'm going to take a second to explain the general idea of what we're going to do;

- We've already described **forward propagation**
  - This is the algorithm which takes your neural network and the initial input into that network and pushes the input through the network
    - It leads to the generation of an output hypothesis, which may be a single real number, but can also be a vector
- We're now going to describe **back propagation**
  - Back propagation basically takes the output you got from your network, compares it to the real value (y) and calculates how wrong the network was (i.e. how wrong the parameters were)
  - It then, using the error you've just calculated, back-calculates the error associated with each unit from the preceding layer (i.e. layer L - 1)

- This goes on until you reach the input layer (where obviously there is no error, as the activation is the input)
- These "error" measurements for each unit can be used to calculate the **partial derivatives**
  - Partial derivatives are the bomb, because gradient descent needs them to minimize the cost function
- We use the partial derivatives with gradient descent to try minimize the cost function and update all the Θ values
- This repeats until gradient descent reports convergence

- A few things which are good to realize from the get go
  - There is a Θ matrix for each layer in the network
    - This has each node in layer l as one dimension and each node in l+1 as the other dimension
  - Similarly, there is going to be a Δ matrix for each layer
    - This has each node as one dimension and each training data example as the other

Back propagation algorithm

- We previously spoke about the neural network cost function
- Now we're going to deal with **back propagation**
  - Algorithm used to minimize the cost function, as it **allows us to calculate partial derivatives**!

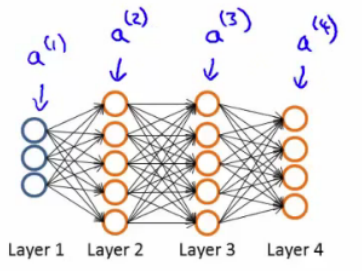
$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

- The cost function used is shown above

  - We want to find parameters Θ which minimize J(Θ)
  - To do so we can use one of the algorithms already described such as
    - Gradient descent
    - Advanced optimization algorithms
- To minimize a cost function we just write code which computes the following
  - **J(Θ)**
    - i.e. the cost function itself!
    - Use the formula above to calculate this value, so we've done that
  - **Partial derivative terms**
    - So now we need some way to do that
      - This is not trivial! Θ is indexed in three dimensions because we have separate parameter values for each node in each layer going to each node in the following layer
      - i.e. each layer has a Θ matrix associated with it!
        - We want to calculate the partial derivative Θ with respect to a single parameter

          $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$
    - Remember that the partial derivative term we calculate above is a REAL number (not a vector or a matrix)
      - Θ is the input parameters
        - Θ^1 is the matrix of weights which define the function mapping from layer 1 to layer 2
        - $\Theta_{10}{}^1$ is the real number parameter which you multiply the bias unit (i.e. 1) with for the bias unit input into the first unit in the second layer
        - $\Theta_{11}{}^1$ is the real number parameter which you multiply the first (real) unit with for the first input into the first unit in the second layer
        - $\Theta_{21}{}^1$ is the real number parameter which you multiply the first (real) unit with for the first input into the second unit in the second layer
        - As discussed, $\Theta_{ij}{}^l$ i
          - i here represents the unit in layer l+1 you're mapping to (destination node)
          - j is the unit in layer l you're mapping from (origin node)
          - l is the layer your mapping from (to layer l+1) (origin layer)
          - NB
            - *The terms destination node, origin node and origin layer are terms I've made up!*
      - So - this partial derivative term is
        - The partial derivative of a 3-way indexed dataset with respect to a real number (which is one of the values in that dataset)
  - **Gradient computation**

- One training example
- Imagine we just have a single pair (x,y) - entire training set
- How would we deal with this example?
- The forward propagation algorithm operates as follows
  - **Layer 1**
    - $a^1 = x$
    - $z^2 = \Theta^1 a^1$
  - **Layer 2**
    - $a^2 = g(z^2)$ (add a_0^2)
    - $z^3 = \Theta^2 a^2$
  - **Layer 3**
    - $a^3 = g(z^3)$ (add a_0^3)
    - $z^4 = \Theta^3 a^3$
  - **Output**
    - $a^4 = h_\Theta(x) = g(z^4)$



- This is the vectorized implementation of forward propagation
  - Lets compute activation values sequentially (below just re-iterates what we had above!)

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

What is back propagation?

- Use it to compute the partial derivatives
- Before we dive into the mechanics, let's get an idea regarding the intuition of the algorithm
  - For each node we can calculate (δ_j^l) - this is **the error of node j in layer l**
    - If we remember, a_j^l is the activation of node j in layer l
    - Remember the activation is a totally calculated value, so we'd expect there to be some error compared to the "real" value
      - The delta term captures this error
      - But the problem here is, "what is this 'real' value, and how do we calculate it?!"
        - The NN is a totally artificial construct
        - The only "real" value we have is our actual classification (our y value) - so that's where we start
- If we use our example and look at the fourth (output) layer, we can first calculate
  - $\delta_j^4 = a_j^4 - y_j$
    - [Activation of the unit] - [the actual value observed in the training example]

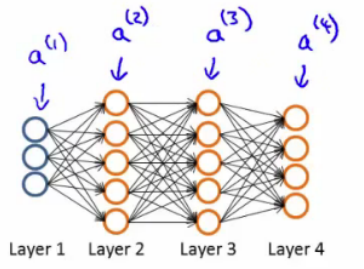
        - We could also write a_j^4 as $h_\Theta(x)_j$
            - Although I'm not sure why we would?
    - This is an individual example implementation
- Instead of focussing on each node, let's think about this as a vectorized problem
    - $\delta^4 = a^4 - y$
        - So here δ^4 is the vector of errors for the 4th layer
        - a^4 is the vector of activation values for the 4th layer
- With δ^4 calculated, we can determine the error terms for the other layers as follows;

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

- Taking a second to break this down
    - Θ^3 is the vector of parameters for the 3->4 layer mapping
    - δ^4 is (as calculated) the error vector for the 4th layer
    - $g'(z^3)$ is the first derivative of the activation function g evaluated by the input values given by z^3
        - You can do the calculus if you want (...), but when you calculate this derivative you get
        - $g'(z^3) = a^3 .* (1 - a^3)$
    - So, more easily
        - **$\delta^3 = (\Theta^3)^T \delta^4 .* (a^3 .* (1 - a^3))$**
    - **.*** is the element wise multiplication between the two vectors
        - Why element wise? Because this is essentially an extension of individual values in a vectorized implementation, so element wise multiplication gives that effect
        - We highlighted it just in case you think it's a typo!

Analyzing the mathematics



- And if we take a second to consider the vector dimensionality (with our example above [3-5-5-4])
    - **Θ^3** = is a matrix which is [4 X 5] (if we don't include the bias term, 4 X 6 if we do)
        - **$(\Theta^3)^T$** = therefore, is a [5 X 4] matrix
    - **δ^4** = is a 4x1 vector
    - So when we multiply a [5 X 4] matrix with a [4 X 1] vector we get a [5 X 1] vector
    - Which, low and behold, is the same dimensionality as the **a^3** vector, meaning we can run our pairwise multiplication

- For δ^3 when you calculate the derivative terms you get
  $a^3 .* (1 - a^3)$
- Similarly For δ^2 when you calculate the derivative terms you get
  $a^2 .* (1 - a^2)$
    - So to calculate δ^2 we do
      $\delta^2 = (\Theta^2)^T \delta^3 .* (a^2 .* (1 - a^2))$
- There's no δ^1 term
    - Because that was the input!

Why do we do this?

- We do all this to get all the δ terms, and we want the δ terms because through a very complicated derivation you can use δ to get the partial derivative of Θ with respect to individual parameters (if you ignore regularization, or regularization is 0, which we deal with later)

- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ = $a_j^l \delta_i^{(l+1)}$
- By doing back propagation and computing the delta terms you can then compute the **partial derivative terms**
  - We need the partial derivatives to minimize the cost function!

Putting it all together to get the partial derivatives!

- What is really happening - lets look at a more complex example
- Training set of m examples

  Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- **First**, set the delta values

  Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j)
  - Set equal to 0 for all values
  - Eventually these Δ values will be used to compute the partial derivative
    - Will be used as accumulators for computing the partial derivatives
- **Next**, loop through the training set

  For $i = 1$ to m
  - i.e. for each example in the training set (dealing with each example as (x,y)
  - Set a^1 (activation of input layer) = x^i
  - **Perform forward propagation** to compute a^l for each layer (l = 1,2, ... L)
    - i.e. run forward propagation
  - **Then**, use the output label for the specific example we're looking at to calculate δ^L where $\delta^L = a^L - y^i$
    - So we initially calculate the delta value for the output layer
    - Then, using **back propagation** we move back through the network from layer L-1 down to layer
  - Finally, use Δ to accumulate the partial derivative terms

    $\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
  - Note here
    - l = layer
    - j = node in that layer
    - i = the error of the affected node in the target layer
  - You can vectorize the Δ expression too, as

    $\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.
- **Finally**
  - After executing the body of the loop, exit the for loop and compute

    $D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

    $D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)}$ if $j = 0$
    - When j = 0 we have no regularization term
- At the end of ALL this
  - You've calculated all the *D* terms above using Δ
    - NB - each D term above is a real number!
  - We can show that each D is equal to the following
    - $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$
  - We have calculated the partial derivative for each parameter
    - We can then use these in gradient descent or one of the advanced optimization algorithms
- Phew!
  - What a load of hassle!

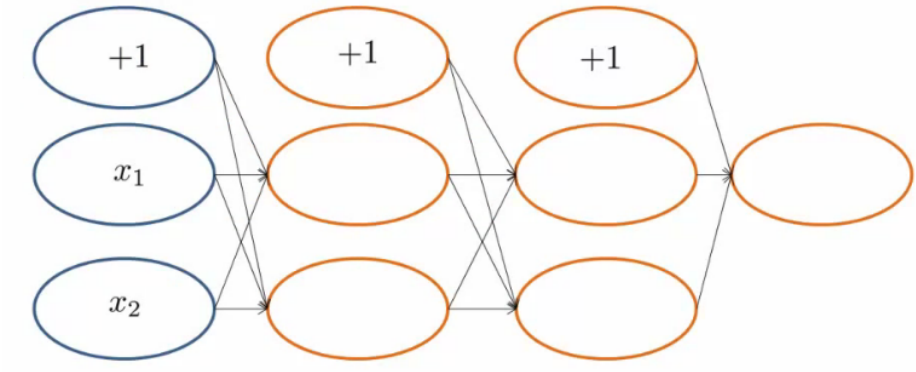
Back propagation intuition

- Some additionally back propagation notes
  - In case you found the preceding unclear, which it shouldn't be as it's fairly heavily modified with my own explanatory notes
- Back propagation is hard(ish...)

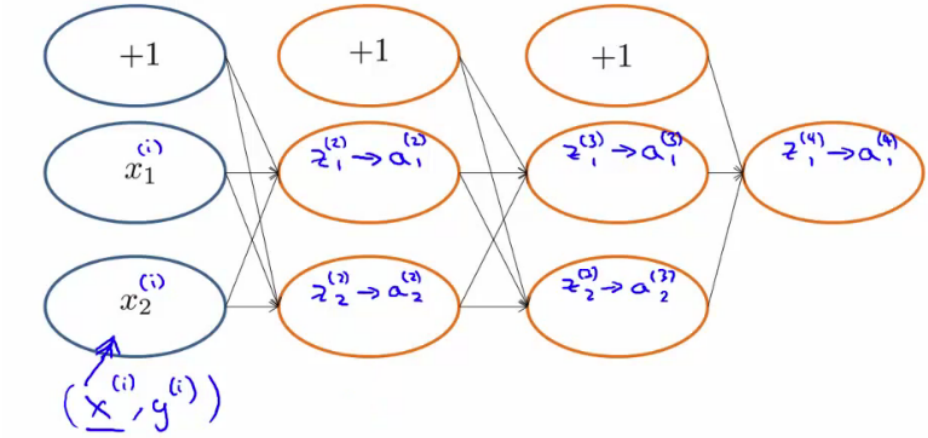
- But don't let that discourage you
- It's hard in as much as it's confusing - it's not difficult, just complex

- Looking at mechanical steps of back propagation

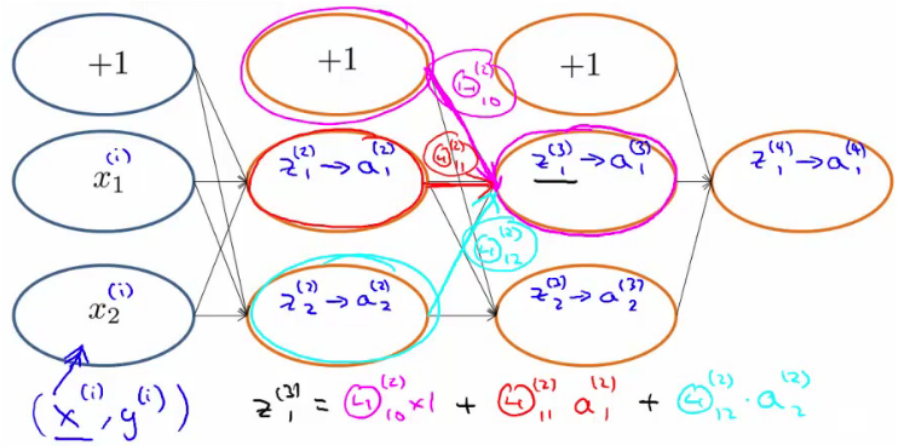
Forward propagation with pictures!



- Feeding input into the input layer (x^i, y^i)
  - Note that x and y here are vectors from 1 to n where n is the number of features
    - So above, our data has two features (hence x_1 and x_2)
- With out input data present we use **forward propagation**



- The sigmoid function applied to the z values gives the activation values
  - Below we show exactly how the z value is calculated for an example



Back propagation

- With forwardprop done we move on to do back propagation
- Back propagation is doing something very similar to forward propagation, but backwards
  - Very similar though
- Let's look at the cost function again...
  - Below we have the cost function if there is a single output (i.e. binary classification)

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

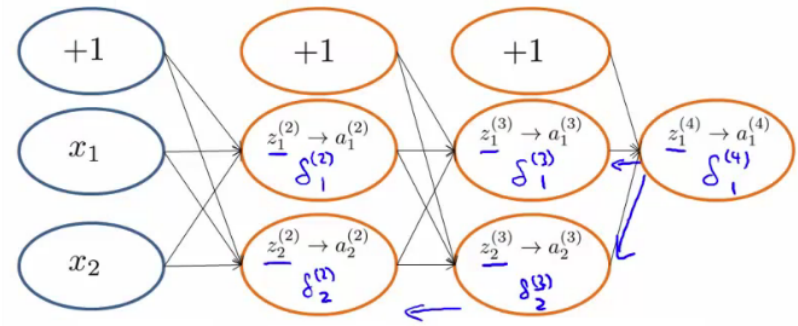
- This function cycles over each example, so the cost for one example really boils down to this

$$\text{cost(i)} = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)})$$

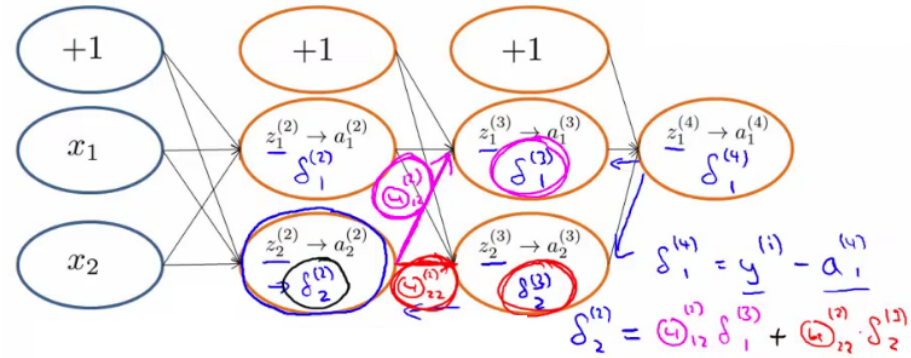
- Which, we can think of as a sigmoidal version of the squared difference (check out the derivation if you don't believe me)
  - So, basically saying, "how well is the network doing on example *i* "?
- We can think about a δ term on a unit as the "error" of cost for the activation value associated with a unit
  - More formally (*don't worry about this...*), δ is

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost(i)}$$

    - Where cost is as defined above
    - Cost function is a function of y value and the hypothesis function
- So - for the output layer, back propagation sets the δ value as [a - y]
  - Difference between activation and actual value
- We then propagate these values backwards;



- Looking at another example to see *how* we actually calculate the delta value;



- So, in effect,
  - Back propagation calculates the δ, and those δ values are the weighted sum of the next layer's delta values, weighted by the parameter associated with the links
  - Forward propagation calculates the activation (a) values, which

- Depending on how you implement you may compute the delta values of the bias values
  - However, these aren't actually used, so it's a bit inefficient, but not a lot more!

Implementation notes - unrolling parameters (matrices)

- Needed for using advanced optimization routines

```
function [jVal, gradient] = costFunction(theta)
  ...
optTheta = fminunc(@costFunction, initialTheta, options)
```

- Is the MATLAB/octave code
  - But theta is going to be matrices
- fminunc takes the costfunction and initial theta values
  - These routines assume theta is a parameter vector
  - Also assumes the gradient created by costFunction is a vector
- For NNs, our parameters are matrices
  - e.g.

Neural Network (L=4):

$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (`Theta1, Theta2, Theta3`)

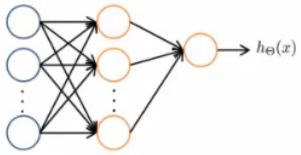
$D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (`D1, D2, D3`)

Example

$s_1 = 10, s_2 = 10, s_3 = 1$

$\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$

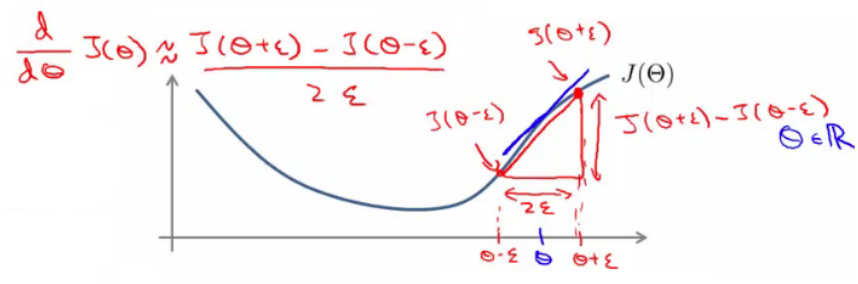
$D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$



- Use the `thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];` notation to unroll the matrices into a long vector
- To go back you use
  - `Theta1 = resape(thetaVec(1:110), 10, 11)`

Gradient checking

- Backpropagation has a lot of details, small bugs can be present and ruin it :-(
  - This may mean it looks like J(Θ) is decreasing, but in reality it may not be decreasing by as much as it should
- So using a numeric method to check the gradient can help diagnose a bug
  - Gradient checking helps make sure an implementation is working correctly
- **Example**
  - Have an function J(Θ)
  - Estimate derivative of function at point Θ (where Θ is a real number)
  - How?
    - Numerically
      - Compute Θ + ε
      - Compute Θ - ε
      - Join them by a straight line
      - Use the slope of that line as an approximation to the derivative



- Usually, epsilon is pretty small (0.0001)
  - If epsilon becomes REALLY small then the term BECOMES the slopes derivative
- The is the two sided difference (as opposed to one sided difference, which would be J(Θ + ε) - J(Θ) /ε
- If Θ is a vector with n elements we can use a similar approach to look at the partial derivatives

$$\frac{\partial}{\partial\theta_1}J(\theta) \approx \frac{J(\theta_1+\epsilon,\theta_2,\theta_3,\ldots,\theta_n)-J(\theta_1-\epsilon,\theta_2,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial\theta_2}J(\theta) \approx \frac{J(\theta_1,\theta_2+\epsilon,\theta_3,\ldots,\theta_n)-J(\theta_1,\theta_2-\epsilon,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial\theta_n}J(\theta) \approx \frac{J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n+\epsilon)-J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n-\epsilon)}{2\epsilon}$$

- So, in octave we use the following code the numerically compute the derivatives

```
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;
```

- So on each loop thetaPlus = theta except for thetaPlus(i)
  - Resets thetaPlus on each loop
- Create a vector of partial derivative approximations
- Using the vector of gradients from backprop (DVec)
  - Check that gradApprox is basically equal to DVec
  - Gives confidence that the Backproc implementation is correc
- Implementation note
  - Implement back propagation to compute DVec
  - Implement numerical gradient checking to compute gradApprox
  - Check they're basically the same (up to a few decimal places)
  - Before using the code for learning turn off gradient checking
    - Why?
      - GradAprox stuff is very computationally expensive
      - In contrast backprop is much more efficient (just more fiddly)

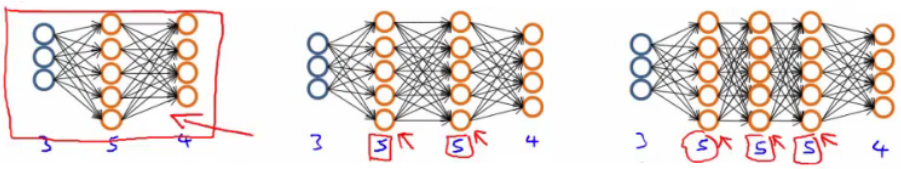
Random initialization

- Pick random small initial values for all the theta values
  - If you start them on zero (which does work for linear regression) then the algorithm fails - all activation values for each layer are the same
- So chose random values!

- Between 0 and 1, then scale by epsilon (where epsilon is a constant)

Putting it all together

- **1) - pick a network architecture**
  - Number of
    - **Input units** - number of dimensions x (dimensions of feature vector)
    - **Output units** - number of classes in classification problem
    - **Hidden units**
      - Default might be
        - 1 hidden layer
      - Should probably have
        - Same number of units in each layer
        - Or 1.5-2 x number of input features
      - Normally
        - More hidden units is better
        - But more is more computational expensive
  - We'll discuss architecture more later



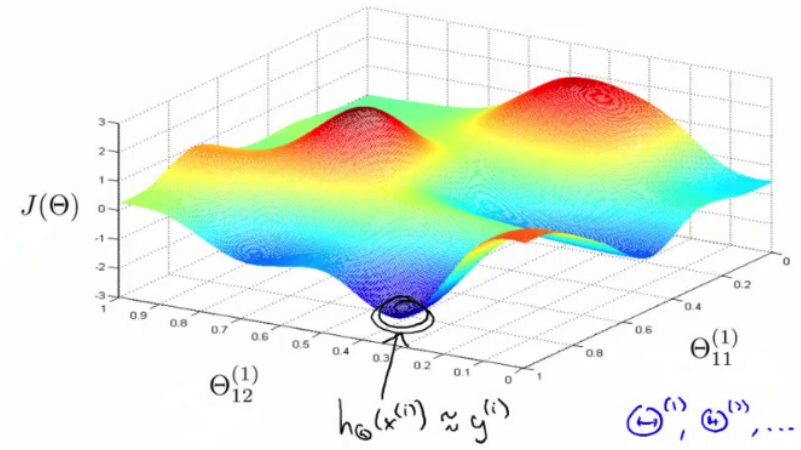
- **2) - Training a neural network**
  - **2.1)** Randomly initialize the weights
    - Small values near 0
  - **2.2)** Implement forward propagation to get $h_\Theta(x)^i$ for any x^i
  - **2.3)** Implement code to compute the cost function $J(\Theta)$
  - **2.4)** Implement back propagation to compute the partial derivatives
  - General implementation below

```
for i = 1:m {
     Forward propagation on (x^i, y^i) --> get activation (a) terms
     Back propagation on (x^i, y^i) --> get delta (δ) terms
     Compute Δ := Δ^l + δ^(l+1)(a^l)^T
}

With this done compute the partial derivative terms
```

  - Notes on implementation
    - Usually done with a for loop over training examples (for forward and back propagation)
    - *Can* be done without a for loop, but this is a much more complicated way of doing things
    - Be careful
- **2.5)** Use gradient checking to compare the partial derivatives computed using the above algorithm and numerical estimation of gradient of $J(\Theta)$
  - Disable the gradient checking code for when you actually run it
- **2.6)** Use gradient descent or an advanced optimization method with back propagation to try to minimize $J(\Theta)$ as a function of parameters Θ
  - Here $J(\Theta)$ is non-convex
    - Can be susceptible to local minimum
    - In practice this is not usually a huge problem

- Can't guarantee programs with find global optimum should find good local optimum at least



- e.g. above pretending data only has two features to easily display what's going on
  - Our minimum here represents a hypothesis output which is pretty close to y
  - If you took one of the peaks hypothesis is far from y
- Gradient descent will start from some random point and move downhill
  - Back propagation calculates gradient down that hill

10: Advice for applying Machine Learning

Previous Next Index

Deciding what to try next

- We now know many techniques
  - But, there is a big difference between someone who knows an algorithm vs. someone less familiar and doesn't understand how to apply them
  - Make sure you know how to chose the best avenues to explore the various techniques
  - Here we focus deciding what avenues to try

Debugging a learning algorithm

- So, say you've implemented regularized linear regression to predict housing prices

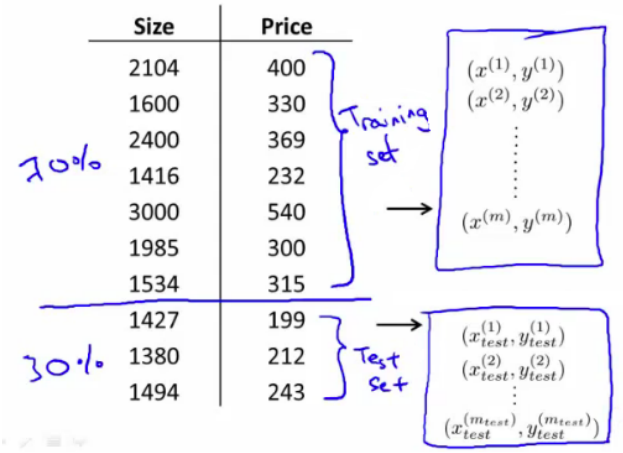
$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{m}\theta_j^2\right]$$

  - Trained it
  - But, when you test on new data you find it makes unacceptably large errors in its predictions
  - :-(
- What should you try next?
  - There are many things you can do;
    - **Get more training data**
      - Sometimes more data doesn't help
      - Often it does though, although you should always do some preliminary testing to make sure more data will actually make a difference (discussed later)
    - **Try a smaller set a features**
      - Carefully select small subset
      - You can do this by hand, or use some dimensionality reduction technique (e.g. PCA - we'll get to this later)
    - **Try getting additional features**
      - Sometimes this isn't helpful
      - LOOK at the data
      - Can be very time consuming
    - **Adding polynomial features**
      - You're grasping at straws, aren't you...
    - **Building your own, new, better features** based on your knowledge of the problem
      - Can be risky if you accidentally over fit your data by creating new features which are inherently specific/relevant to your training data
    - **Try decreasing or increasing λ**
      - Change how important the regularization term is in your calculations
  - These changes can become MAJOR projects/headaches (6 months +)
    - Sadly, most common method for choosing one of these examples is to go by gut feeling (randomly)
    - Many times, see people spend huge amounts of time only to discover that the avenue is fruitless
      - No apples, pears, or any other fruit. Nada.
  - There are some simple techniques which can let you rule out half the things on the list
    - Save you a lot of time!
- **Machine learning diagnostics**
  - Tests you can run to see what is/what isn't working for an algorithm
  - See what you can change to improve an algorithm's performance
  - These can take time to implement and understand (week)
    - But, they can also save you spending months going down an avenue which will *never* work

Evaluating a hypothesis

- When we fit parameters to training data, try and minimize the error
  - We might think a low error is good - doesn't necessarily mean a good parameter set
    - Could, in fact, be indicative of overfitting

        - This means you model will fail to generalize
    - How do you tell if a hypothesis is overfitting?
        - Could plot $h_\theta(x)$
        - But with lots of features may be impossible to plot
- Standard way to evaluate a hypothesis is
    - Split data into two portions
        - 1st portion is **training set**
        - 2nd portion is **test set**
    - Typical split might be 70:30 (training:test)



- NB if data is ordered, send a random percentage
    - (Or randomly order, then send data)
    - Data is typically ordered in some way anyway
- So a typical **train and test scheme** would be
    - 1) Learn parameters θ from training data, minimizing J(θ) using 70% of the training data]
    - 2) Compute the test error
        - $J_{test}(\theta)$ = average square error as measured on the test set

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} \left(h_\theta(x^{(i)}_{test}) - y^{(i)}_{test} \right)^2$$

    - This is the definition of the **test set error**
- What about if we were using logistic regression
    - The same, learn using 70% of the data, test with the remaining 30%

$$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y^{(i)}_{test} \log h_\theta(x^{(i)}_{test}) + (1 - y^{(i)}_{test}) \log h_\theta(x^{(i)}_{test})$$

    - Sometimes there a better way - misclassification error (0/1 misclassification)
        - We define the error as follows

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

        - Then the test error is

$$\text{Test error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\theta(x^{(i)}_{test}), y^{(i)}_{test})$$

            - i.e. its the fraction in the test set the hypothesis mislabels
- These are the standard techniques for evaluating a learned hypothesis

Model selection and training validation test sets

- How to chose regularization parameter or degree of polynomial (**model selection problems**)
- We've already seen the problem of overfitting
  - More generally, this is why training set error is a poor predictor of hypothesis accuracy for new data (generalization)
- Model selection problem
  - Try to chose the degree for a polynomial to fit data
    1. $h_\theta(x) = \theta_0 + \theta_1 x$
    2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$
    3. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3$
    ⋮
    10. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}$
  - d = what degree of polynomial do you want to pick
    - An additional parameter to try and determine your training set
      - d =1 (linear)
      - d=2 (quadratic)
      - ...
      - d=10
    - Chose a model, fit that model and get an estimate of how well you hypothesis will generalize
  - You could
    - Take model 1, minimize with training data which generates a parameter vector θ^1 (where d =1)
    - Take mode 2, do the same, get a *different* θ^2 (where d = 2)
    - And so on
    - Take these parameters and look at the test set error for each using the previous formula
      - $J_{test}(\theta^1)$
      - $J_{test}(\theta^2)$
      - ...
      - $J_{test}(\theta^{10})$
  - You could then
    - See which model has the lowest test set error
  - Say, for example, d=5 is the lowest
    - Now take the d=5 model and say, how well does it generalize?
      - You could use $J_{test}(\theta^5)$
      - BUT, this is going to be an optimistic estimate of generalization error, because our parameter is fit to that test set (i.e. specifically chose it because the test set error is small)
      - So not a good way to evaluate if it will generalize
  - To address this problem, we do something a bit different for model selection
- Improved model selection
  - Given a training set instead split into three pieces
    - 1 - **Training set** (60%) - m values
    - 2 - **Cross validation** (CV) set (20%)m_{cv}
    - 3 - **Test set** (20%) m_{test}
  - As before, we can calculate
    - Training error
    - Cross validation error
    - Test error

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

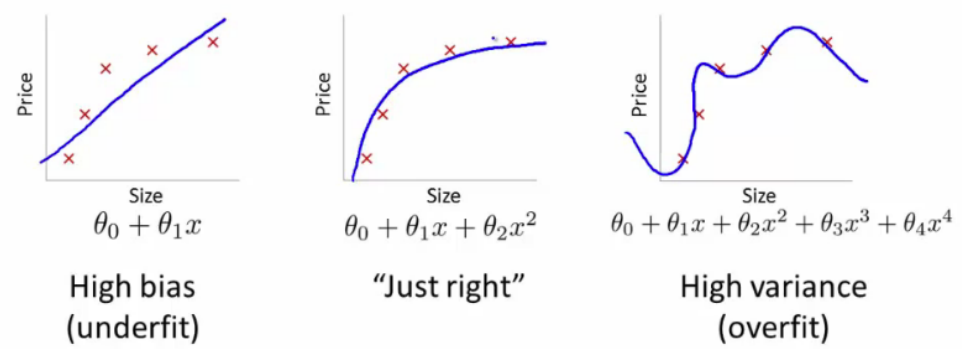
Cross Validation error: $J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$

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

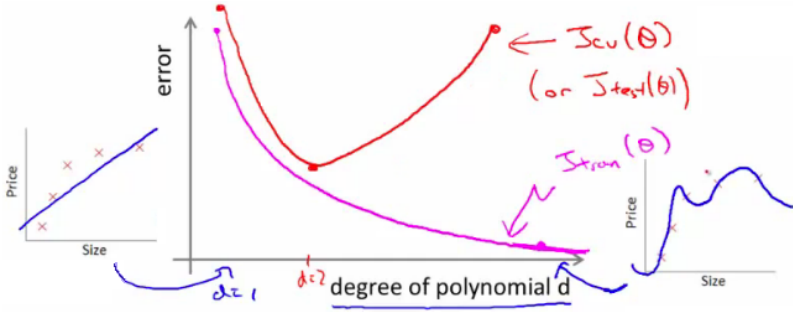
- So
  - Minimize cost function for each of the models as before
  - Test these hypothesis on the cross validation set to generate the cross validation error
  - Pick the hypothesis with the lowest cross validation error
    - e.g. pick θ^5
  - Finally
    - Estimate generalization error of model using the test set
- Final note
  - In machine learning as practiced today - many people will select the model using the test set and then check the model is OK for generalization using the test error (which we've said is bad because it gives a bias analysis)
    - With a MASSIVE test set this is maybe OK
  - But considered much better practice to have separate training and validation sets

Diagnosis - bias vs. variance

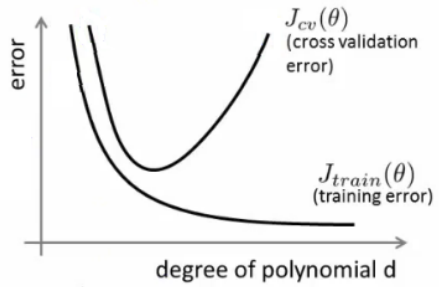
- If you get bad results usually because of one of
  - **High bias** - under fitting problem
  - **High variance** - over fitting problem
- Important to work out which is the problem
  - Knowing which will help let you improve the algorithm
- Bias/variance shown graphically below



- The degree of a model will increase as you move towards overfitting
- Lets define training and cross validation error as before
- Now plot
  - x = degree of polynomial d
  - y = error for both training and cross validation (two lines)
    - CV error and test set error will be very similar



    - This plot helps us understand the error
  - We want to minimize both errors
    - Which is why that d=2 model is the sweet spot
- How do we apply this for diagnostics
  - If cv error is high we're either at the high or the low end of d



- if d is too small --> this probably corresponds to a high bias problem
- if d is too large --> this probably corresponds to a high variance problem

- **For the high bias case, we find both cross validation and training error are high**
  - Doesn't fit training data well
  - Doesn't generalize either
- **For high variance, we find the cross validation error is high but training error is low**
  - So we suffer from overfitting (training is low, cross validation is high)
  - i.e. training set fits well
  - But generalizes poorly

Regularization and bias/variance

- How is bias and variance effected by regularization?

Linear regression with regularization

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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

- The equation above describes fitting a high order polynomial with regularization (used to keep parameter values small)
  - Consider three cases
    - **λ = large**
      - All θ values are heavily penalized
      - So most parameters end up being close to zero
      - So hypothesis ends up being close to 0
      - So **high bias -> under fitting data**
    - **λ = intermediate**
      - Only this values gives the fitting which is reasonable
    - **λ = small**
      - Lambda = 0
      - So we make the regularization term 0
      - So **high variance -> Get overfitting** (minimal regularization means it obviously doesn't do what it's meant to)

- How can we automatically chose a good value for λ?
  - To do this we define another function $J_{train}(\theta)$ which is the optimization function *without* the regularization term (average squared errors)

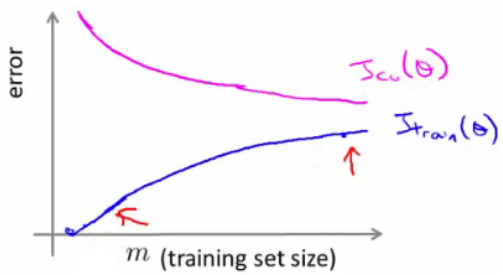
$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

  - Define cross validation error and test set errors as before (i.e. without regularization term)
    - So they are 1/2 average squared error of various sets

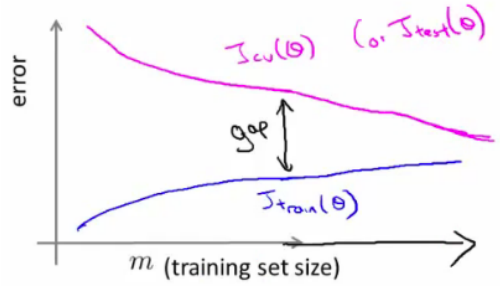
- **Choosing λ**
  - Have a set or range of values to use
  - Often increment by factors of 2 so
    - model(1)= λ = 0
    - model(2)= λ = 0.01
    - model(3)= λ = 0.02
    - model(4) = λ = 0.04
    - model(5) = λ = 0.08
      .
      .
      .
    - model(p) = λ = 10
  - This gives a number of models which have different λ
  - With these models
    - Take each one (p^{th})
    - Minimize the cost function
    - This will generate some parameter vector
      - Call this $\theta^{(p)}$
    - So now we have a set of parameter vectors corresponding to models with different λ values
  - Take all of the hypothesis and use the cross validation set to validate them
    - Measure average squared error on cross validation set
    - Pick the model which gives the lowest error
    - Say we pick $\theta^{(5)}$
  - Finally, take the one we've selected ($\theta^{(5)}$) and test it with the test set
- **Bias/variance as a function of λ**
  - Plot λ vs.
    - J_{train}
      - When λ is small you get a small value (regularization basically goes to 0)
      - When λ is large you get a large vale corresponding to high bias
    - J_{cv}
      - When λ is small we see high variance
        - Too small a value means we over fit the data
      - When λ is large we end up underfitting, so this is bias
        - So cross validation error is high
  - Such a plot can help show you you're picking a good value for λ

Learning curves

- A learning curve is often useful to plot for algorithmic sanity checking or improving performance
- What is a learning curve?
  - Plot J_{train} (average squared error on training set) or J_{cv} (average squared error on cross validation set)
  - Plot against m (number of training examples)
    - m is a constant
    - So artificially reduce m and recalculate errors with the smaller training set sizes
  - J_{train}
    - Error on smaller sample sizes is smaller (as less variance to accommodate)
    - So as m grows error grows
  - J_{cv}
    - Error on cross validation set
    - When you have a tiny training set your generalize badly
    - But as training set grows your hypothesis generalize better
    - So cv error will decrease as m increases



- What do these curves look like if you have
  - **High bias**
    - e.g. setting straight line to data
    - J_{train}
      - Training error is small at first and grows
      - Training error becomes close to cross validation
      - So the performance of the cross validation and training set end up being similar (but very poor)
    - J_{cv}
      - Straight line fit is similar for a few vs. a lot of data
      - So it doesn't generalize any better with lots of data because the function just doesn't fit the data
        - No increase in data will help it fit
    - The problem with high bias is because cross validation and training error are both high
    - Also implies that if a learning algorithm as high bias as we get more examples the cross validation error doesn't decrease
      - **So if an algorithm is already suffering from high bias, more data does not help**
      - So knowing if you're suffering from high bias is good!
      - In other words, high bias is a problem with the underlying way you're modeling your data
        - So more data won't improve that model
        - It's too simplistic
  - **High variance**
    - e.g. high order polynomial
    - J_{train}
      - When set is small, training error is small too
      - As training set sizes increases, value is still small
      - But slowly increases (in a near linear fashion)
      - Error is still low
    - J_{cv}
      - Error remains high, even when you have a moderate number of examples
      - Because the problem with high variance (overfitting) is your model doesn't generalize
    - An indicative diagnostic that you have high variance is that there's a big gap between training error and cross validation error
    - If a learning algorithm is suffering from high variance, more data is probably going to help



- **So if an algorithm is already suffering from high variance, more data will probably help**
  - Maybe
- These are clean curves
- In reality the curves you get are far dirtier
- But, learning curve plotting can help diagnose the problems your algorithm will be suffering from

What to do next (revisited)

- How do these ideas help us chose how we approach a problem?
  - Original example
    - Trained a learning algorithm (regularized linear regression)
    - But, when you test on new data you find it makes unacceptably large errors in its predictions
    - What should try next?
  - How do we decide what to do?
    - **Get more examples** --> helps to fix high variance
      - Not good if you have high bias
    - **Smaller set of features** --> fixes high variance (overfitting)
      - Not good if you have high bias
    - **Try adding additional features** --> fixes high bias (because hypothesis is too simple, make hypothesis more specific)
    - **Add polynomial terms** --> fixes high bias problem
    - **Decreasing λ** --> fixes high bias
    - **Increases λ** --> fixes high variance
- Relating it all back to neural networks - selecting a network architecture
  - One option is to use a small neural network
    - Few (maybe one) hidden layer and few hidden units
    - Such networks are prone to under fitting
    - But they are computationally cheaper
  - Larger network
    - More hidden layers
      - How do you decide that a larger network is good?
  - Using a single hidden layer is good default
    - Also try with 1, 2, 3, see which performs best on cross validation set
    - So like before, take three sets (training, cross validation)
  - More units
    - This is computational expensive
    - Prone to over-fitting
      - Use regularization to address over fitting

11: Machine Learning System Design

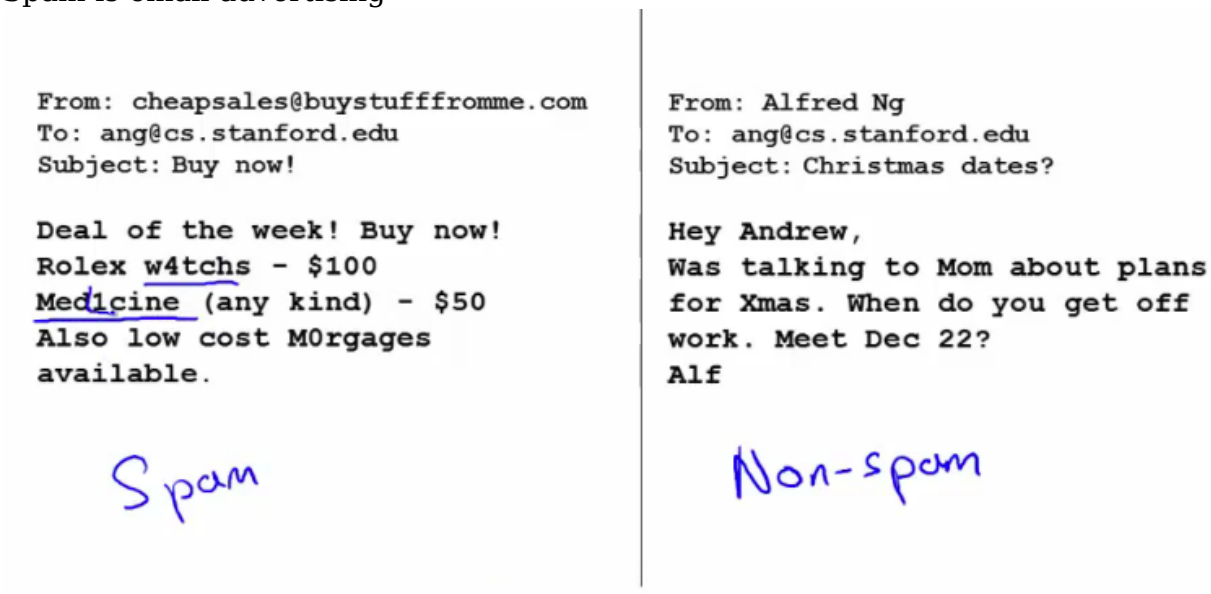
Previous Next Index

Machine learning systems design

- In this section we'll touch on how to put together a system
- Previous sections have looked at a wide range of different issues in significant focus
- This section is less mathematical, but material will be very useful non-the-less
  - Consider the system approach
  - You can understand all the algorithms, but if you don't understand how to make them work in a complete system that's no good!

Prioritizing what to work on - spam classification example

- The idea of prioritizing what to work on is perhaps the most important skill programmers typically need to develop
  - It's so easy to have many ideas you want to work on, and as a result do none of them well, because doing one well is harder than doing six superficially
    - So you need to make sure you complete projects
    - Get something "shipped" - even if it doesn't have all the bells and whistles, that final 20% getting it ready is often the toughest
    - If you only release when you're totally happy you rarely get practice doing that final 20%
  - So, back to machine learning...
- Building a spam classifier
- Spam is email advertising

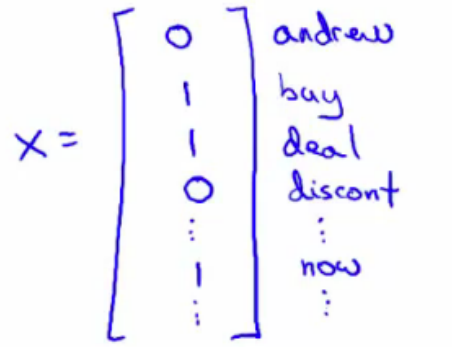


- What kind of features might we define
  - Spam (1)

        - Misspelled word
    - Not spam (0)
        - Real content
- How do we build a classifier to distinguish between the two
    - Feature representation
        - How do represent x (features of the email)?
            - y = spam (1) or not spam (0)

One approach - choosing your own features

- Chose 100 words which are indicative of an email being spam or not spam
    - Spam --> e.g. buy, discount, deal
    - Non spam --> Andrew, now
    - All these words go into one long vector
- Encode this into a **reference vector**
    - See which words appear in a message
- Define a feature vector x
    - Which is 0 or 1 if a word corresponding word in the reference vector is present or not
        - This is a bitmap of the word content of your email
    - i.e. don't recount if a word appears more than once



    - In practice its more common to have a training set and pick the most frequently n words, where n is 10 000 to 50 000
        - So here you're not specifically choosing your own features, but you are choosing *how* you select them from the training set data

What's the best use of your time to improve system accuracy?

- Natural inclination is to collect lots of data
    - Honey pot anti-spam projects try and get fake email addresses into spammers' hands, collect loads of spam
    - This doesn't always help though
- Develop sophisticated features based on email routing information (contained in email header)
    - Spammers often try and obscure origins of email
    - Send through unusual routes

- Develop sophisticated features for message body analysis
  - Discount == discounts?
  - DEAL == deal?
- Develop sophisticated algorithm to detect misspelling
  - Spammers use misspelled word to get around detection systems
- Often a research group **randomly focus on one option**
  - May not be the most fruitful way to spend your time
  - If you brainstorm a set of options this is **really good**
    - Very tempting to just try something

Error analysis

- When faced with a ML problem lots of ideas of how to improve a problem
  - Talk about error analysis - how to better make decisions
- If you're building a machine learning system often good to start by building a simple algorithm which you can implement quickly
  - Spend at most 24 hours developing an initially bootstrapped algorithm
    - Implement and test on cross validation data
  - Plot learning curves to decide if more data, features etc will help algorithmic optimization
    - Hard to tell in advance what is important
    - Learning curves really help with this
    - Way of avoiding **premature optimization**
      - We should let evidence guide decision making regarding development trajectory
  - **Error analysis**
    - Manually examine the samples (in cross validation set) that your algorithm made errors on
    - See if you can work out why
      - Systematic patterns - help design new features to avoid these shortcomings
    - e.g.
      - Built a spam classifier with 500 examples in CV set
        - Here, error rate is high - gets 100 wrong
      - Manually look at 100 and categorize them depending on features
        - e.g. type of email
      - Looking at those email
        - May find **most common type** of spam emails are pharmacy emails, phishing emails
          - See which type is most common - focus your work on those ones
        - What **features would have helped** classify them correctly
          - e.g. deliberate misspelling
          - Unusual email routing
          - Unusual punctuation

            - May fine some "spammer technique" is causing a lot of your misses
                - Guide a way around it
    - Importance of **numerical evaluation**
        - Have a way of numerically evaluated the algorithm
        - If you're developing an algorithm, it's really good to have some performance calculation which gives a single real number to tell you how well its doing
        - e.g.
            - Say were deciding if we should treat a set of similar words as the same word
            - This is done by stemming in NLP (e.g. "Porter stemmer" looks at the etymological stem of a word)
            - This may make your algorithm better or worse
                - Also worth consider weighting error (false positive vs. false negative)
                    - e.g. is a false positive really bad, or is it worth have a few of one to improve performance a lot
            - Can use numerical evaluation to compare the changes
                - See if a change improves an algorithm or not
        - A single real number may be hard/complicated to compute
            - But makes it much easier to evaluate how changes impact your algorithm
    - You should do error analysis on the cross validation set instead of the test set

Error metrics for skewed analysis

- Once case where it's hard to come up with good error metric - skewed classes
- Example
    - Cancer classification
        - Train logistic regression model $h_\theta(x)$ where
            - Cancer means y = 1
            - Otherwise y = 0
        - Test classifier on test set
            - Get 1% error
                - So this looks pretty good..
            - But only 0.5% have cancer
                - Now, 1% error looks very bad!
    - So when one number of examples is very small this is an example of skewed classes
        - LOTS more of one class than another
        - So standard error metrics aren't so good
- Another example
    - Algorithm has 99.2% accuracy
    - Make a change, now get 99.5% accuracy
        - Does this really represent an improvement to the algorithm?

- Did we do something useful, or did we just create something which predicts y = 0 more often
  - Get very low error, but classifier is still not great

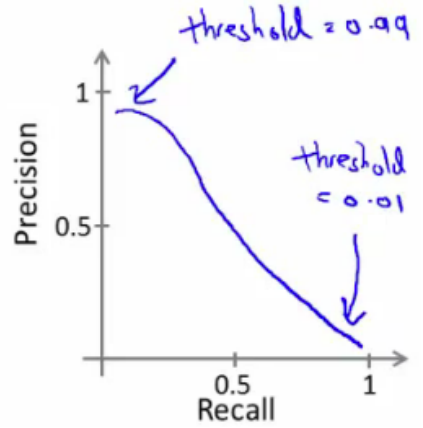
Precision and recall

- Two new metrics - **precision** and **recall**
  - Both give a value between 0 and 1
  - Evaluating classifier on a test set
  - For a test set, the actual class is 1 or 0
  - Algorithm predicts some value for class, predicting a value for each example in the test set
    - Considering this, classification can be
      - True positive (we guessed 1, it was 1)
      - False positive (we guessed 1, it was 0)
      - True negative (we guessed 0, it was 0)
      - False negative (we guessed 0, it was 1)
  - **Precision**
    - *How often does our algorithm cause a false alarm?*
    - Of all patients we predicted have cancer, what fraction of them *actually* have cancer
      - = true positives / # predicted positive
      - = true positives / (true positive + false positive)
    - High precision is good (i.e. closer to 1)
      - You want a big number, because you want false positive to be as close to 0 as possible
  - **Recall**
    - *How sensitive is our algorithm?*
    - Of all patients in set that actually have cancer, what fraction did we correctly detect
      - = true positives / # actual positives
      - = true positive / (true positive + false negative)
    - High recall is good (i.e. closer to 1)
      - You want a big number, because you want false negative to be as close to 0 as possible
  - By computing precision and recall get a better sense of how an algorithm is doing
    - This can't really be gamed
    - Means we're much more sure that an algorithm is good
  - Typically we say the presence of a rare class is what we're trying to determine (e.g. positive (1) is the existence of the rare thing)

Trading off precision and recall

- For many applications we want to control the trade-off between precision and recall
- Example
  - Trained a logistic regression classifier

        - Predict 1 if $h_\theta(x) >= 0.5$
        - Predict 0 if $h_\theta(x) < 0.5$
    - This classifier may give some value for precision and some value for recall
    - Predict 1 only if very confident
        - One way to do this modify the algorithm we could modify the prediction threshold
            - Predict 1 if $h_\theta(x) >= 0.8$
            - Predict 0 if $h_\theta(x) < 0.2$
        - Now we can be more confident a 1 is a true positive
        - But classifier has lower recall - predict y = 1 for a smaller number of patients
            - Risk of false negatives
    - Another example - avoid false negatives
        - This is probably worse for the cancer example
            - Now we may set to a lower threshold
                - Predict 1 if $h_\theta(x) >= 0.3$
                    - Predict 0 if $h_\theta(x) < 0.7$
            - i.e. 30% chance they have cancer
            - So now we have have a higher recall, but lower precision
                - Risk of false positives, because we're less discriminating in deciding what means the person has cancer
- This threshold defines the trade-off
    - We can show this graphically by plotting precision vs. recall



    - This curve can take many different shapes depending on classifier details
    - Is there a way to automatically chose the threshold
        - Or, if we have a few algorithms, how do we compare different algorithms or parameter sets?

	Precision(P)	Recall (R)
Algorithm 1	0.5	0.4
Algorithm 2	0.7	0.1
Algorithm 3	0.02	1.0

    - How do we decide which of these algorithms is best?
        - We spoke previously about using a single real number evaluation metric

        - By switching to precision/recall we have two numbers
        - Now comparison becomes harder
            - Better to have just one number
        - How can we convert P & R into one number?
            - One option is the average - (P + R)/2
                - This is not such a good solution
                    - Means if we have a classifier which predicts y = 1 all the time you get a high recall and low precision
                    - Similarly, if we predict Y rarely get high precision and low recall
                    - So averages here would be 0.45, 0.4 and 0.51
                        - 0.51 is best, despite having a recall of 1 - i.e. predict y=1 for everything
                - So average isn't great
            - **F_1Score** (**fscore**)
                - = 2 * (PR/ [P + R])
                - Fscore is like taking the average of precision and recall giving a higher weight to the lower value
            - Many formulas for computing comparable precision/accuracy values
                - If P = 0 or R = 0 the Fscore = 0
                - If P = 1 and R = 1 then Fscore = 1
                - The remaining values lie between 0 and 1
- Threshold offers a way to control trade-off between precision and recall
- Fscore gives a single real number evaluation metric

    - If you're trying to automatically set the threshold, one way is to try a range of threshold values and evaluate them on your cross validation set
        - Then pick the threshold which gives the best fscore.

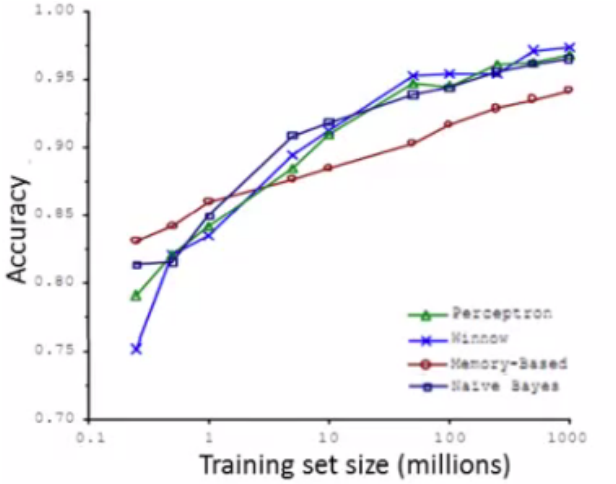
Data for machine learning

- Now switch tracks and look at how much data to train on
- On early videos caution on just blindly getting more data
    - Turns out under certain conditions getting more data is a very effective way to improve performance

Designing a high accuracy learning system

- There have been studies of using different algorithms on data
    - Data - confusing words (e.g. two, to or too)
    - Algorithms
        - Perceptron (logistic regression)
        - Winnow
            - Like logistic regression
            - Used less now
        - Memory based
            - Used less now
            - Talk about this later

        - Naive Bayes
            - Cover later
    - Varied training set size and tried algorithms on a range of sizes



    - What can we conclude
        - Algorithms give remarkably similar performance
        - As training set sizes increases accuracy increases
        - Take an algorithm, give it more data, should beat a "better" one with less data
        - Shows that
            - Algorithm choice is pretty similar
            - More data helps

- When is this true and when is it not?
    - If we can correctly assume that features x have enough information to predict y accurately, then more data will probably help
        - A useful test to determine if this is true can be, "given x, can a human expert predict y?"
    - So lets say we use a learning algorithm with many parameters such as logistic regression or linear regression with many features, or neural networks with many hidden features
        - These are powerful learning algorithms with many parameters which can fit complex functions
            - Such algorithms are low bias algorithms
                - Little systemic bias in their description - flexible
        - Use a small training set
            - Training error should be small
        - Use a very large training set
            - If the training set error is close to the test set error
            - Unlikely to over fit with our complex algorithms
            - So the test set error should also be small
    - Another way to think about this is we want our algorithm to have low bias and low variance
        - Low bias --> use complex algorithm
        - Low variance --> use large training set

12: Support Vector Machines (SVMs)

Previous Next Index

Support Vector Machine (SVM) - Optimization objective

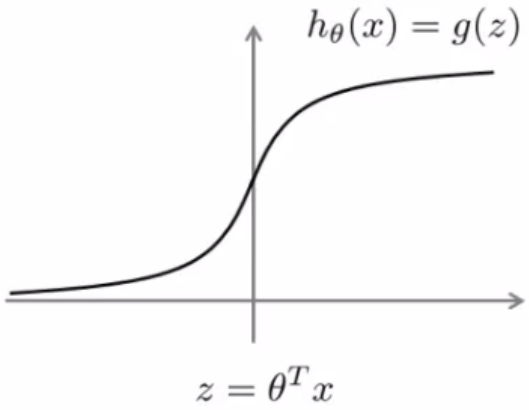
- So far, we've seen a range of different algorithms
  - With supervised learning algorithms - performance is pretty similar
    - What matters more often is;
      - The amount of training data
      - Skill of applying algorithms
- One final supervised learning algorithm that is widely used - **support vector machine (SVM)**
  - Compared to both logistic regression and neural networks, a SVM sometimes gives a cleaner way of learning non-linear functions
  - Later in the course we'll do a survey of different supervised learning algorithms

An alternative view of logistic regression

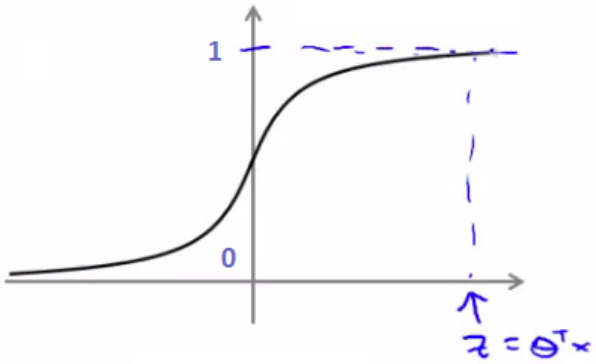
- Start with logistic regression, see how we can modify it to get the SVM
  - As before, the logistic regression hypothesis is as follows

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

  - And the sigmoid activation function looks like this



  - In order to explain the math, we use z as defined above
- What do we want logistic regression to do?
  - We have an example where y = 1
    - Then we hope $h_\theta(x)$ is close to 1
    - With $h_\theta(x)$ close to 1, ($\theta^T x$) must be **much larger** than 0



  - Similarly, when y = 0
    - Then we hope $h_\theta(x)$ is close to 0
    - With $h_\theta(x)$ close to 0, ($\theta^T x$) must be **much less** than 0
  - This is our classic view of logistic regression
    - Let's consider another way of thinking about the problem
- Alternative view of logistic regression
  - If you look at cost function, each example contributes a term like the one below to the overall cost function

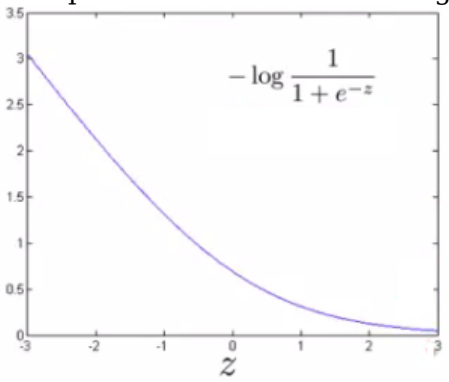
$$-(y \log h_\theta(x) + (1-y) \log(1 - h_\theta(x)))$$

    - For the overall cost function, we sum over all the training examples using the above function, and have a 1/m term
- If you then plug in the hypothesis definition ($h_\theta(x)$), you get an expanded cost function equation;

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

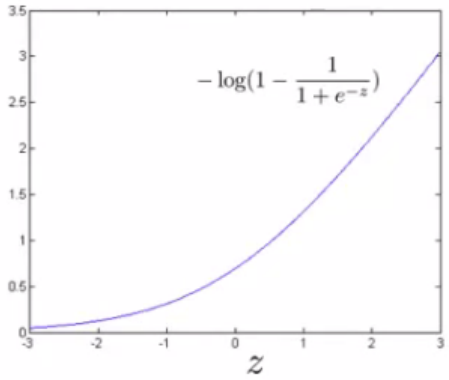
  - So each training example contributes that term to the cost function for logistic regression

- If y = 1 then only the first term in the objective matters
  - If we plot the functions vs. z we get the following graph



    - This plot shows the cost contribution of an example when y = 1 given z
      - So if z is big, the cost is low - this is good!
      - But if z is 0 or negative the cost contribution is high
      - This is why, when logistic regression sees a positive example, it tries to set $\theta^T x$ to be a very large term

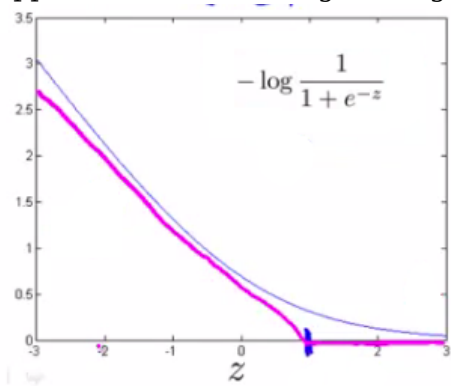
- If y = 0 then only the second term matters
  - We can again plot it and get a similar graph



    - Same deal, if z is small then the cost is low
      - But if s is large then the cost is massive

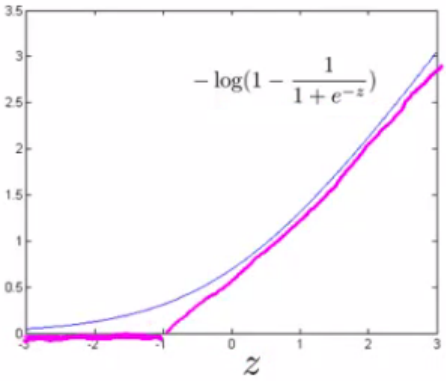
SVM cost functions from logistic regression cost functions

- To build a SVM we must redefine our cost functions
  - When y = 1
    - Take the y = 1 function and create a new cost function
    - Instead of a curved line create two straight lines (magenta) which acts as an approximation to the logistic regression y = 1 function



      - Take point (1) on the z axis
        - Flat from 1 onwards
        - Grows when we reach 1 or a lower number
      - This means we have two straight lines
        - Flat when cost is 0
        - Straight growing line after 1
    - So this is the new y=1 cost function
      - Gives the SVM a computational advantage and an easier optimization problem
      - We call this function **$\text{cost}_1(z)$**

- Similarly
  - When y = 0
    - Do the equivalent with the y=0 function plot



- We call this function **$\text{cost}_0(z)$**

- So here we define the two cost function terms for our SVM graphically
  - How do we implement this?

The complete SVM cost function

- As a comparison/reminder we have logistic regression below

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

  - If this looks unfamiliar its because we previously had the - sign outside the expression
- For the SVM we take our two logistic regression y=1 and y=0 terms described previously and replace with
  - $\text{cost}_1(\theta^T x)$
  - $\text{cost}_0(\theta^T x)$
- So we get

$$\min_\theta \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1-y^{(i)}) cost_0(\theta^T x^{(i)})\right] + \frac{\lambda}{2m}\sum_{i=1}^{n}\theta_j^2$$

SVM notation is slightly different

- In convention with SVM notation we rename a few things here
- 1) Get rid of the 1/m terms
  - This is just a slightly different convention
  - By removing 1/m we should get the same optimal values for
    - 1/m is a constant, so should get same optimization
    - e.g. say you have a minimization problem which minimizes to u = 5
      - If your cost function * by a constant, you still generates the minimal value
      - That minimal value is different, but that's irrelevant
- 2) For logistic regression we had two terms;
  - Training data set term (i.e. that we sum over m) = **A**
  - Regularization term (i.e. that we sum over n) = **B**
    - So we could describe it as A + λB
    - Need some way to deal with the trade-off between regularization and data set terms
    - Set different values for λ to parametrize this trade-off
  - Instead of parameterization this as A + λB

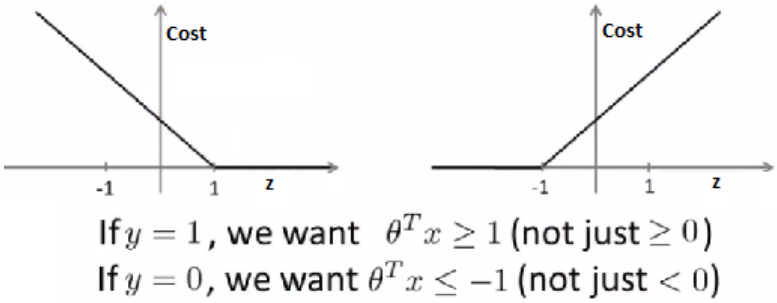
        - For SVMs the convention is to use a different parameter called C
        - So do CA + B
        - If C were equal to 1/λ then the two functions (CA + B and A + λB) would give the same value
- So, our overall equation is

$$\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}cost_1(\theta^T x^{(i)}) + (1-y^{(i)})cost_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

- Unlike logistic, $h_\theta(x)$ doesn't give us a probability, but instead we get a direct prediction of 1 or 0
    - So if $\theta^T x$ is equal to or greater than 0 --> $h_\theta(x) = 1$
    - Else --> $h_\theta(x) = 0$

Large margin intuition

- Sometimes people refer to SVM as **large margin classifiers**
    - We'll consider what that means and what an SVM hypothesis looks like
    - The SVM cost function is as above, and we've drawn out the cost terms below



    - Left is $cost_1$ and right is $cost_0$
    - What does it take to make terms small
        - If y =1
            - $cost_1(z) = 0$ only when z >= 1
        - If y = 0
            - $cost_0(z) = 0$ only when z <= -1
    - Interesting property of SVM
        - If you have a positive example, you only really *need* z to be greater or equal to 0
            - If this is the case then you predict 1
        - SVM wants a bit more than that - doesn't want to *just* get it right, but have the value be quite a bit bigger than zero
            - Throws in an extra safety margin factor
- Logistic regression does something similar
- What are the consequences of this?
    - Consider a case where we set C to be huge
        - C = 100,000
        - So considering we're minimizing CA + B
            - If C is huge we're going to pick an A value so that A is equal to zero
            - What is the optimization problem here - how do we make A = 0?
        - Making A = 0
            - If y = 1

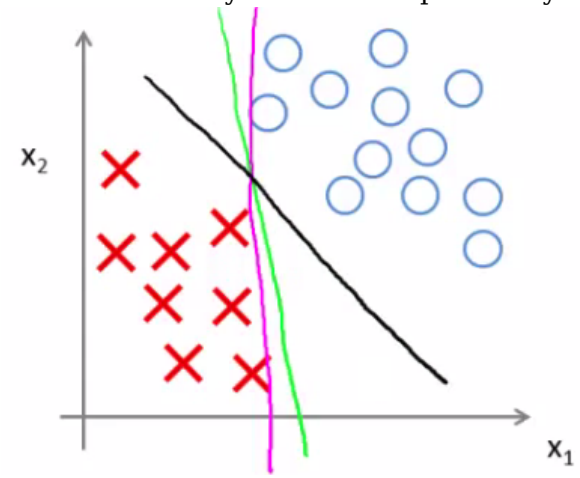
        - Then to make our "A" term 0 need to find a value of θ so (θ^T x) is greater than or equal to 1
    - Similarly, if y = 0
        - Then we want to make "A" = 0 then we need to find a value of θ so (θ^T x) is equal to or less than -1
- So - if we think of our optimization problem a way to ensure that this first "A" term is equal to 0, we re-factor our optimization problem into just minimizing the "B" (regularization) term, because
    - When A = 0 --> A*C = 0
- So we're minimizing B, under the constraints shown below

$$\min \frac{1}{2} \sum_{i=1}^{n} \theta_j^2$$

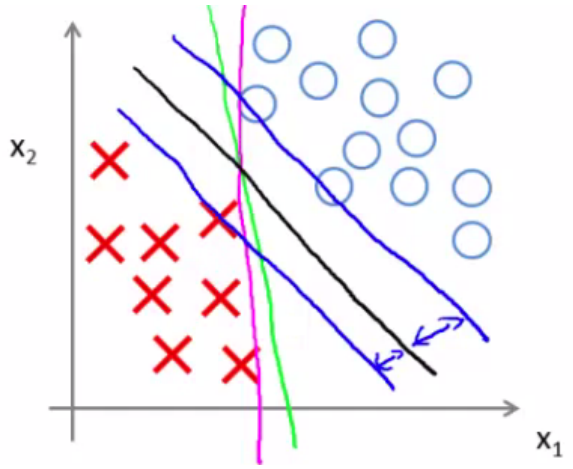
$$\text{s.t.} \quad \theta^T x^{(i)} \geq 1 \quad \text{if} \quad y^{(i)} = 1$$

$$\theta^T x^{(i)} \leq -1 \quad \text{if} \quad y^{(i)} = 0$$

- Turns out when you solve this problem you get interesting decision boundaries

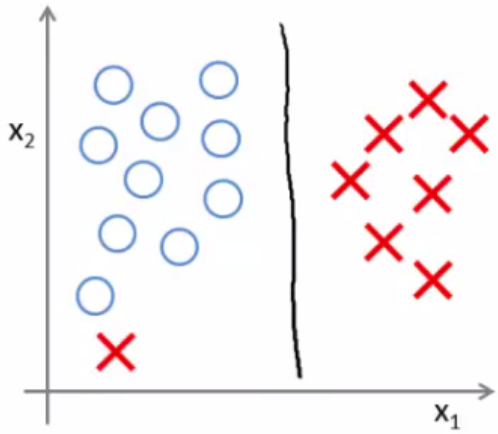


- The green and magenta lines are functional decision boundaries which could be chosen by logistic regression
    - But they probably don't generalize too well
- The black line, by contrast is the the chosen by the SVM because of this safety net imposed by the optimization graph
    - More robust separator
- Mathematically, that black line has a larger minimum distance (margin) from any of the training examples

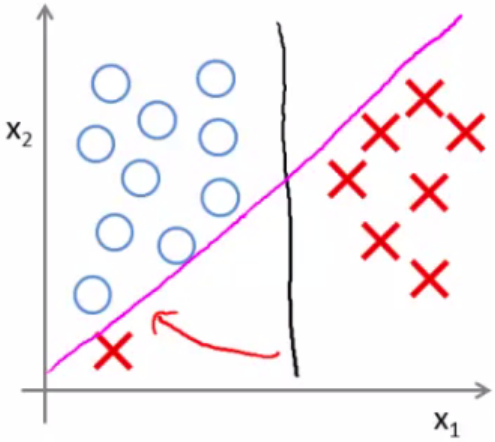


- By separating with the largest margin you incorporate robustness into your decision making process

- We looked at this at when C is very large
  - SVM is more sophisticated than the large margin might look
    - If you were just using large margin then SVM would be very sensitive to outliers



    - You would risk making a ridiculous hugely impact your classification boundary
      - A single example might not represent a good reason to change an algorithm
      - If C is very large then we *do* use this quite naive maximize the margin approach



      - So we'd change the black to the magenta

        - But if C is reasonably small, or a not too large, then you stick with the black decision boundary
    - What about non-linearly separable data?
        - Then SVM still does the right thing if you use a normal size C
        - So the idea of SVM being a large margin classifier is only really relevant when you have no outliers and you can easily linearly separable data
    - Means we ignore a few outliers

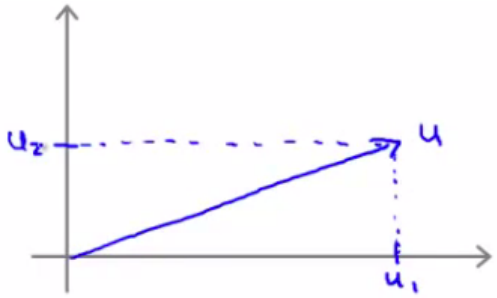
Large margin classification mathematics (optional)

Vector inner products

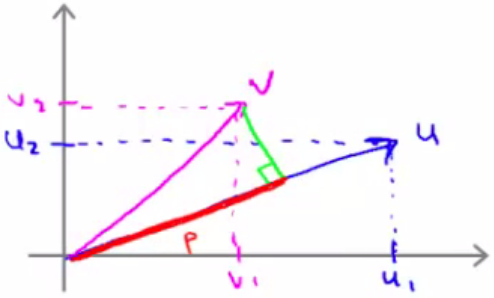
- Have two (2D) vectors u and v - what is the inner product ($u^T v$)?

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

    - Plot u on graph
        - i.e u_1 vs. u_2

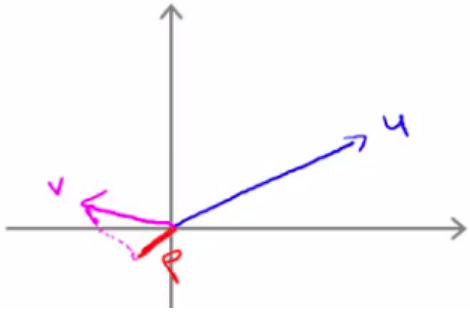


    - One property which is good to have is the **norm** of a vector
        - Written as ||u||
            - This is the euclidean length of vector u
        - So ||u|| = SQRT($u_1^2 + u_2^2$) = real number
            - i.e. length of the arrow above
            - Can show via Pythagoras
    - For the inner product, take v and orthogonally project down onto u
        - First we can plot v on the same axis in the same way (v_1 vs v_1)
        - Measure the length/magnitude of the projection



        - So here, the green line is the projection
            - p = length along u to the intersection
            - p is the magnitude of the projection of vector v onto vector u
    - Possible to show that

        - $u^T v$ = p * ||u||
            - So this is one way to compute the inner product
        - $u^T v = u_1v_1 + u_2v_2$
        - So therefore
            - **p * ||u|| = $u_1v_1 + u_2v_2$**
            - This is an important rule in linear algebra
        - We can reverse this too
            - So we could do
                - $v^T u = v_1u_1 + v_2u_2$
                - Which would obviously give you the same number
    - p can be negative if the angle between them is 90 degrees or more



        - So here p is negative
- Use the vector inner product theory to try and understand SVMs a little better

SVM decision boundary

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

$$\text{s.t.} \quad \theta^T x^{(i)} \geq 1 \qquad \text{if } y^{(i)} = 1$$

$$\theta^T x^{(i)} \leq -1 \qquad \text{if } y^{(i)} = 0$$

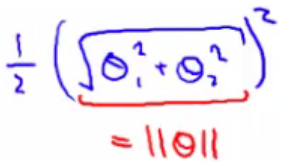
- For the following explanation - two simplification
    - Set $\theta_0 = 0$ (i.e. ignore intercept terms)
    - Set n = 2 - (x_1, x_2)
        - i.e. each example has only 2 features
- Given we only have two parameters we can simplify our function to

$$\frac{1}{2}(\theta_1^2 + \theta_2^2)$$

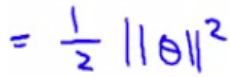
- And, can be re-written as

$$\frac{1}{2}\left(\sqrt{\theta_1^2 + \theta_2^2}\right)^2$$

    - Should give same thing
- We may notice that

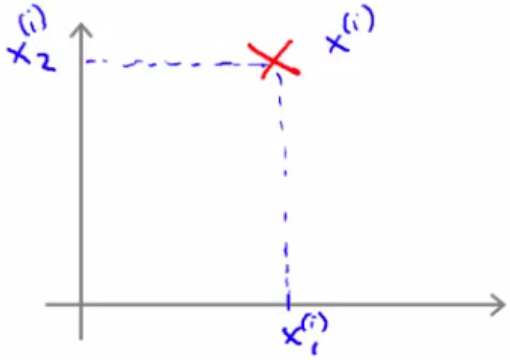


- The term in red is the norm of θ
  - If we take θ as a 2x1 vector
  - If we assume $\theta_0 = 0$ its still true
- So, finally, this means our optimization function can be re-defined as

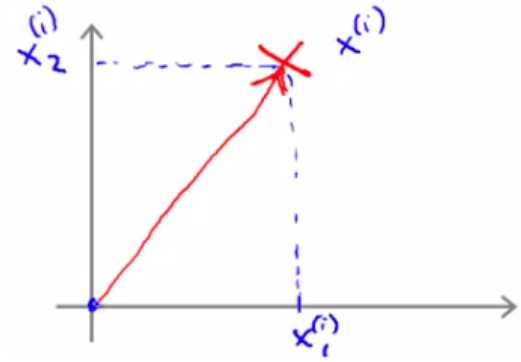


- So the SVM is minimizing the squared norm

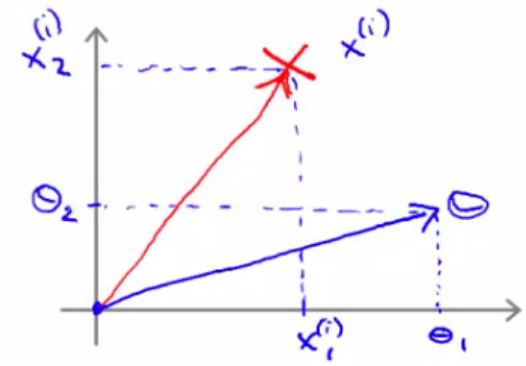
- Given this, what are the ($\theta^T x$) parameters doing?
  - Given θ and given example x what is this equal to
    - We can look at this in a comparable manner to how we just looked at u and v
  - Say we have a single positive training example (red cross below)



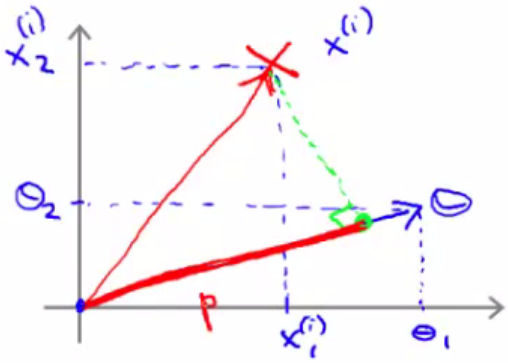
  - Although we haven't been thinking about examples as vectors it can be described as such



  - Now, say we have our parameter vector θ and we plot that on the same axis



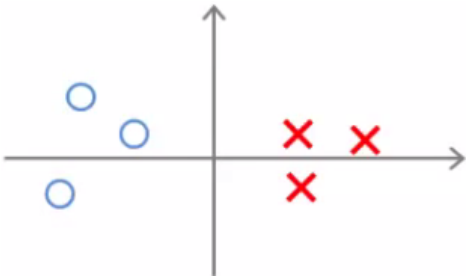
- The next question is what is the inner product of these two vectors



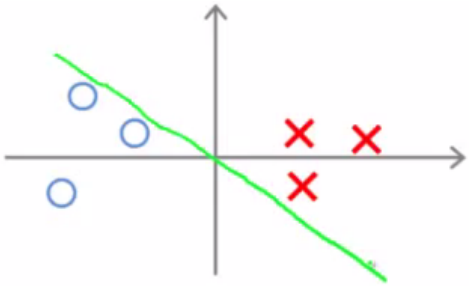
  - p, is in fact p^i, because it's the length of p for example i
    - Given our previous discussion we know
      $(\theta^T x^i) = p^i * ||\theta||$
      $= \theta_1 x^i_1 + \theta_2 x^i_2$
    - So these are both equally valid ways of computing $\theta^T x^i$
- What does this mean?
  - The constraints we defined earlier
    - $(\theta^T x) >= 1$ if y = 1
    - $(\theta^T x) <= -1$ if y = 0
  - Can be replaced/substituted with the constraints
    - $p^i * ||\theta|| >= 1$ if y = 1
    - $p^i * ||\theta|| <= -1$ if y = 0
  - Writing that into our optimization objective

$$\min_\theta \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\|\theta\|^2$$
$$\text{s.t.} \quad p^{(i)} \cdot \|\theta\| \geq 1 \quad \text{if } y^{(i)} = 1$$
$$p^{(i)} \cdot \|\theta\| \leq -1 \quad \text{if } y^{(i)} = 1$$

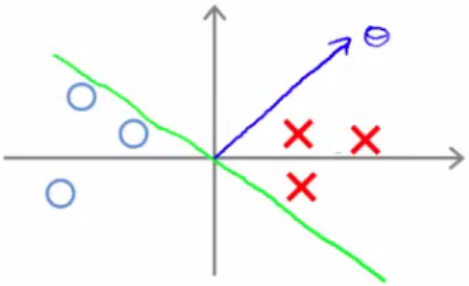
- So, given we've redefined these functions let us now consider the training example below



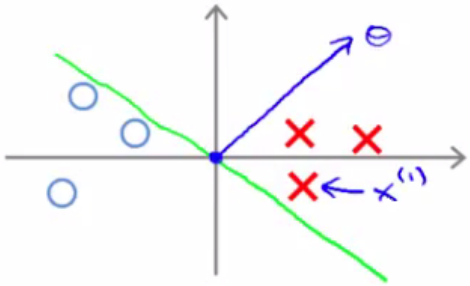
- Given this data, what boundary will the SVM choose? Note that we're still assuming $\theta_0 = 0$, which means the boundary has to pass through the origin (0,0)
  - Green line - small margins



    - SVM would not chose this line
      - Decision boundary comes very close to examples
      - Lets discuss *why* the SVM would **not** chose this decision boundary
  - Looking at this line
    - We can show that θ is at 90 degrees to the decision boundary

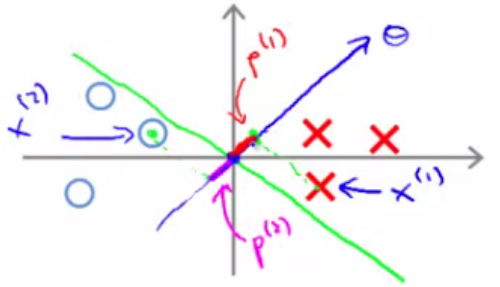


    - **θ is always at 90 degrees to the decision boundary** (can show with linear algebra, although we're not going to!)
- So now lets look at what this implies for the optimization objective
  - Look at first example (x^1)

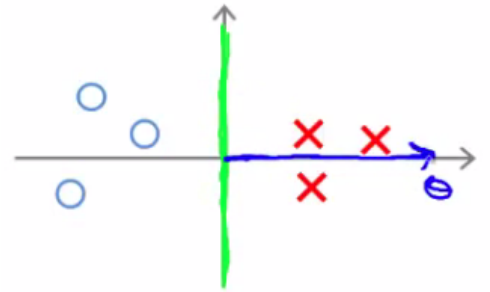


  - Project a line from x^1 on to to the θ vector (so it hits at 90 degrees)
    - The distance between the intersection and the origin is ($\mathbf{p^1}$)
  - Similarly, look at second example (x^2)

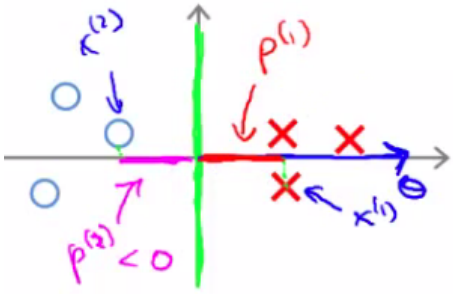
        - Project a line from x^2 into to the θ vector
        - This is the magenta line, which will be **negative** (p^2)
    - If we overview these two lines below we see a graphical representation of what's going on;



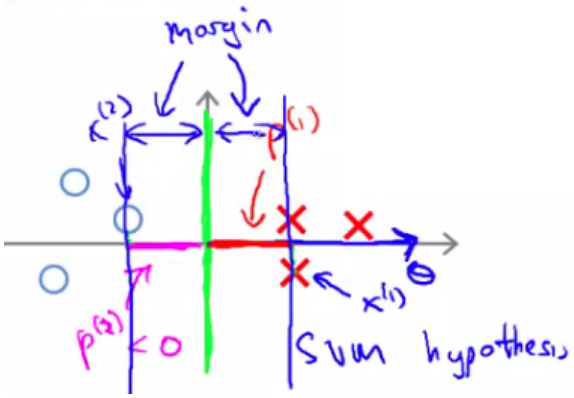
    - We find that both these p values are going to be pretty small
    - If we look back at our optimization objective
        - We know we need $p^1 * ||\theta||$ to be bigger than or equal to 1 for positive examples
            - If p is small
                - Means that ||θ|| must be pretty large
        - Similarly, for negative examples we need $p^2 * ||\theta||$ to be smaller than or equal to -1
            - We saw in this example p^2 is a small negative number
                - So ||θ|| must be a large number
    - Why is this a problem?
        - The optimization objective is trying to find a set of parameters where the norm of theta is small
            - So this doesn't seem like a good direction for the parameter vector (because as p values get smaller ||θ|| must get larger to compensate)
                - So we should make p values larger which allows ||θ|| to become smaller
- So lets chose a different boundary



    - Now if you look at the projection of the examples to θ we find that p^1 becomes large and ||θ|| can become small
    - So with some values drawn in



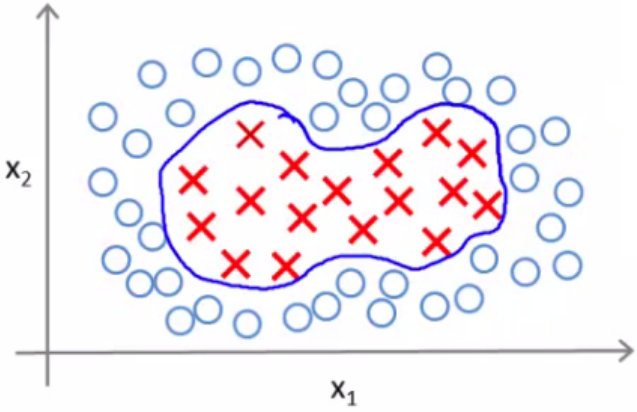
- This means that by choosing this second decision boundary we can make ||θ|| smaller
  - Which is why the SVM choses this hypothesis as better
  - This is how we generate the large margin effect



  - The magnitude of this margin is a function of the p values
    - So by maximizing these p values we minimize ||θ||
- Finally, we did this derivation assuming $\theta_0 = 0$,
  - If this is the case we're entertaining only decision boundaries which pass through (0,0)
  - If you allow θ_0 to be other values then this simply means you can have decision boundaries which cross through the x and y values at points other than (0,0)
  - Can show with basically same logic that this works, and even when θ_0 is non-zero when you have optimization objective described above (when C is very large) that the SVM is looking for a large margin separator between the classes

Kernels - 1: Adapting SVM to non-linear classifiers

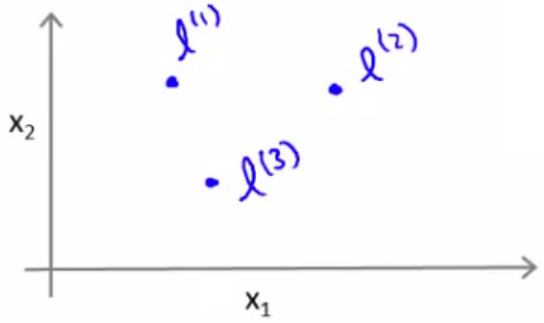
- What are kernels and how do we use them
  - We have a training set
  - We want to find a non-linear boundary



  - Come up with a complex set of polynomial features to fit the data
    - Have $h_\theta(x)$ which
      - Returns 1 if the combined weighted sum of vectors (weighted by the parameter vector) is less than or equal to 0
      - Else return 0
    - Another way of writing this (new notation) is
      - That a hypothesis computes a decision boundary by taking the sum of the parameter vector multiplied by a **new feature vector f**, which simply

contains the various high order x terms
- e.g.
  - $h_\theta(x) = \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3$
  - Where
    - $f_1 = x_1$
    - $f_2 = x_1 x_2$
    - $f_3 = ...$
    - i.e. not specific values, but each of the terms from your complex polynomial function
- Is there a better choice of feature f than the high order polynomials?
  - As we saw with computer imaging, high order polynomials become computationally expensive

- New features
  - Define three features in this example (ignore x_0)
  - Have a graph of x_1 vs. x_2 (don't plot the values, just define the space)
  - Pick three points in that space



  - These points l^1, l^2, and l^3, were chosen manually and are called **landmarks**
    - Given x, define f1 as the similarity between (x, l^1)
      - = exp(- (|| x - l^1 ||^2) / $2\sigma^2$)
      - = $\exp\left(-\frac{\|x-l^{(1)}\|^2}{2\sigma^2}\right)$
      - **|| x - l^1 ||** is the euclidean distance between the point x and the landmark l^1 squared
        - Disussed more later
      - If we remember our statistics, we know that
        - σ is the **standard deviation**
        - σ^2 is commonly called the **variance**
    - Remember, that as discussed

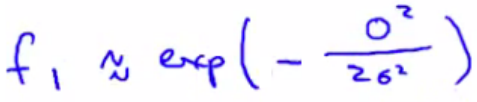
$$\|x-l^{(1)}\|^2 = \sum_{j=1}^{n}(x_j - l_j^{(1)})^2$$

  - So, f2 is defined as
    - f2 = similarity(x, l^1) = exp(- (|| x - l^2 ||^2) / $2\sigma^2$)
  - And similarly
    - f3 = similarity(x, l^2) = exp(- (|| x - l^1 ||^2) / $2\sigma^2$)
  - This similarity function is called a **kernel**
    - This function is a **Gaussian Kernel**

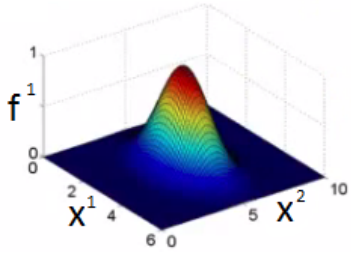
- So, instead of writing similarity between x and l we might write
  - $f1 = k(x, l^1)$

Diving deeper into the kernel

- So lets see what these kernels do and why the functions defined make sense
  - Say x is close to a landmark
    - Then the squared distance will be ~0
      - So

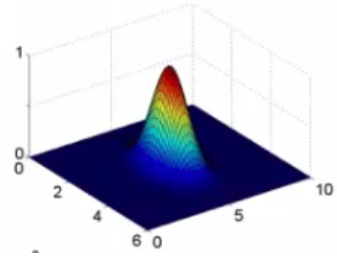


        - Which is basically e^{-0}
          - Which is close to 1
    - Say x is far from a landmark
      - Then the squared distance is big
        - Gives $e^{-\text{large number}}$
          - Which is close to zero
    - Each landmark defines a new features
- If we plot f1 vs the kernel function we get a plot like this
  - Notice that when x = [3,5] then f1 = 1
  - As x moves away from [3,5] then the feature takes on values close to zero
  - So this measures how close x is to this landmark

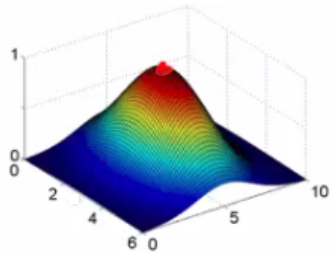


What does σ do?

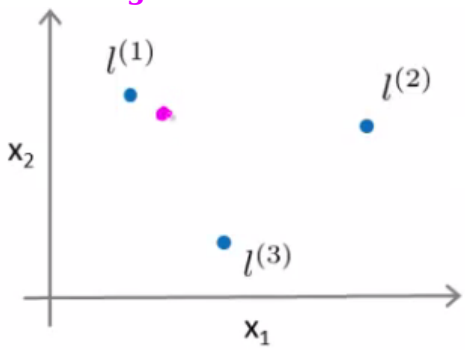
- **σ^2** is a parameter of the Gaussian kernel
  - Defines the steepness of the rise around the landmark
- Above example $\sigma^2 = 1$
- Below $\sigma^2 = 0.5$



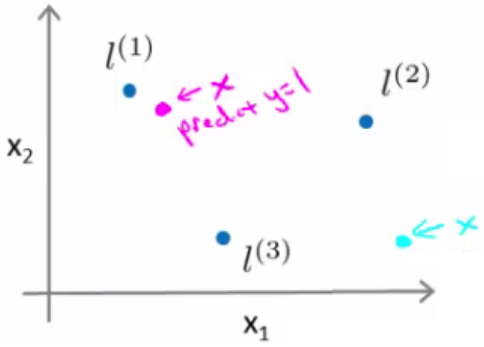
  - We see here that as you move away from 3,5 the feature f1 falls to zero much more rapidly
- The inverse can be seen if $\sigma^2 = 3$



- Given this definition, what kinds of hypotheses can we learn?
  - With training examples x we predict "1" when
  - $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 >= 0$
    - For our example, lets say we've already run an algorithm and got the
      - $\theta_0 = -0.5$
      - $\theta_1 = 1$
      - $\theta_2 = 1$
      - $\theta_3 = 0$
    - Given our placement of three examples, what happens if we evaluate an example at the **magenta dot** below?

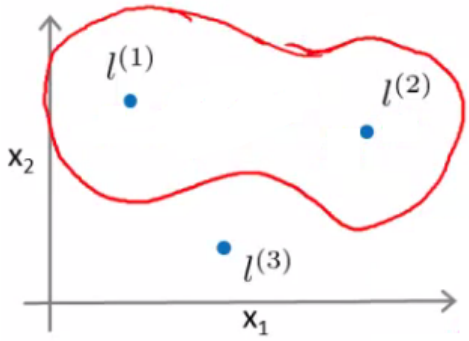


    - Looking at our formula, we know f1 will be close to 1, but f2 and f3 will be close to 0
      - So if we look at the formula we have
        - $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 >= 0$
        - -0.5 + 1 + 0 + 0 = 0.5
          - 0.5 is greater than 1
    - If we had **another point** far away from all three



      - This equates to -0.5
        - So we predict 0

- Considering our parameter, for points near l^1 and l^2 you predict 1, but for points near l^3 you predict 0
- Which means we create a non-linear decision boundary that goes a lil' something like this;



  - Inside we predict y = 1
  - Outside we predict y = 0
- So this show how we can create a non-linear boundary with landmarks and the kernel function in the support vector machine
  - But
    - How do we get/chose the landmarks
    - What other kernels can we use (other than the Gaussian kernel)

Kernels II

- Filling in missing detail and practical implications regarding kernels
- Spoke about picking landmarks manually, defining the kernel, and building a hypothesis function
  - Where do we get the landmarks from?
  - For complex problems we probably want lots of them

Choosing the landmarks

- Take the training data
- For each example place a landmark at exactly the same location
- So end up with m landmarks
  - One landmark per location per training example
  - Means our features measure how close to a training set example something is
- Given a new example, compute all the f values
  - Gives you a feature vector f (f_0 to f_m)
    - $f_0 = 1$ always
- A more detailed look at generating the f vector
  - If we had a training example - features we compute would be using (x^i, y^i)
    - So we just cycle through each landmark, calculating how close to that landmark actually x^i is
      - f_1^i, = $k(x^i, l^1)$
      - f_2^i, = $k(x^i, l^2)$
      - ...
      - f_m^i, = $k(x^i, l^m)$

        - Somewhere in the list we compare x to itself... (i.e. when we're at f_i^i)
            - So because we're using the Gaussian Kernel this evalues to 1
        - Take these m features (f_1, f_2 ... f_m) group them into an [m +1 x 1] dimensional vector called f
            - f^i is the f feature vector for the ith example
            - And add a 0th term = 1
- Given these kernels, how do we use a support vector machine

SVM hypothesis prediction with kernels

- Predict y = 1 if (θ^T f) >= 0
    - Because θ = [m+1 x 1]
    - And f = [m +1 x 1]
- So, this is how you make a prediction assuming you already have θ
    - How do you get θ?

SVM training with kernels

- Use the SVM learning algorithm

$$\min_\theta C\sum_{i=1}^{m} y^{(i)} cost_1(\theta^T f^{(i)}) + (1-y^{(i)}) cost_0(\theta^T f^{(i)}) + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

    - Now, we minimize using f as the feature vector instead of x
    - By solving this minimization problem you get the parameters for your SVM
- In this setup, m = n
    - Because number of features is the number of training data examples we have
- One final mathematic detail (not crucial to understand)
    - If we ignore θ_0 then the following is true

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

    - What many implementations do is

$$\theta^T \mathbf{M} \theta$$

        - Where the matrix M depends on the kernel you use
        - Gives a slightly different minimization - means we determine a rescaled version of θ
        - Allows more efficient computation, and scale to much bigger training sets
        - If you have a training set with 10 000 values, means you get 10 000 features
            - Solving for all these parameters can become expensive
            - So by adding this in we avoid a for loop and use a matrix multiplication algorithm instead
- You can apply kernels to other algorithms
    - But they tend to be very computationally expensive
    - But the SVM is far more efficient - so more practical
- Lots of good off the shelf software to minimize this function

- **SVM parameters (C)**

    - Bias and variance trade off
    - Must chose C
        - C plays a role similar to 1/LAMBDA (where LAMBDA is the regularization parameter)
    - Large C gives a hypothesis of **low bias high variance** --> overfitting
    - Small C gives a hypothesis of **high bias low variance** --> underfitting
- **SVM parameters (σ^2)**
    - Parameter for calculating f values
        - Large σ^2 - f features vary more smoothly - higher bias, lower variance
        - Small σ^2 - f features vary abruptly - low bias, high variance

SVM - implementation and use

- So far spoken about SVM in a very abstract manner
- What do you need to do this
    - Use SVM software packages (e.g. liblinear, libsvm) to solve parameters θ
    - Need to specify
        - Choice of parameter C
        - Choice of kernel

Choosing a kernel

- We've looked at the **Gaussian kernel**
    - Need to define σ (σ^2)
        - Discussed σ^2
    - When would you chose a Gaussian?
        - If n is small and/or m is large
            - e.g. 2D training set that's large
    - If you're using a Gaussian kernel then you may need to implement the kernel function
        - e.g. a function
          fi = kernel(x1,x2)
            - Returns a real number
        - Some SVM packages will expect you to define kernel
        - Although, some SVM implementations include the Gaussian and a few others
            - Gaussian is probably most popular kernel
    - NB - make sure you perform **feature scaling** before using a Gaussian kernel
        - If you don't features with a large value will dominate the f value
- Could use no kernel - **linear kernel**
    - Predict y = 1 if (θ^T x) >= 0
        - So no f vector
        - Get a standard linear classifier
    - Why do this?
        - If n is large and m is small then
            - Lots of features, few examples
            - Not enough data - risk overfitting in a high dimensional feature-space
- Other choice of kernel
    - Linear and Gaussian are most common
    - Not all similarity functions you develop are valid kernels
        - Must satisfy **Merecer's Theorem**
        - SVM use numerical optimization tricks

        - Mean certain optimizations can be made, but they must follow the theorem
    - **Polynomial Kernel**
        - We measure the similarity of x and l by doing one of
            - $(x^T l)^2$
            - $(x^T l)^3$
            - $(x^T l+1)^3$
        - General form is
            - $(x^T l+Con)^D$
        - If they're similar then the inner product tends to be large
        - Not used that often
        - Two parameters
            - Degree of polynomial (D)
            - Number you add to l (Con)
        - Usually performs worse than the Gaussian kernel
        - Used when x and l are both non-negative
    - **String kernel**
        - Used if input is text strings
        - Use for text classification
    - **Chi-squared kernel**
    - **Histogram intersection kernel**

Multi-class classification for SVM

- Many packages have built in multi-class classification packages
- Otherwise use one-vs all method
- Not a big issue

Logistic regression vs. SVM

- When should you use SVM and when is logistic regression more applicable
- If n (features) is large vs. m (training set)
    - e.g. text classification problem
        - Feature vector dimension is 10 000
        - Training set is 10 - 1000
        - Then use logistic regression or SVM with a linear kernel
- If n is small and m is intermediate
    - n = 1 - 1000
    - m = 10 - 10 000
    - Gaussian kernel is good
- If n is small and m is large
    - n = 1 - 1000
    - m = 50 000+
        - SVM will be slow to run with Gaussian kernel
    - In that case
        - Manually create or add more features
        - Use logistic regression of SVM with a linear kernel
- Logistic regression and SVM with a linear kernel are pretty similar
    - Do similar things
    - Get similar performance

- A lot of SVM's power is using diferent kernels to learn complex non-linear functions
- For all these regimes a well designed NN should work
  - But, for some of these problems a NN might be slower - SVM well implemented would be faster
- SVM has a convex optimization problem - so you get a global minimum
- It's not always clear how to chose an algorithm
  - Often more important to get enough data
  - Designing new features
  - Debugging the algorithm
- SVM is widely perceived a very powerful learning algorithm

13: Clustering

Previous Next Index

Unsupervised learning - introduction

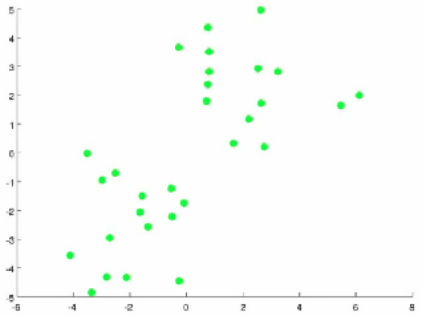
- Talk about **clustering**
  - **Learning from unlabeled data**
- Unsupervised learning
  - Useful to contras with supervised learning
- Compare and contrast
  - Supervised learning
    - Given a set of labels, fit a hypothesis to it
  - Unsupervised learning
    - Try and determining structure in the data
    - Clustering algorithm groups data together based on data features
- What is clustering good for
  - **Market segmentation** - group customers into different market segments
  - **Social network analysis** - Facebook "smartlists"
  - **Organizing computer clusters** and data centers for network layout and location
  - **Astronomical data analysis** - Understanding galaxy formation

K-means algorithm

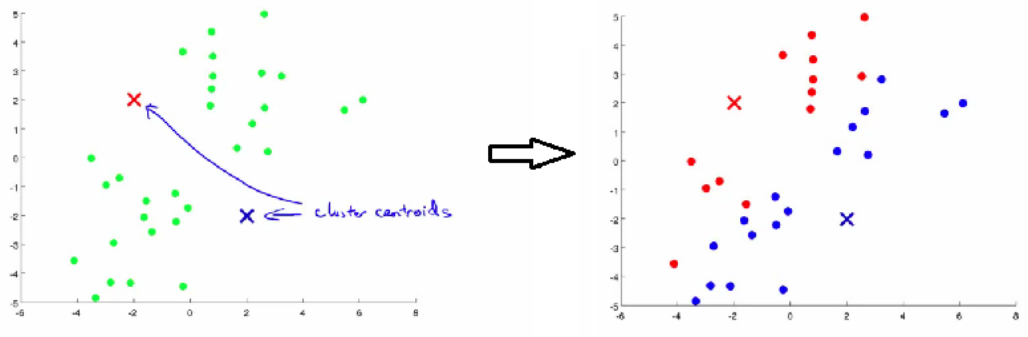
- Want an algorithm to automatically group the data into coherent clusters
- K-means is **by far** the most widely used clustering algorithm

Overview

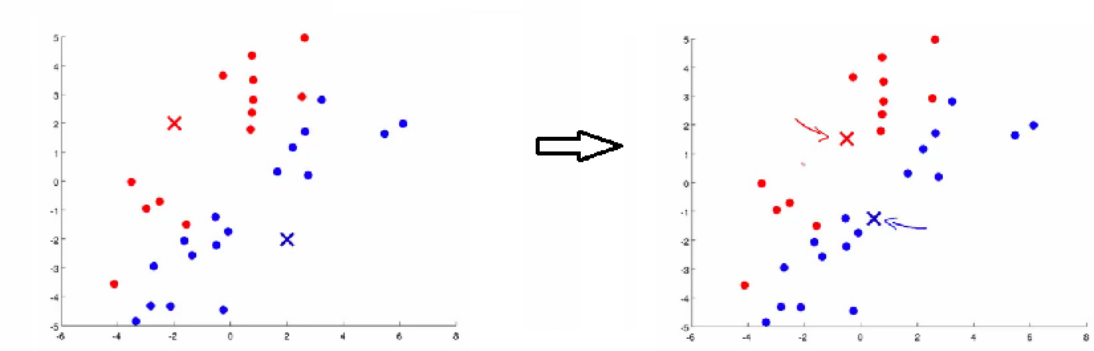
- Take unlabeled data and group into two clusters



- Algorithm overview
  - 1) Randomly allocate two points as the **cluster centroids**
    - Have as many cluster centroids as clusters you want to do (*K* cluster centroids, in fact)
    - In our example we just have two clusters
  - 2) Cluster assignment step
    - Go through each example and depending on if it's closer to the red or blue centroid assign each point to one of the two clusters
    - To demonstrate this, we've gone through the data and "colour" each point red or blue



- 3) Move centroid step
  - Take each centroid and move to the average of the correspondingly assigned data-points



  - Repeat 2) and 3) until convergence
- More formal definition
  - **Input:**
    - K (number of clusters in the data)
    - Training set $\{x^1, x^2, x^3 \ldots, x^n)$
  - **Algorithm:**
    - Randomly initialize K cluster centroids as $\{\mu_1, \mu_2, \mu_3 \ldots \mu_K\}$

```
Repeat {
        for i = 1 to m
            c(i) := index (from 1 to K) of cluster centroid
                    closest to x(i)
        for k = 1 to K
            μk := average (mean) of points assigned to cluster k }
```

      - Loop 1
        - This inner loop repeatedly sets the $c^{(i)}$ variable to be the index of the closes variable of cluster centroid closes to x^i
        - i.e. take i^{th} example, measure squared distance to each cluster centroid, assign $c^{(i)}$to the cluster closest

$$\min_k \|x^{(i)} - \mu_k\|^2 \rightarrow c^{(i)}$$

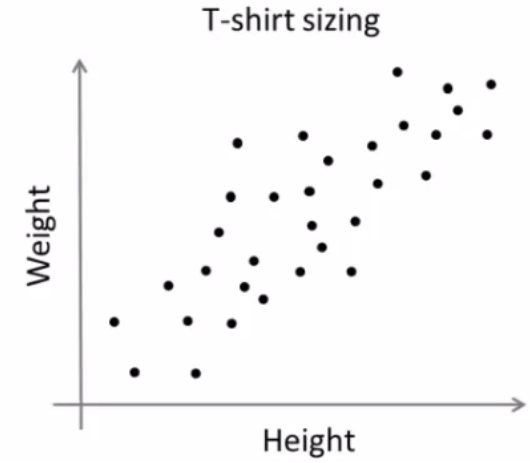
      - Loop 2
        - Loops over each centroid calculate the average mean based on all the points

associated with each centroid from $c^{(i)}$

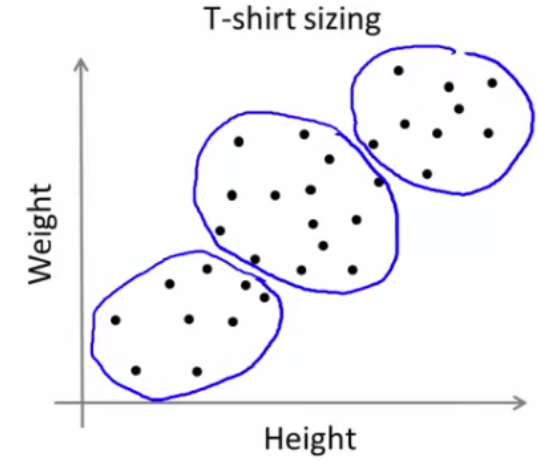
- What if there's a centroid with no data
  - Remove that centroid, so end up with K-1 classes
  - Or, randomly reinitialize it
    - Not sure when though...

K-means for non-separated clusters

- So far looking at K-means where we have well defined clusters
- But often K-means is applied to datasets where there aren't well defined clusters
  - e.g. T-shirt sizing



- Not obvious discrete groups
- Say you want to have three sizes (S,M,L) how big do you make these?
  - One way would be to run K-means on this data
  - May do the following



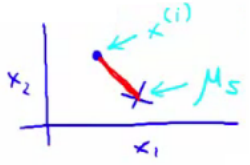
  - So creates three clusters, even though they aren't really there
  - Look at first population of people
    - Try and design a small T-shirt which fits the 1st population
    - And so on for the other two
  - This is an example of market segmentation
    - Build products which suit the needs of your subpopulations

K means optimization objective

- Supervised learning algorithms have an optimization objective (cost function)
  - K-means does too
- K-means has an optimization objective like the supervised learning functions we've seen
  - Why is this good?
  - Knowing this is useful because it helps for debugging
  - Helps find better clusters
- While K-means is running we keep track of two sets of variables
  - c^i is the index of clusters {1,2, ..., K} to which x^i is currently assigned
    - i.e. there are *m* c^i values, as each example has a c^i value, and that value is one the the clusters (i.e. can only be one of K different values)
  - μ_k, is the cluster associated with centroid *k*
    - Locations of cluster centroid k
    - So there are K
    - So these the centroids which exist in the training data space
  - μ_{c^i}, is the cluster centroid of the cluster to which example x^i has been assigned to
    - This is more for convenience than anything else
      - You could look up that example i is indexed to cluster j (using the c vector), where j is between 1 and K
      - Then look up the value associated with cluster j in the μ vector (i.e. what are the features associated with μ_j)
      - But instead, for easy description, we have this variable which gets exactly the same value
    - Lets say x^i as been assigned to cluster 5
      - Means that
        - $c^i = 5$
        - $\mu_{c^i} = \mu_5$
- Using this notation we can write the optimization objective;

$$J(c^{(1)},\dots,c^{(m)},\mu_1,\dots,\mu_K) = \frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - \mu_{c^{(i)}}||^2$$

  - i.e. squared distances between training example x^i and the cluster centroid to which x^i has been assigned to
    - This is just what we've been doing, as the visual description below shows;



    - The red line here shows the distances between the example x^i and the cluster to which that example has been assigned
      - Means that when the example is very close to the cluster, this value is small
      - When the cluster is very far away from the example, the value is large
  - This is sometimes called the **distortion** (or **distortion cost function**)
  - So we are finding the values which minimizes this function;

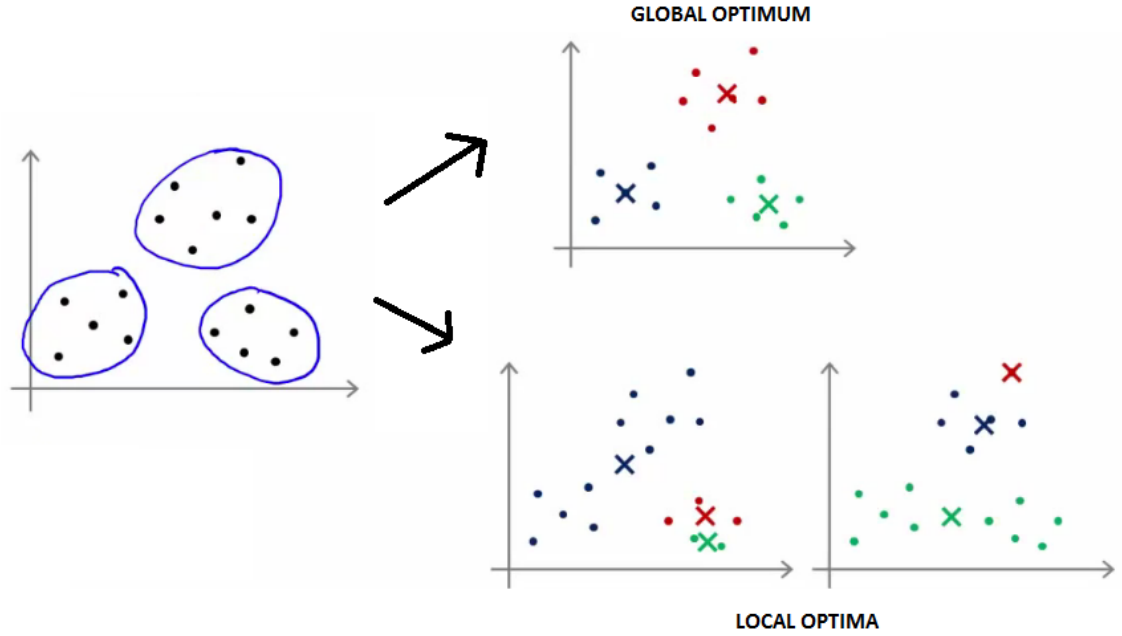
$$\min_{\substack{c^{(1)},\dots,c^{(m)},\\ \mu_1,\dots,\mu_K}} J(c^{(1)},\dots,c^{(m)},\mu_1,\dots,\mu_K)$$

- If we consider the k-means algorithm
  - The **cluster assigned step** is minimizing J(...) with respect to c^1, c^2 ... c^i

        - i.e. find the centroid closest to each example
        - Doesn't change the centroids themselves
    - The **move centroid step**
        - We can show this step is choosing the values of μ which minimizes J(...) with respect to μ
    - So, we're partitioning the algorithm into two parts
        - First part minimizes the c variables
        - Second part minimizes the J variables
- We can use this knowledge to help debug our K-means algorithm

Random initialization

- How we initialize K-means
    - And how avoid local optimum
- Consider clustering algorithm
    - Never spoke about how we initialize the centroids
        - A few ways - one method is most recommended
- Have number of centroids set to less than number of examples (K < m) (if K > m we have a problem)0
    - Randomly pick K training examples
    - Set μ_1 up to μ_K to these example's values
- K means can converge to different solutions depending on the initialization setup
    - Risk of local optimum



    - The local optimum are valid convergence, but local optimum not global ones
- If this is a concern
    - We can do multiple random initializations
        - See if we get the same result - many same results are likely to indicate a global optimum
- Algorithmically we can do this as follows;

For i = 1 to 100 {
Randomly initialize K-means.
Run K-means. Get $c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K$
Compute cost function (distortion)
$J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K)$ }

    - A typical number of times to initialize K-means is 50-1000
    - Randomly initialize K-means
        - For each 100 random initialization run K-means

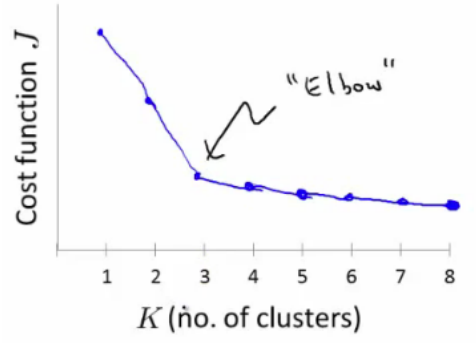
        - Then compute the distortion on the set of cluster assignments and centroids at convergent
        - End with 100 ways of cluster the data
        - Pick the clustering which gave the lowest distortion
- If you're running K means with 2-10 clusters can help find better global optimum
    - If K is larger than 10, then multiple random initializations are less likely to be necessary
    - First solution is probably good enough (better granularity of clustering)

How do we choose the number of clusters?

- Choosing K?
    - Not a great way to do this automatically
    - Normally use visualizations to do it manually
- What are the intuitions regarding the data?
- Why is this hard
    - Sometimes very ambiguous
        - e.g. two clusters or four clusters
        - Not necessarily a correct answer
    - This is why doing it automatic this is hard

Elbow method

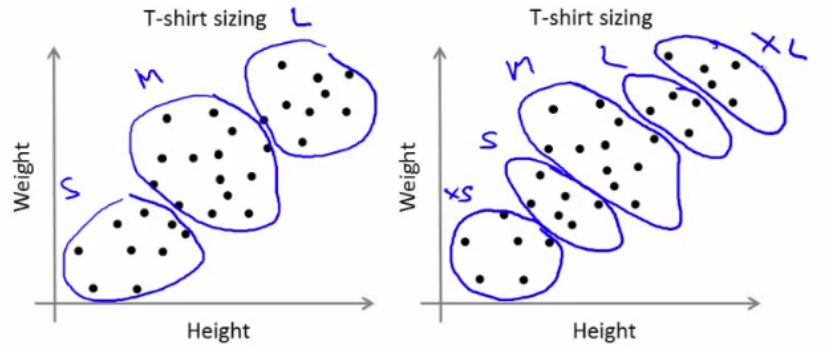
- Vary K and compute cost function at a range of K values
- As K increases J(...) minimum value should decrease (i.e. you decrease the granularity so centroids can better optimize)
    - Plot this (K vs J())
- Look for the "elbow" on the graph



- Chose the "elbow" number of clusters
- If you get a nice plot this is a reasonable way of choosing K
- Risks
    - Normally you don't get a a nice line -> no clear elbow on curve
    - Not really that helpful

Another method for choosing K

- Using K-means for market segmentation
- Running K-means for a later/downstream purpose
    - See how well different number of clusters serve you later needs
- e.g.
    - T-shirt size example
        - If you have three sizes (S,M,L)
        - Or five sizes (XS, S, M, L, XL)
        - Run K means where K = 3 and K = 5
    - How does this look



- This gives a way to chose the number of clusters
  - Could consider the cost of making extra sizes vs. how well distributed the products are
  - How important are those sizes though? (e.g. more sizes might make the customers happier)
  - So applied problem may help guide the number of clusters

14: Dimensionality Reduction (PCA)

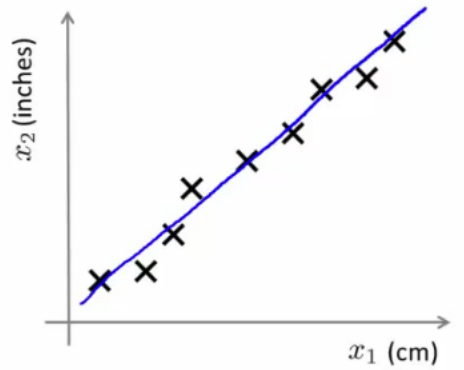
Previous Next Index

Motivation 1: Data compression

- Start talking about a second type of unsupervised learning problem - **dimensionality reduction**
  - Why should we look at dimensionality reduction?

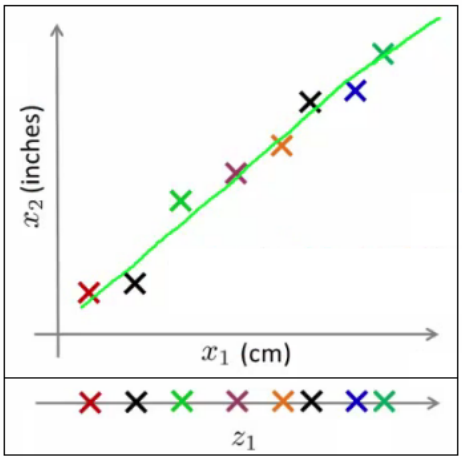
Compression

- Speeds up algorithms
- Reduces space used by data for them
- What is dimensionality reduction?
  - So you've collected many features - maybe more than you need
    - Can you "simply" your data set in a rational and useful way?
  - Example
    - Redundant data set - different units for same attribute
    - Reduce data to 1D (2D->1D)



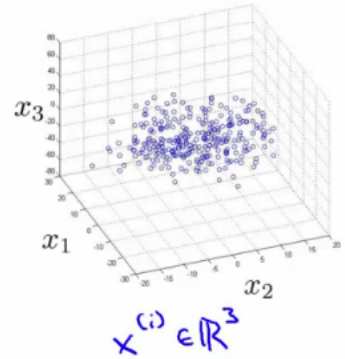
    - Example above isn't a perfect straight line because of round-off error
  - Data redundancy can happen when different teams are working independently
    - Often generates redundant data (especially if you don't control data collection)
  - Another example
    - Helicopter flying - do a survey of pilots (x1 = skill, x2 = pilot enjoyment)
      - These features may be highly correlated
      - This correlation can be combined into a single attribute called aptitude (for example)

- What does dimensionality reduction mean?
  - In our example we plot a line
  - Take exact example and record position on that line

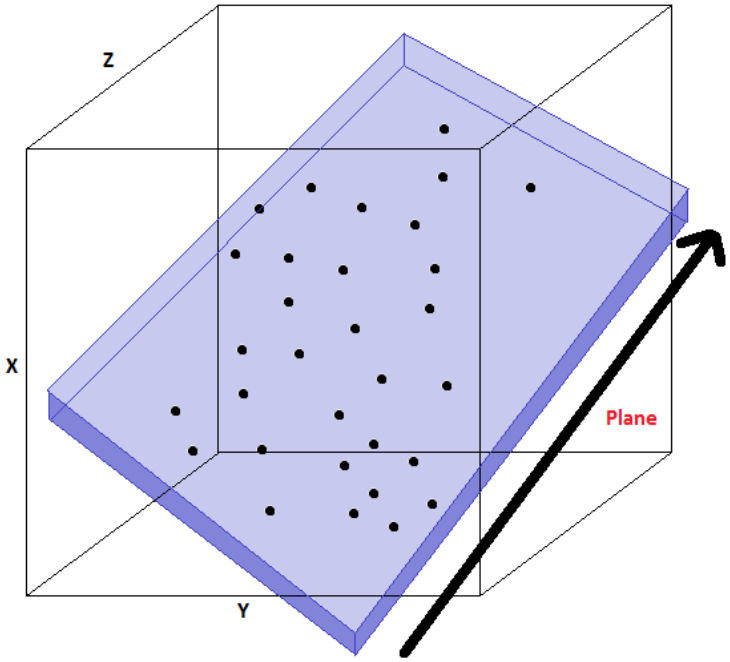


- So before x^1 was a 2D feature vector (X and Y dimensions)
- Now we can represent x^1 as a 1D number (Z dimension)

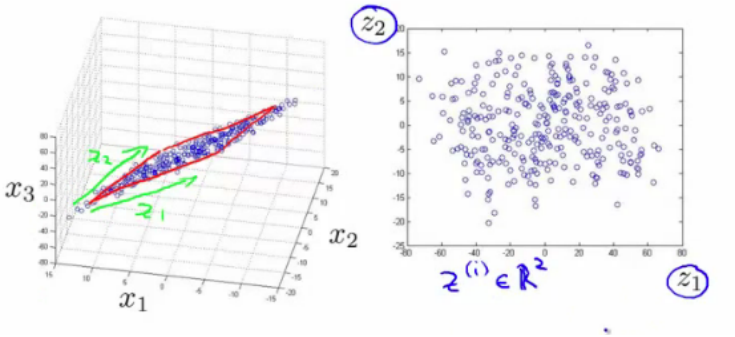
- So we approximate original examples
  - Allows us to half the amount of storage
  - Gives lossy compression, but an acceptable loss (probably)
    - The loss above comes from the rounding error in the measurement, however
- Another example 3D -> 2D
  - So here's our data



  - Maybe all the data lies in one plane
    - This is sort of hard to explain in 2D graphics, but that plane may be aligned with one of the axis
      - Or or may not...
      - Either way, the plane is a small, a constant 3D space
    - In the diagram below, imagine all our data points are sitting "inside" the blue tray (has a dark blue exterior face and a light blue inside)



        - Because they're all in this relative shallow area, we can basically ignore one of the dimension, so we draw two new lines (z1 and z2) along the x and y planes of the box, and plot the locations in that box
        - i.e. we loose the data in the z-dimension of our "shallow box" (NB "z-dimensions" here refers to the dimension relative to the box (i.e it's depth) and NOT the z dimension of the axis we've got drawn above) but because the box is shallow it's OK to lose this. Probably....
    - Plot values along those projections



    - So we've now reduced our 3D vector to a 2D vector
- In reality we'd normally try and do 1000D -> 100D

Motivation 2: Visualization

- It's hard to visualize highly dimensional data
    - Dimensionality reduction can improve how we display information in a tractable manner for human consumption
    - Why do we care?
        - Often helps to develop algorithms if we can understand our data better
        - Dimensionality reduction helps us do this, see data in a helpful
        - Good for explaining something to someone if you can "show" it in the data
- Example;
    - Collect a large data set about many facts of a country around the world

Country	GDP (trillions of US$)	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	...
...	...	...	...	...	...	...	

- So
  - x_1 = GDP
  - ...
  - x_6 = mean household
- Say we have 50 features per country
- How can we understand this data better?
  - Very hard to plot 50 dimensional data

- Using dimensionality reduction, instead of each country being represented by a 50-dimensional feature vector
  - Come up with a different feature representation (z values) which summarize these features

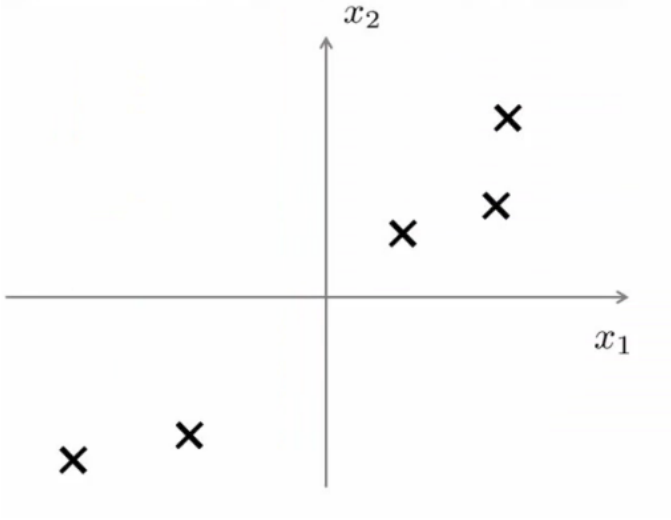
Country	z_1	z_2
Canada	1.6	1.2
China	1.7	0.3
India	1.6	0.2
Russia	1.4	0.5
Singapore	0.5	1.7
USA	2	1.5
...	...	...

- This gives us a 2-dimensional vector
  - Reduce 50D -> 2D
  - Plot as a 2D plot
- Typically you don't generally ascribe meaning to the new features (so we have to determine what these summary values mean)
  - e.g. may find horizontal axis corresponds to overall country size/economic activity
  - and y axis may be the per-person well being/economic activity
- So despite having 50 features, there may be two "dimensions" of information, with features associated with each of those dimensions
  - It's up to you to asses what of the features can be grouped to form summary features, and how best to do that (feature scaling is probably important)

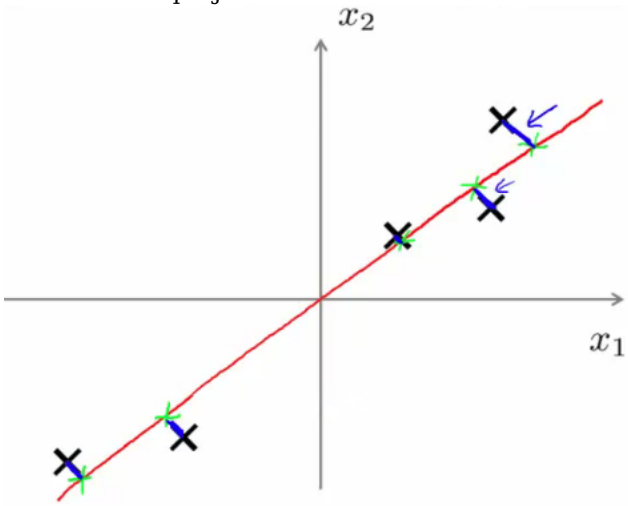
- Helps show the two main dimensions of variation in a way that's easy to understand

Principle Component Analysis (PCA): Problem Formulation

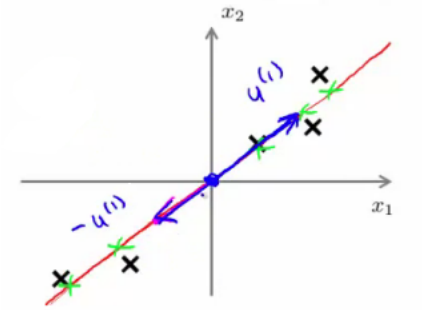
- For the problem of dimensionality reduction the most commonly used algorithm is **PCA**
  - Here, we'll start talking about how we formulate precisely what we want PCA to do
- So
  - Say we have a 2D data set which we wish to reduce to 1D



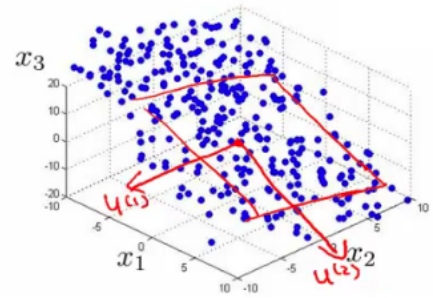
  - In other words, find a single line onto which to project this data
    - How do we determine this line?
      - The distance between each point and the projected version should be small (blue lines below are short)
      - PCA tries to find a lower dimensional surface so the sum of squares onto that surface is minimized
      - The blue lines are sometimes called the **projection error**
        - PCA tries to find the surface (a straight line in this case) which has the minimum projection error



      - As an aside, you should normally do **mean normalization** and **feature scaling** on your data before PCA
- A more formal description is
  - For 2D-1D, we must find a vector $u^{(1)}$, which is of some dimensionality
  - Onto which you can project the data so as to minimize the projection error



- $u^{(1)}$ can be positive or negative ($-u^{(1)}$) which makes no difference
  - Each of the vectors define the same red line
- In the more general case
  - To reduce from nD to kD we
    - Find *k* vectors ($u^{(1)}$, $u^{(2)}$, ... $u^{(k)}$) onto which to project the data to minimize the projection error
    - So lots of vectors onto which we project the data
    - Find a set of vectors which we project the data onto the linear subspace spanned by that set of vectors
      - We can define a point in a plane with k vectors
  - e.g. 3D->2D
    - Find pair of vectors which define a 2D plane (surface) onto which you're going to project your data
    - Much like the "shallow box" example in compression, we're trying to create the shallowest box possible (by defining two of it's three dimensions, so the box' depth is minimized)

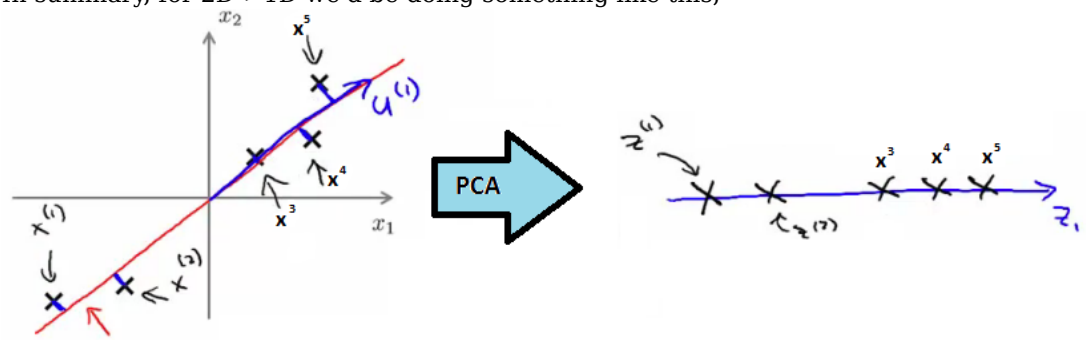


- How does PCA relate to linear regression?
  - PCA is **not** linear regression
    - Despite cosmetic similarities, very different
  - For linear regression, fitting a straight line to minimize the **straight line** between a point and a squared line
    - NB - **VERTICAL distance** between point
  - For PCA minimizing the magnitude of the shortest **orthogonal distance**
    - Gives very different effects
  - More generally
    - With linear regression we're trying to predict "y"
    - With PCA there is no "y" - instead we have a list of features and all features are treated equally
      - If we have 3D dimensional data 3D->2D
        - Have 3 features treated symmetrically

PCA Algorithm

- Before applying PCA must do data preprocessing
  - Given a set of m unlabeled examples we must do
    - **Mean normalization**
      - Replace each x_j^i with $x_j - \mu_j$,
        - In other words, determine the mean of each feature set, and then for each feature

subtract the mean from the value, so we re-scale the mean to be 0
        - **Feature scaling (depending on data)**
            - If features have very different scales then scale so they all have a comparable range of values
                - e.g. x_j^i is set to $(x_j - \mu_j) / s_j$
                    - Where s_j is some measure of the range, so could be
                        - Biggest - smallest
                        - Standard deviation (more commonly)
- With preprocessing done, PCA finds the lower dimensional sub-space which minimizes the sum of the square
    - In summary, for 2D->1D we'd be doing something like this;



    - Need to compute two things;
        - Compute the **u vectors**
            - The new planes
        - Need to compute the **z vectors**
            - z vectors are the new, lower dimensionality feature vectors
- A mathematical derivation for the u vectors is very complicated
    - But once you've done it, the procedure to find each u vector is not that hard

Algorithm description

- Reducing data from *n*-dimensional to k-dimensional
    - Compute the covariance matrix

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

        - This is commonly denoted as Σ (greek upper case sigma) - NOT summation symbol
        - Σ = sigma
            - This is an [n x n] matrix
                - Remember than x^i is a [n x 1] matrix
        - In MATLAB or octave we can implement this as follows;

```
sigma = (1/m) * ( X' * X )
```

    - Compute eigenvectors of matrix Σ

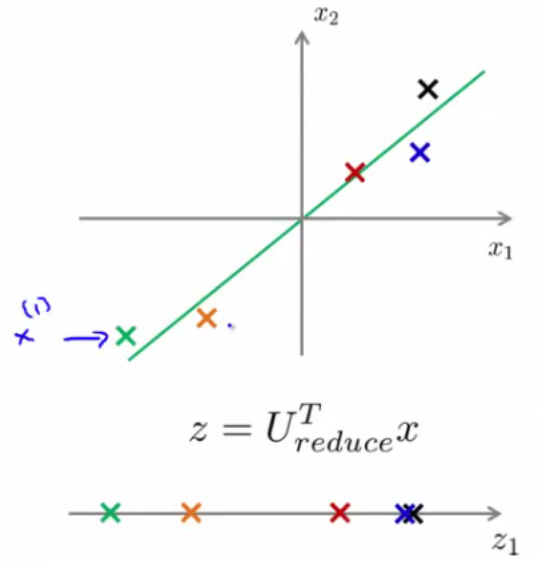
        - `[U,S,V] = svd(sigma)`
            - svd = singular value decomposition
                - More numerically stable than `eig`
            - `eig` = also gives eigenvector
    - U,S and V are matrices
        - U matrix is also an [n x n] matrix
        - Turns out the columns of U are the u vectors we want!
        - So to reduce a system from n-dimensions to k-dimensions
            - Just take the first *k-vectors* from U (first k columns)

$$U = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(n)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n \times n}$$

- Next we need to find some way to change x (which is n dimensional) to z (which is k dimensional)
  - (reduce the dimensionality)
  - Take first k columns of the u matrix and stack in columns
    - n x k matrix - call this U_{reduce}
  - We calculate z as follows
    - z = $(U_{reduce})^T$ * x
      - So [k x n] * [n x 1]
      - Generates a matrix which is
        - k * 1
      - If that's not witchcraft I don't know what is!
- Exactly the same as with supervised learning except we're now doing it with unlabeled data
- So in summary
  - Preprocessing
  - Calculate sigma (covariance matrix)
  - Calculate eigenvectors with **svd**
  - Take k vectors from U (U_{reduce}= U(:,1:k);)
  - Calculate z (z =U_{reduce}' * x;)
- No mathematical derivation
  - Very complicated
  - But it works

Reconstruction from Compressed Representation

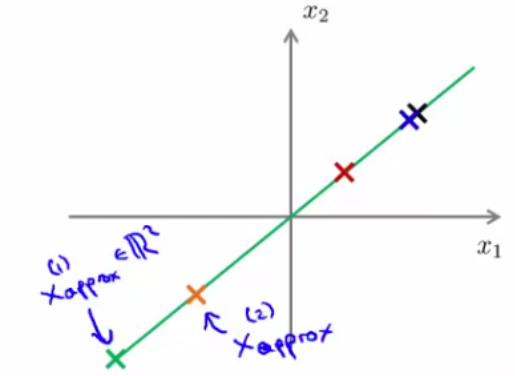
- Earlier spoke about PCA as a compression algorithm
  - If this is the case, is there a way to **decompress** the data from low dimensionality back to a higher dimensionality format?
- Reconstruction
  - Say we have an example as follows



  - We have our examples (x^1, x^2 etc.)

- Project onto z-surface
- Given a point z^1, how can we go back to the 2D space?

- Considering
  - z (vector) = $(U_{reduce})^T * x$
- To go in the opposite direction we must do
  - $x_{approx} = U_{reduce} * z$
    - To consider dimensions (and prove this really works)
      - U_{reduce} = [n x k]
      - z [k * 1]
    - So
      - x_{approx} = [n x 1]
- So this creates the following representation



- We lose some of the information (i.e. everything is now perfectly on that line) but it is now projected into 2D space

Choosing the number of Principle Components

- How do we chose *k* ?
  - k = number of **principle components**
  - Guidelines about how to chose k for PCA
- To chose k think about how PCA works
  - PCA tries to minimize averaged squared projection error

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

  - Total variation in data can be defined as the average over data saying how far are the training examples from the origin

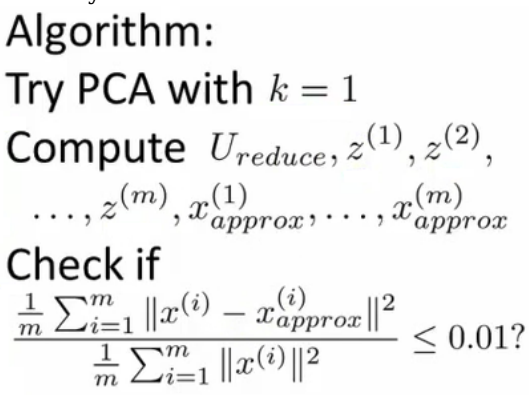
$$\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2$$

- When we're choosing k typical to use something like this

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

  - Ratio between averaged squared projection error with total variation in data
    - Want ratio to be small - means we retain 99% of the variance
  - If it's small (0) then this is because the numerator is small
    - The numerator is small when $x^i = x_{approx}^i$
      - i.e. we lose very little information in the dimensionality reduction, so when we decompress we regenerate the same data

- So we chose k in terms of this ratio
- Often can significantly reduce data dimensionality while retaining the variance
- How do you do this



Advice for Applying PCA

- Can use PCA to speed up algorithm running time
  - Explain how
  - And give general advice

Speeding up supervised learning algorithms

- Say you have a supervised learning problem
  - Input x and y
    - x is a 10 000 dimensional feature vector
    - e.g. 100 x 100 images = 10 000 pixels
    - Such a huge feature vector will make the algorithm slow
  - With PCA we can reduce the dimensionality and make it tractable
  - How
    - 1) Extract xs
      - So we now have an unlabeled training set
    - 2) Apply PCA to x vectors
      - So we now have a reduced dimensional feature vector z
    - 3) This gives you a new training set
      - Each vector can be re-associated with the label
    - 4) Take the reduced dimensionality data set and feed to a learning algorithm
      - Use y as labels and z as feature vector
    - 5) If you have a new example map from higher dimensionality vector to lower dimensionality vector, then feed into learning algorithm
- PCA maps one vector to a lower dimensionality vector
  - x -> z
  - Defined by PCA **only** on the training set
  - The mapping computes a set of parameters
    - Feature scaling values
    - U_{reduce}
      - Parameter learned by PCA
      - Should be obtained only by determining PCA on your training set
  - So we use those learned parameters for our
    - Cross validation data
    - Test set
- Typically you can reduce data dimensionality by 5-10x without a major hit to algorithm

Applications of PCA

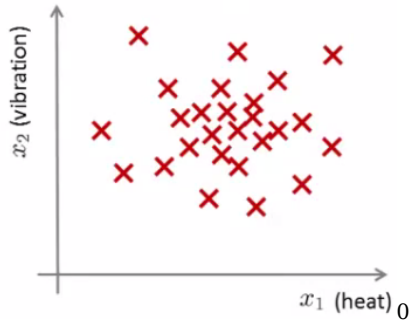
- **Compression**
  - Why
    - Reduce memory/disk needed to store data
    - Speed up learning algorithm
  - How do we chose k?
    - % of variance retained
- **Visualization**
  - Typically chose k =2 or k = 3
  - Because we can plot these values!
- One thing often done wrong regarding PCA
  - A bad use of PCA: Use it to prevent over-fitting
    - Reasoning
      - If we have x^i we have n features, z^i has k features which can be lower
      - If we *only* have k features then maybe we're less likely to over fit...
    - This doesn't work
      - BAD APPLICATION
      - Might work OK, but not a good way to address over fitting
      - Better to use regularization
    - PCA throws away some data without knowing what the values it's losing
      - Probably OK if you're keeping most of the data
      - But if you're throwing away some crucial data bad
      - So you have to go to like 95-99% variance retained
        - So here regularization will give you AT LEAST as good a way to solve over fitting
- A second PCA myth
  - Used for compression or visualization - good
  - Sometimes used
    - Design ML system with PCA from the outset
      - But, what if you did the whole thing without PCA
    - See how a system performs without PCA
      - ONLY if you have a reason to believe PCA will help should you then add PCA
    - PCA is easy enough to add on as a processing step
      - Try without first!

15: Anomaly Detection

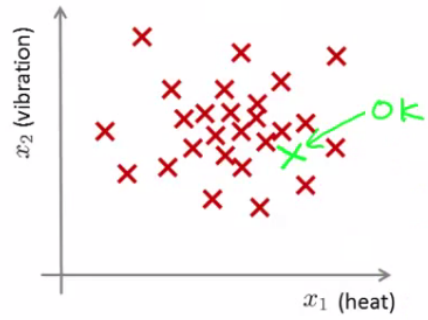
Previous Next Index

Anomaly detection - problem motivation

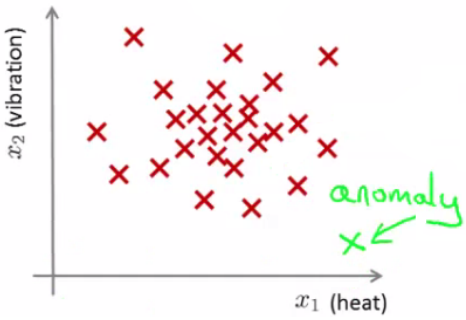
- Anomaly detection is a reasonably commonly used type of machine learning application
  - Can be thought of as a solution to an unsupervised learning problem
  - But, has aspects of supervised learning
- What is anomaly detection?
  - Imagine you're an aircraft engine manufacturer
  - As engines roll off your assembly line you're doing QA
    - Measure some features from engines (e.g. heat generated and vibration)
  - You now have a dataset of x^1 to x^m (i.e. m engines were tested)
  - Say we plot that dataset



  - Next day you have a new engine
    - An anomaly detection method is used to see if the new engine is anomalous (when compared to the previous engines)
  - If the new engine looks like this;

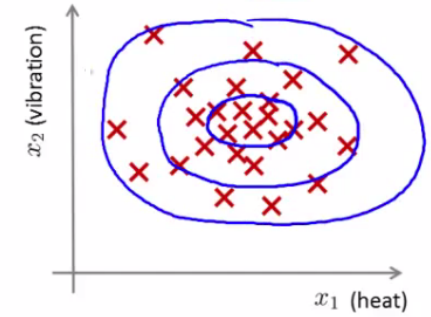


    - Probably OK - looks like the ones we've seen before
  - But if the engine looks like this



    - Uh oh! - this looks like an **anomalous data-point**
- More formally
  - We have a dataset which contains **normal** (data)
    - How we ensure they're normal is up to us
    - In reality it's OK if there are a few which aren't actually normal

    - Using that dataset as a reference point we can see if other examples are **anomalous**
- How do we do this?
    - First, using our training dataset we build a model
        - We can access this model using **p(x)**
            - This asks, "What is the probability that example x is normal"
    - Having built a model
        - if $p(x_{test}) < \varepsilon$ --> flag this as an anomaly
        - if $p(x_{test}) >= \varepsilon$ --> this is OK
        - ε is some threshold probability value which we define, depending on how sure we need/want to be
    - We expect our model to (graphically) look something like this;



        - i.e. this would be our model if we had 2D data

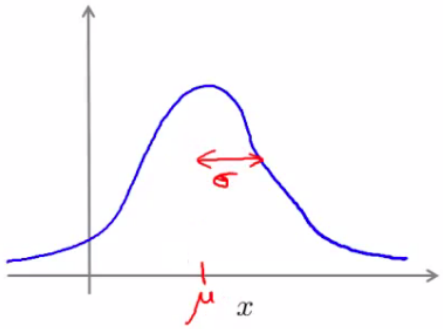
Applications

- Fraud detection
    - Users have activity associated with them, such as
        - Length on time on-line
        - Location of login
        - Spending frequency
    - Using this data we can build a model of what normal users' activity is like
    - What is the probability of "normal" behavior?
    - Identify unusual users by sending their data through the model
        - Flag up anything that looks a bit weird
        - Automatically block cards/transactions
- Manufacturing
    - Already spoke about aircraft engine example
- Monitoring computers in data center
    - If you have many machines in a cluster
    - Computer features of machine
        - x_1 = memory use
        - x_2 = number of disk accesses/sec
        - x_3 = CPU load
    - In addition to the measurable features you can also define your own complex features
        - x_4 = CPU load/network traffic
    - If you see an anomalous machine
        - Maybe about to fail
        - Look at replacing bits from it

The Gaussian distribution (optional)

- Also called the **normal distribution**
- Example
    - Say x (data set) is made up of real numbers
        - Mean is μ
        - Variance is σ^2
            - σ is also called the **standard deviation** - specifies the width of the Gaussian probability
        - The data has a Gaussian distribution
    - Then we can write this $\sim N(\mu,\sigma^2)$
        - $\sim$ means = is distributed as
        - N (should really be "script" N (even curlier!) -> means normal distribution

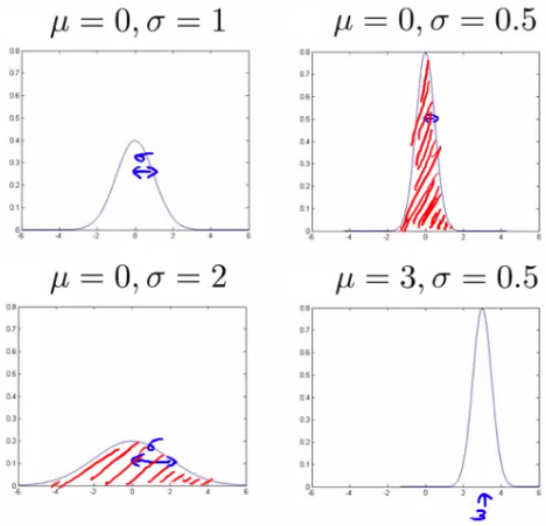
        - μ, σ^2 represent the mean and variance, respectively
            - These are the two parameters a Gaussian means
    - Looks like this;



    - This specifies the probability of x taking a value
        - As you move away from μ
- Gaussian equation is
    - $P(x : \mu, \sigma^2)$ (probability of x, parameterized by the mean and squared variance)

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

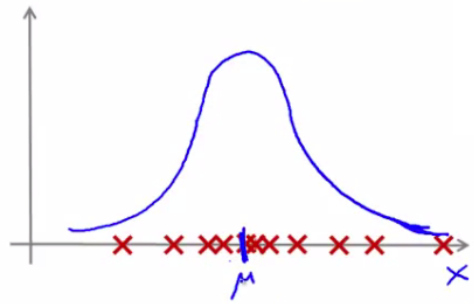
- Some examples of Gaussians below
    - Area is always the same (must = 1)
    - But width changes as standard deviation changes



Parameter estimation problem

- What is it?
    - Say we have a data set of m examples
    - Give each example is a real number - we can plot the data on the x axis as shown below

- Problem is - say you suspect these examples come from a Gaussian
  - Given the dataset can you estimate the distribution?
- Could be something like this



- Seems like a reasonable fit - data seems like a higher probability of being in the central region, lower probability of being further away

- Estimating μ and σ^2
  - μ = average of examples
  - σ^2 = standard deviation squared

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

  - As a side comment
    - These parameters are the maximum likelihood estimation values for μ and σ^2
    - You can also do 1/(m) or 1/(m-1) doesn't make too much difference
      - Slightly different mathematical problems, but in practice it makes little difference

Anomaly detection algorithm

- Unlabeled training set of m examples
  - Data = $\{x^1, x^2, ..., x^m\}$
    - Each example is an n-dimensional vector (i.e. a feature vector)
    - We have n features!
  - Model P(x) from the data set
    - What are high probability features and low probability features
    - x is a vector
    - So model p(x) as
      - = $p(x_1; \mu_1, \sigma_1^2) * p(x_2; \mu_2, \sigma_2^2) * ... p(x_n; \mu_n, \sigma_n^2)$
    - Multiply the probability of each features by each feature
      - We model each of the features by assuming each feature is distributed according to a Gaussian distribution
      - $p(x_i; \mu_i, \sigma_i^2)$
        - The probability of feature x_i given μ_i and σ_i^2, using a Gaussian distribution
  - As a side comment
    - Turns out this equation makes an **independence assumption** for the features, although algorithm works if features are independent or not
      - Don't worry too much about this, although if you're features are tightly linked you should be able to do some dimensionality reduction anyway!
  - We can write this chain of multiplication more compactly as follows;

$$= \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

    - Capital PI (Π) is the product of a set of values
  - The problem of estimation this distribution is sometimes call the problem of **density estimation**

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 $p(x)$:

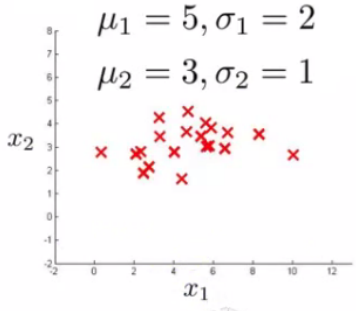
$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Anomaly if $p(x) < \varepsilon$

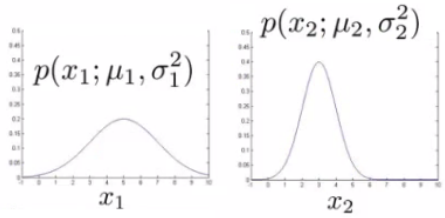
- **1 - Chose features**
  - Try to come up with features which might help identify something anomalous - may be unusually large or small values
  - More generally, chose features which describe the general properties
  - This is nothing unique to anomaly detection - it's just the idea of building a sensible feature vector
- **2 - Fit parameters**
  - Determine parameters for each of your examples μ_i and σ_i^2
    - Fit is a bit misleading, really should just be "Calculate parameters for 1 to n"
  - So you're calculating standard deviation and mean for each feature
  - You should of course used some vectorized implementation rather than a loop probably
- **3 - compute p(x)**
  - You compute the formula shown (i.e. the formula for the Gaussian probability)
  - If the number is very small, very low chance of it being "normal"

Anomaly detection example

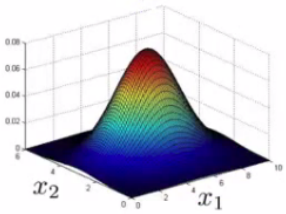
- x_1
  - Mean is about 5
  - Standard deviation looks to be about 2
- x_2
  - Mean is about 3
  - Standard deviation about 1
- So we have the following system



- If we plot the Gaussian for x_1 and x_2 we get something like this



- If you plot the product of these things you get a surface plot like this



    - With this surface plot, the height of the surface is the probability - p(x)
    - We can't always do surface plots, but for this example it's quite a nice way to show the probability of a 2D feature vector
- Check if a value is anomalous
    - Set epsilon as some value
    - Say we have two new data points new data-point has the values
        - x^1_{test}
        - x^2_{test}
    - We compute
        - $p(x^1_{test}) = 0.436 >= epsilon$ (~40% chance it's normal)
            - Normal
        - $p(x^2_{test}) = 0.0021 < epsilon$ (~0.2% chance it's normal)
            - Anomalous
    - What this is saying is if you look at the surface plot, all values above a certain height are normal, all the values below that threshold are probably anomalous

Developing and evaluating and anomaly detection system

- Here talk about developing a system for anomaly detection
    - How to evaluate an algorithm
- Previously we spoke about the importance of real-number evaluation
    - Often need to make a lot of choices (e.g. features to use)
        - Easier to evaluate your algorithm if it returns a **single number** to show if changes you made improved or worsened an algorithm's performance
    - To develop an anomaly detection system quickly, would be helpful to have a way to evaluate your algorithm
- Assume we have some labeled data
    - So far we've been treating anomalous detection with unlabeled data
    - If you have labeled data allows evaluation
        - i.e. if you think something iss anomalous you can be sure if it is or not
- So, taking our engine example
    - You have some labeled data
        - Data for engines which were non-anomalous -> y = 0
        - Data for engines which were anomalous -> y = 1
    - Training set is the collection of normal examples
        - OK even if we have a few anomalous data examples
    - Next define
        - Cross validation set
        - Test set
        - For both assume you can include a few examples which have anomalous examples
    - Specific example
        - Engines
            - Have 10 000 good engines
                - OK even if a few bad ones are here...
                - LOTS of y = 0
            - 20 flawed engines
                - Typically when y = 1 have 2-50
        - Split into
            - Training set: 6000 good engines (y = 0)
            - CV set: 2000 good engines, 10 anomalous
            - Test set: 2000 good engines, 10 anomalous
            - Ratio is 3:1:1
        - Sometimes we see a different way of splitting
            - Take 6000 good in training
            - Same CV and test set (4000 good in each) different 10 anomalous,
            - Or even 20 anomalous (same ones)

        - This is bad practice - should use different data in CV and test set
    - Algorithm evaluation
        - Take trainings set { x^1, x^2, ..., x^m }
            - Fit model p(x)
        - On cross validation and test set, test the example x
            - y = 1 if p(x) < epsilon (anomalous)
            - y = 0 if p(x) >= epsilon (normal)
        - Think of algorithm a trying to predict if something is anomalous
            - But you have a label so can check!
            - Makes it look like a supervised learning algorithm
- What's a good metric to use for evaluation
    - y = 0 is very common
        - So classification would be bad
    - Compute fraction of true positives/false positive/false negative/true negative
    - Compute precision/recall
    - Compute F1-score
- Earlier, also had **epsilon** (the threshold value)
    - Threshold to show when something is anomalous
    - If you have CV set you can see how varying epsilon effects various evaluation metrics
        - Then pick the value of epsilon which maximizes the score on your CV set
    - Evaluate algorithm using cross validation
    - Do final algorithm evaluation on the test set

Anomaly detection vs. supervised learning

- If we have labeled data, we not use a supervised learning algorithm?
    - Here we'll try and understand when you should use supervised learning and when anomaly detection would be better

Anomaly detection

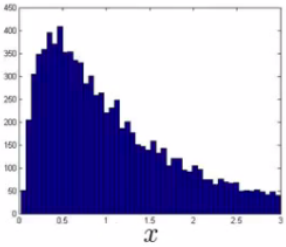
- **Very small number of positive examples**
    - Save positive examples just for CV and test set
    - Consider using an anomaly detection algorithm
    - Not enough data to "learn" positive examples
- **Have a very large number of negative examples**
    - Use these negative examples for p(x) fitting
    - Only need negative examples for this
- **Many "types" of anomalies**
    - Hard for an algorithm to learn from positive examples when anomalies may look nothing like one another
        - So anomaly detection doesn't know what they look like, but knows what they *don't* look like
    - When we looked at SPAM email,
        - Many types of SPAM
        - For the spam problem, usually enough positive examples
        - So this is why we usually think of SPAM as supervised learning
- Application and why they're anomaly detection
    - **Fraud detection**
        - Many ways you may do fraud
        - If you're a major on line retailer/very subject to attacks, sometimes might shift to supervised learning
    - **Manufacturing**
        - If you make HUGE volumes maybe have enough positive data -> make supervised
            - Means you make an assumption about the kinds of errors you're going to see
            - It's the unknown unknowns we don't like!
    - **Monitoring machines in data**

Supervised learning

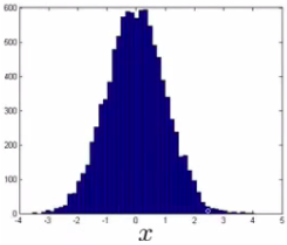
- **Reasonably large number of positive and negative examples**
- Have enough positive examples to give your algorithm the opportunity to see what they look like
    - If you expect anomalies to look anomalous in the same way
- Application
    - Email/SPAM classification
    - Weather prediction
    - Cancer classification

Choosing features to use

- One of the things which has a huge effect is which features are used
- **Non-Gaussian features**
  - Plot a histogram of data to check it has a Gaussian description - nice sanity check
    - Often still works if data is non-Gaussian
    - Use **hist** command to plot histogram
  - Non-Gaussian data might look like this



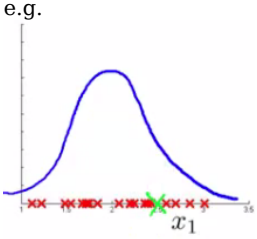
  - Can play with different transformations of the data to make it look more Gaussian
  - Might take a log transformation of the data
    - i.e. if you have some feature x_1, replace it with $\log(x_1)$



      - This looks much more Gaussian
    - Or do $\log(x_1+c)$
      - Play with c to make it look as Gaussian as possible
    - Or do $x^{1/2}$
    - Or do $x^{1/3}$

Error analysis for anomaly detection

- Good way of coming up with features
- Like supervised learning error analysis procedure
  - Run algorithm on CV set
  - See which one it got wrong
  - Develop new features based on trying to understand *why* the algorithm got those examples wrong
- Example
  - p(x) large for normal, p(x) small for abnormal
  - e.g.

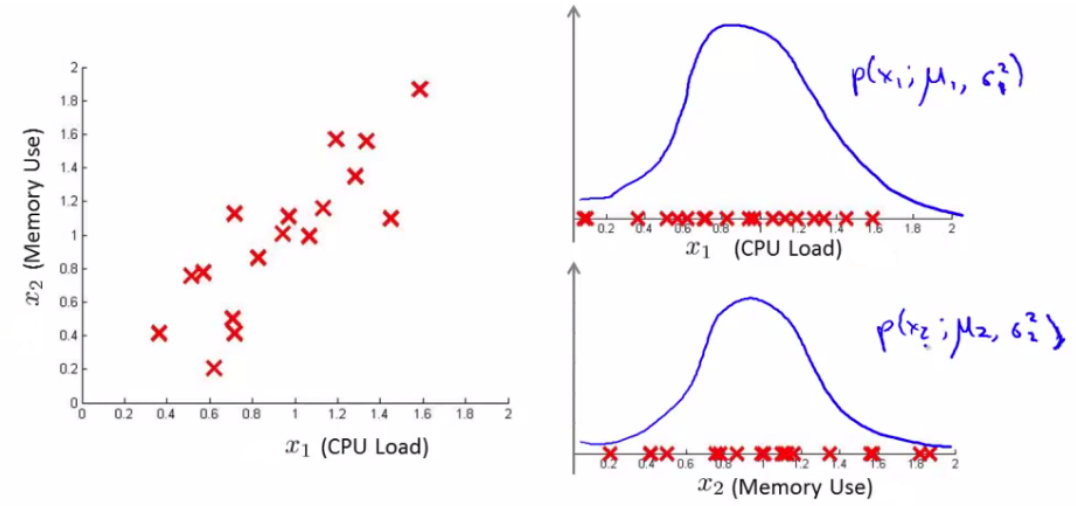


  - Here we have one dimension, and our anomalous value is sort of buried in it (in green - Gaussian superimposed in blue)
    - Look at data - see what went wrong
    - Can looking at that example help develop a new feature (x2) which can help distinguish further anomalous
- Example - data center monitoring
  - Features
    - x_1 = memory use
    - x_2 = number of disk access/sec
    - x_3 = CPU load
    - x_4 = network traffic
  - We suspect CPU load and network traffic grow linearly with one another
    - If server is serving many users, CPU is high and network is high
    - Fail case is infinite loop, so CPU load grows but network traffic is low

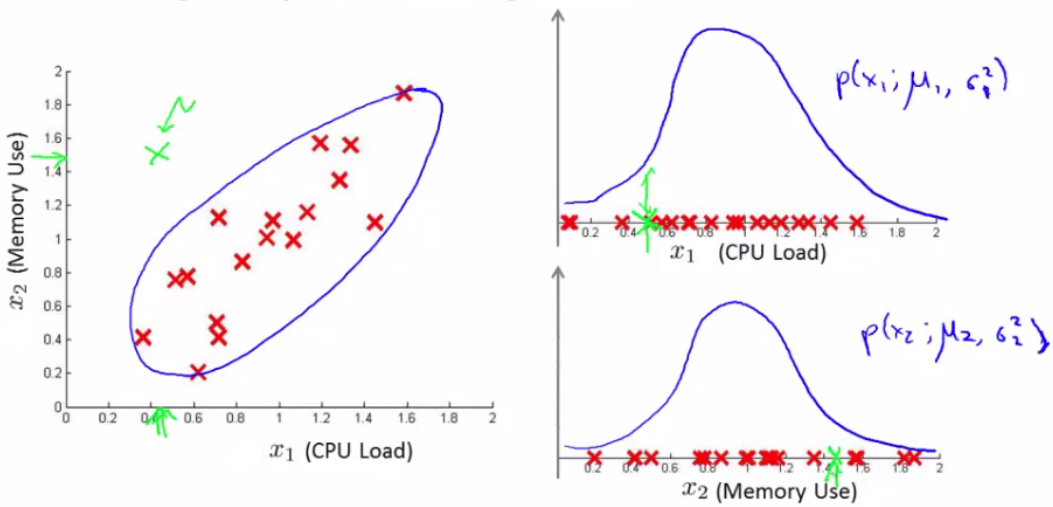
- New feature - CPU load/network traffic
- May need to do feature scaling

Multivariate Gaussian distribution

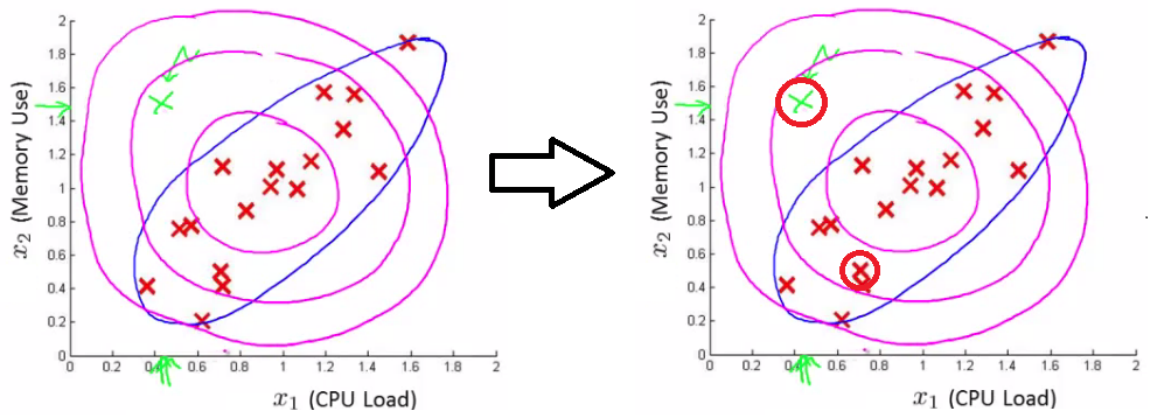
- Is a slightly different technique which can sometimes catch some anomalies which non-multivariate Gaussian distribution anomaly detection fails to
  - Unlabeled data looks like this



  - Say you can fit a Gaussian distribution to CPU load and memory use
  - Lets say in the test set we have an example which looks like an anomaly (e.g. $x_1 = 0.4$, $x_2 = 1.5$)
    - Looks like most of data lies in a region far away from this example
      - Here memory use is high and CPU load is low (if we plot x_1 vs. x_2 our green example looks miles away from the others)
  - Problem is, if we look at each feature individually they may fall within acceptable limits - the issue is we know we shouldn't don't get those kinds of values **together**
    - But individually, they're both acceptable



  - This is because our function makes probability prediction in concentric circles around the the means of both



- Probability of the two red circled examples is basically the same, even though we can clearly see the green one as an outlier

    - Doesn't understand the meaning

Multivariate Gaussian distribution model

- To get around this we develop the **multivariate Gaussian distribution**
    - Model p(x) all in one go, instead of each feature separately
        - What are the parameters for this new model?
            - μ - which is an *n* dimensional vector (where n is number of features)
            - Σ - which is an [n x n] matrix - the **covariance matrix**

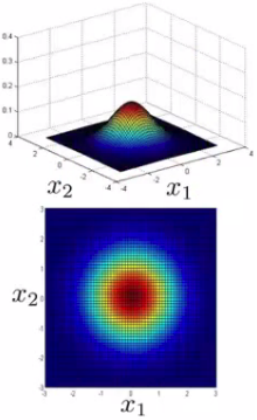
- For the sake of completeness, the formula for the multivariate Gaussian distribution is as follows

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

    - NB don't memorize this - you can always look it up
    - What does this mean?
        - $|\Sigma|$ = absolute value of Σ (determinant of sigma)
            - This is a mathematic function of a matrix
            - You can compute it in MATLAB using `det(sigma)`
- More importantly, what does this p(x) look like?
    - 2D example

$$\mu = \begin{bmatrix}0\\0\end{bmatrix} \Sigma = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

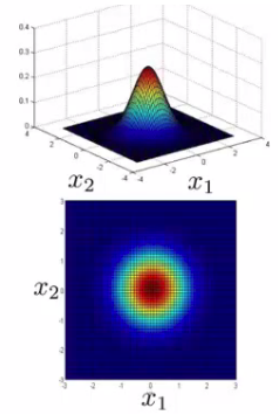
        - Sigma is sometimes call the identity matrix



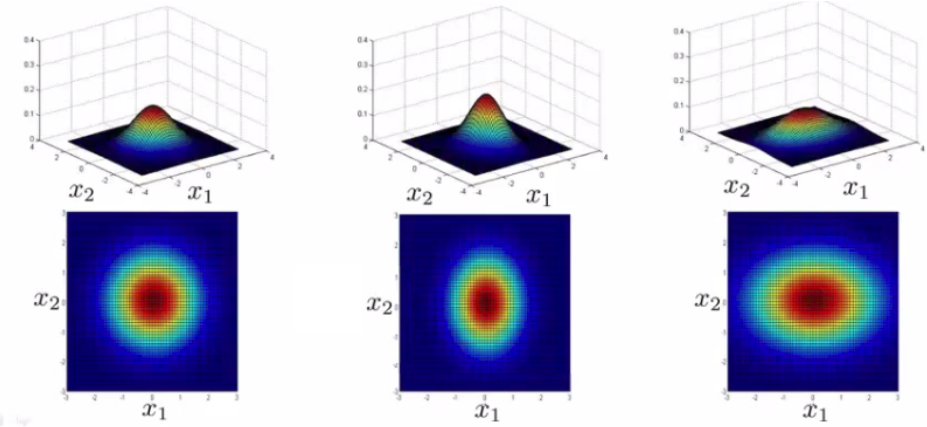
        - p(x) looks like this
            - For inputs of x_1 and x_2 the height of the surface gives the value of p(x)
    - What happens if we change Sigma?

$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 0.6 & 0 \\ 0 & 0.6 \end{bmatrix}$$

- So now we change the plot to



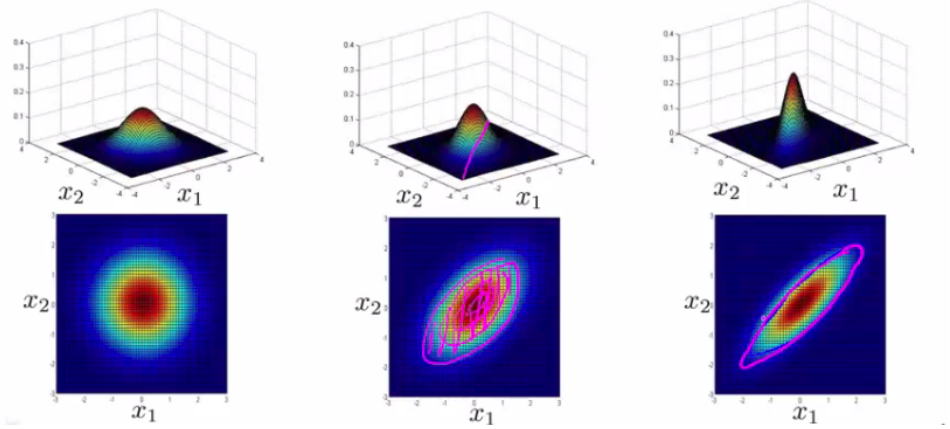
  - Now the width of the bump decreases and the height increases
- If we set sigma to be different values this changes the identity matrix and we change the shape of our graph



- Using these values we can, therefore, define the shape of this to better fit the data, rather than assuming symmetry in every dimension

- One of the cool things is you can use it to model correlation between data
  - If you start to change the off-diagonal values in the covariance matrix you can control how well the various dimensions correlation

$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix} \quad \mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$$



  - So we see here the final example gives a very tall thin distribution, shows a strong positive correlation
  - We can also make the off-diagonal values negative to show a negative correlation
- Hopefully this shows an example of the kinds of distribution you can get by varying sigma

  - We can, of course, also move the mean (μ) which varies the peak of the distribution

Applying multivariate Gaussian distribution to anomaly detection

- Saw some examples of the kinds of distributions you can model
  - Now let's take those ideas and look at applying them to different anomaly detection algorithms
- As mentioned, multivariate Gaussian modeling uses the following equation;

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

- Which comes with the parameters μ and Σ
  - Where
    - μ - the mean (n-dimenisonal vector)
    - Σ - covariance matrix ([nxn] matrix)
- Parameter fitting/estimation problem
  - If you have a set of examples
    - $\{x^1, x^2, ..., x^m\}$
  - The formula for estimating the parameters is

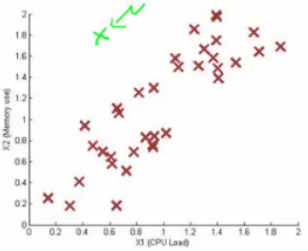
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

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

  - Using these two formulas you get the parameters

Anomaly detection algorithm with multivariate Gaussian distribution

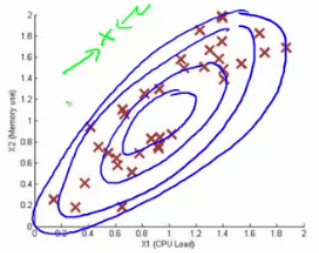
- **1)** Fit model - take data set and calculate μ and Σ using the formula above
- **2)** We're next given a new example (x_{test}) - see below



  - For it compute p(x) using the following formula for multivariate distribution

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

- **3)** Compare the value with ε (threshold probability value)
  - if $p(x_{test}) < \varepsilon$ --> flag this as an anomaly
  - if $p(x_{test}) >= \varepsilon$ --> this is OK
- If you fit a multivariate Gaussian model to our data we build something like this



- Which means it's likely to identify the green value as anomalous
- Finally, we should mention how multivariate Gaussian relates to our original simple Gaussian model (where each feature is looked at individually)
  - Original model corresponds to multivariate Gaussian where the Gaussians' contours are axis aligned
  - i.e. the normal Gaussian model is a special case of multivariate Gaussian distribution
    - This can be shown mathematically

- Has this constraint that the covariance matrix sigma as ZEROs on the non-diagonal values

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

where $\Sigma = \begin{bmatrix} \sigma_1^2 & & & 0 \\ & \sigma_2^2 & & \\ & & \ddots & \\ 0 & & & \sigma_n^2 \end{bmatrix}$

- If you plug your variance values into the covariance matrix the models are actually identical

Original model vs. Multivariate Gaussian

Original Gaussian model

- Probably used more often
- There is a need to manually create features to capture anomalies where x_1 and x_2 take unusual combinations of values
  - So **need to make extra features**
  - Might not be obvious what they should be
    - This is always a risk - where you're using your own expectation of a problem to "predict" future anomalies
    - Typically, the things that catch you out aren't going to be the things you though of
      - If you thought of them they'd probably be avoided in the first place
    - Obviously this is a bigger issue, and one which may or may not be relevant depending on your problem space
- Much **cheaper computationally**
- **Scales much better** to very large feature vectors
  - Even if n = 100 000 the original model works fine
- **Works well even with a small training set**
  - e.g. 50, 100
- Because of these factors it's used more often because it really represents a optimized but axis-symmetric specialization of the general model

Multivariate Gaussian model

- Used less frequently
- **Can capture feature correlation**
  - So no need to create extra values
- **Less computationally efficient**
  - Must compute inverse of matrix which is [n x n]
  - So lots of features is bad - makes this calculation very expensive
  - So if n = 100 000 not very good
- **Needs for m > n**
  - i.e. number of examples must be greater than number of features
  - If this is not true then we have a singular matrix (non-invertible)
  - So should be used only in m >> n
- If you find the matrix is non-invertible, could be for one of two main reasons
  - m < n
    - So use original simple model
  - Redundant features (i.e. linearly dependent)
    - i.e. two features that are the same
    - If this is the case you could use PCA or sanity check your data

16: Recommender Systems

Previous Next Index

Recommender systems - introduction

- Two motivations for talking about recommender systems
  - **Important application of ML systems**
    - Many technology companies find recommender systems to be absolutely key
    - Think about websites (amazon, Ebay, iTunes genius)
      - Try and recommend new content for you based on passed purchase
      - Substantial part of Amazon's revenue generation
    - Improvement in recommender system performance can bring in more income
    - Kind of a funny problem
      - In academic learning, recommender systems receives a small amount of attention
      - But in industry it's an absolutely crucial tool
  - Talk about the big ideas in machine learning
    - Not so much a technique, but an idea
    - As soon, features are really important
      - There's a big idea in machine learning that for some problems you can learn what a good set of features are
      - So not select those features but learn them
    - Recommender systems do this - try and identify the crucial and relevant features

Example - predict movie ratings

- You're a company who sells movies
  - You let users rate movies using a 1-5 star rating
    - To make the example nicer, allow 0-5 (makes math easier)
- You have five movies
- And you have four users
- Admittedly, business isn't going well, but you're optimistic about the future as a result of your truly outstanding (if limited) inventory

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at last	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?

- To introduce some notation
  - n_u - Number of users (called ?^{nu} occasionally as we can't subscript in superscript)
  - n_m - Number of movies
  - r(i, j) - 1 if user j has rated movie i (i.e. bitmap)
  - $y^{(i,j)}$ - rating given by user j to move i (defined only if r(i,j) = 1
- So for this example
  - $n_u = 4$

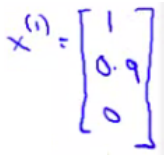
- $n_m = 5$
- Summary of scoring
  - Alice and Bob gave good ratings to rom coms, but low scores to action films
  - Carol and Dave game good ratings for action films but low ratings for rom coms
- We have the data given above
- The problem is as follows
  - Given r(i,j) and $y^{(i,j)}$ - go through and try and predict missing values (?s)
  - Come up with a learning algorithm that can fill in these missing values

Content based recommendation

- Using our example above, how do we predict?
  - For each movie we have a feature which measure degree to which each film is a
    - Romance (x_1)
    - Action (x_2)

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. karate	0	0	5	?	0	0.9

- If we have features like these, each film can be recommended by a feature vector
  - Add an extra feature which is $x_0 = 1$ for each film
  - So for each film we have a [3 x 1] vector, which for film number 1 ("Love at Last") would be



  - i.e. for our dataset we have
    - $\{x^1, x^2, x^3, x^4, x^5\}$
      - Where each of these is a [3x1] vector with an $x_0 = 1$ and then a romance and an action score
  - To be consistent with our notation, n is going to be the number of features NOT counting the x_0 term, so n = 2
- We could treat each rating for each user as a separate linear regression problem
  - For each user j we could learn a parameter vector
  - Then predict that user j will rate movie i with
    - $(\theta^j)^T x^i$ = stars
    - inner product of parameter vector and features
  - So, lets take user 1 (Alice) and see what she makes of the modern classic Cute Puppies of Love (CPOL)
    - We have some parameter vector (θ^1) associated with Alice
      - We'll explain later how we derived these values, but for now just take it that we have a vector

$$\theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}$$

- CPOL has a parameter vector (x^3) associated with it

$$x^{(3)} = \begin{bmatrix} 1 \\ 0.99 \\ 0 \end{bmatrix}$$

- Our prediction will be equal to
  - $(\theta^1)^T x^3 = (0 * 1) + (5 * 0.99) + (0 * 0)$
  - $= 4.95$
    - Which may seem like a reasonable value
- All we're doing here is applying a linear regression method for each user
  - So we determine a future rating based on their interest in romance and action based on previous films
- We should also add one final piece of notation
  - m^j, - Number of movies rated by the user (j)

How do we learn (θ^j)

- Create some parameters which give values as close as those seen in the data when applied

$$\min_{\theta^{(j)}} \frac{1}{2 m^{(j)}} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)} \right)^2$$

- Sum over all values of i (all movies the user has used) when r(i,j) = 1 (i.e. all the films that the user has rated)
- This is just like linear regression with least-squared error
- We can also add a regularization term to make our equation look as follows

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

  - The regularization term goes from k=1 through to m, so (θ^j) ends up being an n+1 feature vector
    - Don't regularize over the bias terms (0)
- If you do this you get a reasonable value
- We're rushing through this a bit, but it's just a linear regression problem

- To make this a little bit clearer you can get rid of the m^j term (it's just a constant so shouldn't make any difference to minimization)
  - So to learn (θ^j)

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

  - But for our recommender system we want to learn parameters for *all* users, so we add an

extra summation term to this which means we determine the minimum (θ^j) value for every user

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

- When you do this as a function of each (θ^j) parameter vector you get the parameters for each user
  - So this is our optimization objective -> $J(\theta^1, \ldots, \theta^{nu})$
- In order to do the minimization we have the following gradient descent

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} \text{ (for } k = 0)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} + \lambda\theta_k^{(j)}\right) \text{ (for } k \neq 0)$$

- Slightly different to our previous gradient descent implementations
  - k = 0 and k != 0 versions
  - We can define the middle term above as

$$\frac{\partial}{\partial \theta_k^{(j)}} J(\theta^{(1)},\ldots,\theta^{(n_u)}) = \left(\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} + \lambda\theta_k^{(j)}\right)$$

  - Difference from linear regression
    - No 1/m terms (got rid of the 1/m term)
    - Otherwise very similar
- This approach is called content-based approach because we assume we have features regarding the content which will help us identify things that make them appealing to a user
  - However, often such features are not available - next we discuss a non-contents based approach!

Collaborative filtering - overview

- The collaborative filtering algorithm has a very interesting property - does feature learning
  - i.e. it can learn for itself what features it needs to learn
- Recall our original data set above for our five films and four raters
  - Here we assume someone had calculated the "romance" and "action" amounts of the films
    - This can be very hard to do in reality
    - Often want more features than just two
- So - let's change the problem and pretend we have a data set where we don't know any of the features associated with the films

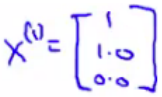
Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	?	?
Romance forever	5	?	?	0	?	?
Cute puppies of love	?	4	0	?	?	?
Nonstop car chases	0	0	5	4	?	?
Swords vs. karate	0	0	5	?	?	?

- Now let's make a different assumption
  - We've polled each user and found out how much each user likes
    - Romantic films
    - Action films
  - Which has generated the following parameter set

$$\theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \theta^{(2)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \theta^{(3)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}, \theta^{(4)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}$$

  - Alice and Bob like romance but hate action
  - Carol and Dave like action but hate romance

- If we can get these parameters from the users we can infer the missing values from our table
  - Lets look at "Love at Last"
    - Alice and Bob loved it
    - Carol and Dave hated it
  - We know from the feature vectors Alice and Bob love romantic films, while Carol and Dave hate them
    - Based on the factor Alice and Bob liked "Love at Last" and Carol and Dave hated it we may be able to (correctly) conclude that "Love at Last" is a romantic film
- This is a bit of a simplification in terms of the maths, but what we're really asking is
  - "What feature vector should x^1 be so that
    - $(\theta^1)^T x^1$ is about 5
    - $(\theta^2)^T x^1$ is about 5
    - $(\theta^3)^T x^1$ is about 0
    - $(\theta^4)^T x^1$ is about 0
  - From this we can guess that x^1 may be



  - Using that same approach we should then be able to determine the remaining feature vectors for the other films

Formalizing the collaborative filtering problem

- We can more formally describe the approach as follows
  - Given $(\theta^1, ..., \theta^{nu})$ (i.e. given the parameter vectors for each users' preferences)
  - We must minimize an optimization function which tries to identify the best parameter vector associated with a film

$$\min_{x^{(i)}} \frac{1}{2} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2$$

    - So we're summing over all the indices j for where we have data for movie i
    - We're minimizing this squared error
  - Like before, the above equation gives us a way to learn the features for one film
    - We want to learn all the features for *all* the films - so we need an additional summation term

How does this work with the previous recommendation system

- Content based recommendation systems
  - Saw that if we have a set of features for movie rating you can learn a user's preferences
- Now
  - If you have your users preferences you can therefore determine a film's features
- This is a bit of a chicken & egg problem
- What you can do is
  - Randomly guess values for θ
  - Then use collaborative filtering to generate x
  - Then use content based recommendation to improve θ
  - Use that to improve x
  - And so on
- This actually works
  - Causes your algorithm to converge on a reasonable set of parameters
  - This is collaborative filtering
- We call it collaborative filtering because in this example the users are collaborating together to help the algorithm learn better features and help the system and the other users

Collaborative filtering Algorithm

- Here we combine the ideas from before to build a collaborative filtering algorithm
- Our starting point is as follows
  - If we're given the film's features we can use that to work out the users' preference

$$\text{Given } x^{(1)}, \ldots, x^{(n_m)}, \text{ estimate } \theta^{(1)}, \ldots, \theta^{(n_u)}:$$

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

  - If we're given the users' preferences we can use them to work out the film's features

$$\text{Given } \theta^{(1)}, \ldots, \theta^{(n_u)}, \text{ estimate } x^{(1)}, \ldots, x^{(n_m)}:$$

$$\min_{x^{(1)},\ldots,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

- One thing you could do is
  - Randomly initialize parameter
  - Go back and forward
- But there's a more efficient algorithm which can solve θ and x simultaneously
  - Define a new optimization objective which is a function of x and θ

Minimizing $x^{(1)}, \ldots, x^{(n_m)}$ and $\theta^{(1)}, \ldots, \theta^{(n_u)}$ simultaneously:

$$J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$$\min_{\substack{x^{(1)}, \ldots, x^{(n_m)} \\ \theta^{(1)}, \ldots, \theta^{(n_u)}}} J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$$

- Understanding this optimization objective
  - The squared error term is the same as the squared error term in the two individual objectives above

$$\sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2$$

    - So it's summing over every movie rated by every users
    - Note the ":" means, "for which"
      - Sum over all pairs (i,j) for which r(i,j) is equal to 1
  - The regularization terms
    - Are simply added to the end from the original two optimization functions
- This newly defined function has the property that
  - If you held x constant and only solved θ then you solve the, "Given x, solve θ" objective above
  - Similarly, if you held θ constant you could solve x
- In order to come up with just one optimization function we treat this function as a function of both film features x and user parameters θ
  - Only difference between this in the back-and-forward approach is that we minimize with respect to both x and θ simultaneously
- When we're learning the features this way
  - Previously had a convention that we have an $x_0 = 1$ term
  - When we're using this kind of approach we have no x_0,
    - So now our vectors (both x and θ) are n-dimensional (not n+1)
  - We do this because we are now learning all the features so if the system needs a feature always = 1 then the algorithm can learn one

Algorithm Structure

- **1)** Initialize θ^1, ..., θ^{nu} and x^1, ..., x^{nm} to small random values
  - A bit like neural networks - initialize all parameters to small random numbers
- **2)** Minimize cost function ($J(x^1, \ldots, x^{nm}, \theta^1, \ldots, \theta^{nu})$ using gradient descent
  - We find that the update rules look like this

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

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

  - Where the top term is the partial derivative of the cost function with respect to x_k^i while the bottom is the partial derivative of the cost function with respect to θ_k^i
  - So here we regularize EVERY parameters (no longer x_0 parameter) so no special case update rule
- **3)** Having minimized the values, given a user (user j) with parameters θ and movie (movie i) with learned features x, we predict a start rating of $(\theta^j)^T x^i$
  - This is the collaborative filtering algorithm, which should give pretty good predictions for how users like new movies

Vectorization: Low rank matrix factorization

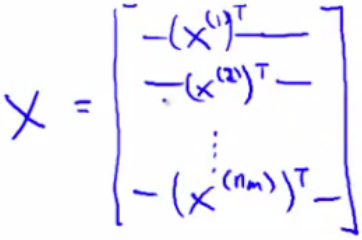
- Having looked at collaborative filtering algorithm, how can we improve this?
  - Given one product, can we determine other relevant products?
- We start by working out another way of writing out our predictions
  - So take all ratings by all users in our example above and group into a matrix Y

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

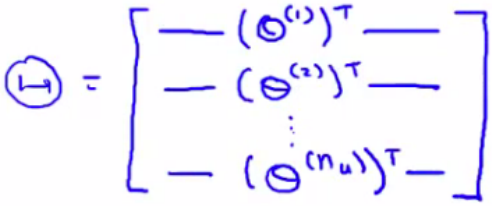
    - 5 movies
    - 4 users
    - Get a [5 x 4] matrix
  - Given [Y] there's another way of writing out all the predicted ratings

$$\begin{bmatrix} (\theta^{(1)})^T(x^{(1)}) & (\theta^{(2)})^T(x^{(1)}) & \dots & (\theta^{(n_u)})^T(x^{(1)}) \\ (\theta^{(1)})^T(x^{(2)}) & (\theta^{(2)})^T(x^{(2)}) & \dots & (\theta^{(n_u)})^T(x^{(2)}) \\ \vdots & \vdots & \vdots & \vdots \\ (\theta^{(1)})^T(x^{(n_m)}) & (\theta^{(2)})^T(x^{(n_m)}) & \dots & (\theta^{(n_u)})^T(x^{(n_m)}) \end{bmatrix}$$

    - With this matrix of predictive ratings
    - We determine the (i,j) entry for EVERY movie
- We can define another matrix X
  - Just like matrix we had for linear regression
  - Take all the features for each movie and stack them in rows



    - Think of each movie as one example
  - Also define a matrix Θ



    - Take each per user parameter vector and stack in rows
- Given our new matrices X and θ
  - We can have a vectorized way of computing the prediction range matrix by doing $X * \theta^T$
- We can given this algorithm another name - **low rank matrix factorization**
  - This comes from the property that the $X * \theta^T$ calculation has a property in linear algebra that we create a **low rank** matrix
    - Don't worry about what a low rank matrix is

Recommending new movies to a user

- Finally, having run the collaborative filtering algorithm, we can use the learned features to find related films
  - When you learn a set of features you don't know what the features will be - lets you identify the features which define a film
  - Say we learn the following features
    - x_1 - romance
    - x_2 - action
    - x_3 - comedy
    - x_4 - ...
  - So we have n features all together
  - After you've learned features it's often very hard to come in and apply a human understandable metric to what those features are
    - Usually learn features which are very meaning full for understanding what users like
- Say you have movie i
  - Find movies j which is similar to i, which you can recommend
  - Our features allow a good way to measure movie similarity
  - If we have two movies x^i and x^j
    - We want to minimize $||x^i - x^j||$
      - i.e. the distance between those two movies
  - Provides a good indicator of how similar two films are in the sense of user perception
    - NB - Maybe ONLY in terms of user perception

Implementation detail: Mean Normalization

- Here we have one final implementation detail - make algorithm work a bit better
- To show why we might need mean normalization let's consider an example where there's a user who hasn't rated *any* movies

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love at last	5	5	0	0	?
Romance forever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Swords vs. karate	0	0	5	?	?

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}$$

  - Lets see what the algorithm does for this user
    - Say n = 2
    - We now have to learn θ^5 (which is an n-dimensional vector)
  - Looking in the first term of the optimization objective
    - There are *no* films for which r(i,j) = 1
    - So this term places no role in determining θ^5
    - So we're just minimizing the final regularization term

$$\min_{\substack{x^{(1)},\ldots,x^{(n_m)} \\ \theta^{(1)},\ldots,\theta^{(n_u)}}} \frac{1}{2}\sum_{(i,j):r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

This term is irrelevant

We're only minimizing this

Which can for our single example be simplified to this $\frac{\lambda}{2}\left[(\theta_1^{(5)})^2 + (\theta_2^{(5)})^2\right]$

- Of course, if the goal is to minimize this term then

$\theta^{(5)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

  - Why - If there's no data to pull the values away from 0 this gives the min value
- So this means we predict ANY movie to be zero
  - Presumably Eve doesn't hate all movies...
  - So if we're doing this we can't recommend any movies to her either
- Mean normalization should let us fix this problem

How does mean normalization work?

- Group all our ratings into matrix Y as before
  - We now have a column of ?s which corresponds to Eves rating

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}$$

  - Now we compute the average rating each movie obtained and stored in an n_m - dimensional column vector

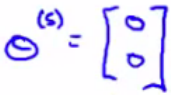
$$\mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$$

  - If we look at all the movie ratings in [Y] we can subtract off the mean rating

$$Y = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

    - Means we normalize each film to have an average rating of 0
  - Now, we take the new set of ratings and use it with the collaborative filtering algorithm
    - Learn θ^j and x^i from the mean normalized ratings
- For our prediction of user j on movie i, predict
  - $(\theta^j)^T x^i + \mu_i$
    - Where these vectors are the mean normalized values

        - We have to add μ because we removed it from our θ values
    - So for user 5 the same argument applies, so



    - So on any movie i we're going to predict
        - $(\theta^5)^T x^i + \mu_i$
            - Where $(\theta^5)^T x^i$ = to 0 (still)
            - But we then add the mean (μ_i) which means Eve has an average rating assigned to each movie for here
- This makes sense
    - If Eve hasn't rated any films, predict the average rating of the films based on everyone
        - This is the best we can do
- As an aside - we spoke here about mean normalization for users with no ratings
    - If you have some movies with no ratings you can also play with versions of the algorithm where you normalize the columns
    - BUT this is probably less relevant - probably shouldn't recommend an unrated movie
- To summarize, this shows how you do mean normalization preprocessing to allow your system to deal with users who have not yet made any ratings
    - Means we recommend the user we know little about the best average rated products

17: Large Scale Machine Learning

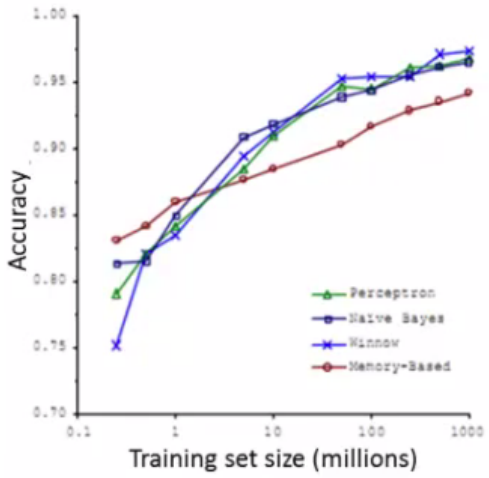
Previous Next Index

Learning with large datasets

- This set of notes look at large scale machine learning - how do we deal with big datasets?
- If you look back at 5-10 year history of machine learning, ML is much better now because we have much more data
  - However, with this increase in data comes great responsibility? No, comes a much more significant computational cost
  - New and exciting problems are emerging that need to be dealt with on both the algorithmic and architectural level

Why large datasets?

- One of best ways to get high performance is take a low bias algorithm and train it on a lot of data
  - e.g. Classification between confusable words
  - We saw that so long as you feed an algorithm lots of data they all perform pretty similarly



  - So it's good to learn with large datasets
- But learning with large datasets comes with its own computational problems

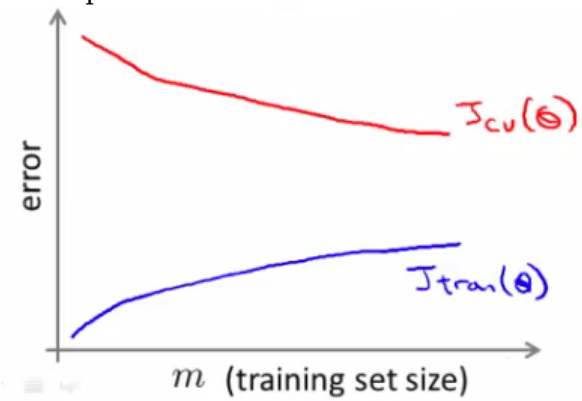
Learning with large datasets

- For example, say we have a data set where m = 100, 000, 000
  - This is pretty realistic for many datasets
    - Census data
    - Website traffic data
  - How do we train a logistic regression model on such a big system?

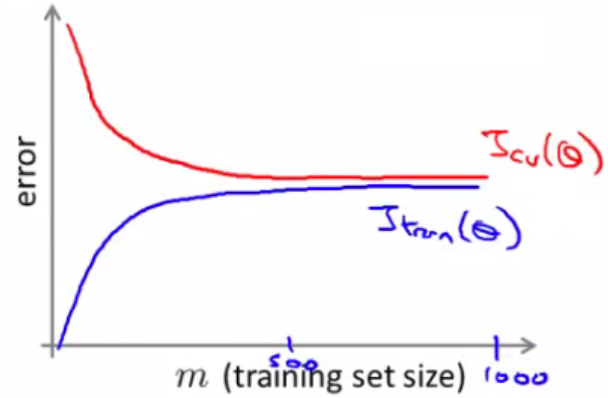
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

    - So you have to sum over 100,000,000 terms per step of gradient descent

- Because of the computational cost of this massive summation, we'll look at more efficient ways around this
  - - Either using a different approach
  - - Optimizing to avoid the summation
- First thing to do is ask if we can train on 1000 examples instead of 100 000 000
  - Randomly pick a small selection
  - Can you develop a system which performs as well?
    - Sometimes yes - if this is the case you can avoid a lot of the headaches associated with big data
- To see if taking a smaller sample works, you can sanity check by plotting error vs. training set size
  - If our plot looked like this



  - Looks like a **high variance problem**
    - More examples should improve performance
  - If plot looked like this



  - This looks like a **high bias problem**
    - More examples may not actually help - save a lot of time and effort if we know this *before hand*
    - One natural thing to do here might be to;
      - Add extra features
      - Add extra hidden units (if using neural networks)

Stochastic Gradient Descent

- For many learning algorithms, we derived them by coming up with an optimization objective (cost function) and using an algorithm to minimize that cost

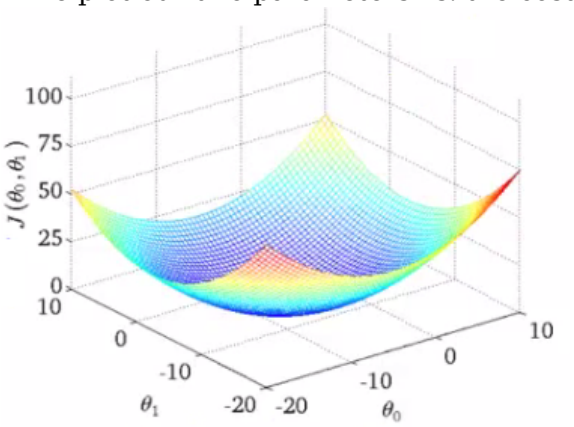
function
  - When you have a large dataset, gradient descent becomes very expensive
  - So here we'll define a different way to optimize for large data sets which will allow us to scale the algorithms
- Suppose you're training a linear regression model with gradient descent
  - **Hypothesis**

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

  - **Cost function**

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

  - If we plot our two parameters vs. the cost function we get something like this

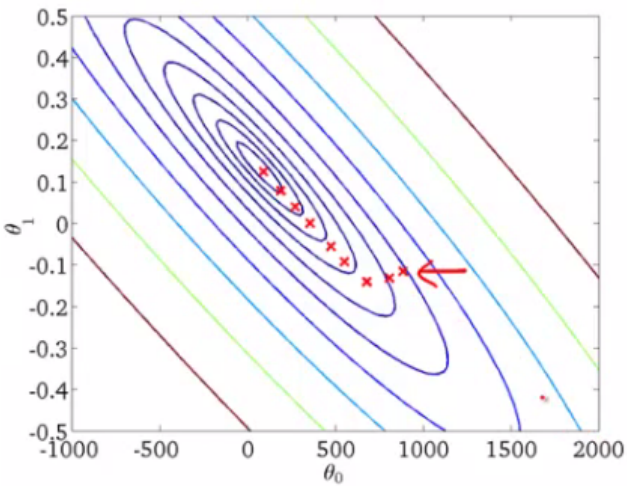


  - Looks like this bowl shape surface plot
- **Quick reminder - how does gradient descent work?**

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

$$(\text{for every } j = 0, \ldots, n) \ \ \}$$

  - In the inner loop we repeatedly update the parameters θ
- We will use linear regression for our algorithmic example here when talking about **stochastic gradient descent**, although the ideas apply to other algorithms too, such as
  - Logistic regression
  - Neural networks
- Below we have a contour plot for gradient descent showing iteration to a global minimum



- As mentioned, if *m* is large gradient descent can be very expensive
- Although so far we just referred to it as gradient descent, this kind of gradient descent is called **batch gradient descent**
  - This just means we look at all the examples at the same time
- Batch gradient descent is not great for huge datasets
  - If you have 300,000,000 records you need to read in all the records into memory from disk because you can't store them all in memory
    - By reading all the records, you can move one step (iteration) through the algorithm
  - Then repeat for EVERY step
    - This means it take a LONG time to converge
    - Especially because disk I/O is typically a system bottleneck anyway, and this will inevitably require a *huge* number of reads
- What we're going to do here is come up with a different algorithm which only needs to look at single example at a time

Stochastic gradient descent

- Define our cost function slightly differently, as

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

  - So the function represents the cost of θ with respect to a specific example (x^i, y^i)
    - And we calculate this value as one half times the squared error on that example
  - Measures how well the hypothesis works on a single example
- The overall cost function can now be re-written in the following form;

$$J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

  - This is equivalent to the batch gradient descent cost function
- With this slightly modified (but equivalent) view of linear regression we can write out how stochastic gradient descent works

- **1) - Randomly shuffle**

Randomly shuffle (reorder) training examples

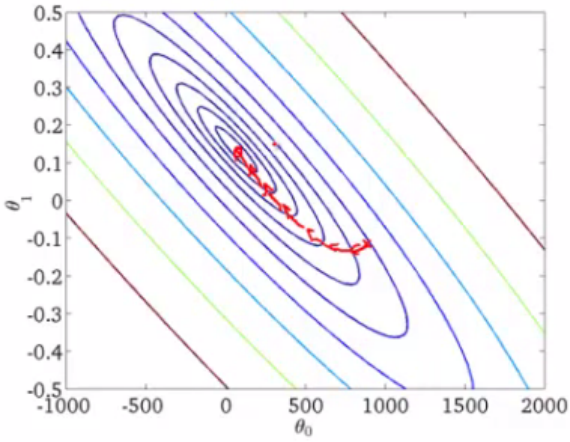
- **2) - Algorithm body**

Repeat {
  for $i := 1, \ldots, m$ {
    $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
    (for every $j = 0, \ldots, n$)
  }
}

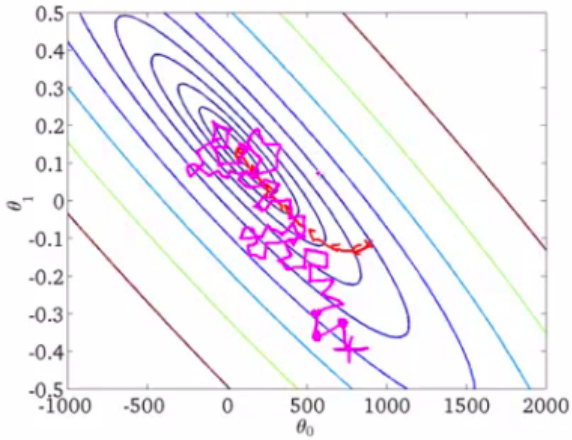
- *So what's going on here?*
  - The term $h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
  - Is the same as that found in the summation for batch gradient descent
  - It's possible to show that this term is equal to the partial derivative with respect to the parameter θ_j of the cost(θ, (x^i, y^i))

$$\frac{\partial}{\partial \theta_j} cost(\theta, (x^{(i)}, y^{(i)}))$$

- What stochastic gradient descent algorithm is doing is scanning through each example
  - The inner for loop does something like this...
    - Looking at example 1, take a step with respect to the cost of just the 1st training example
      - Having done this, we go on to the second training example
    - Now take a second step in parameter space to try and fit the second training example better
      - Now move onto the third training example
    - And so on...
    - Until it gets to the end of the data
  - We may now repeat this who procedure and take multiple passes over the data
- The **randomly shuffling** at the start means we ensure the data is in a random order so we don't bias the movement
  - Randomization should speed up convergence a little bit
- Although stochastic gradient descent is a lot like batch gradient descent, rather than waiting to sum up the gradient terms over all *m* examples, we take just one example and make progress in improving the parameters already
  - Means we update the parameters on EVERY step through data, instead of at the end of each loop through all the data
- What does the algorithm do to the parameters?
  - As we saw, batch gradient descent does something like this to get to a global minimum



- With stochastic gradient descent every iteration is much faster, but every iteration is flitting a single example



  - What you find is that you "generally" move in the direction of the global minimum, but not always
  - Never actually converges like batch gradient descent does, but ends up wandering around some region close to the global minimum
    - In practice, this isn't a problem - as long as you're close to the minimum that's probably OK
- One final implementation note
  - May need to loop over the entire dataset 1-10 times
  - If you have a truly massive dataset it's possible that by the time you've taken a single pass through the dataset you may already have a perfectly good hypothesis
    - In which case the inner loop might only need to happen 1 if m is very very large
- If we contrast this to batch gradient descent
  - We have to make k passes through the entire dataset, where k is the number of steps needed to move through the data

Mini Batch Gradient Descent

- **Mini-batch gradient descent** is an additional approach which can work even faster than stochastic gradient descent
- To summarize our approaches so far

    - Batch gradient descent: Use all *m* examples in each iteration
    - Stochastic gradient descent: Use 1 example in each iteration
    - Mini-batch gradient descent: Use *b* examples in each iteration
        - *b* = mini-batch size
- So just like batch, except we use tiny batches
    - Typical range for *b* = 2-100 (10 maybe)
- For example
    - *b* = 10
    - Get 10 examples from training set
    - Perform gradient descent update using the ten examples

Mini-batch algorithm

Say $b = 10, m = 1000.$

Repeat {

for $i = 1, 11, 21, 31, \ldots, 991$ {

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every $j = 0, \ldots, n$) } }

- We for-loop through *b*-size batches of *m*
- Compared to batch gradient descent this allows us to get through data in a much more efficient way
    - After just *b* examples we begin to improve our parameters
    - Don't have to update parameters after *every* example, and don't have to wait until you cycled through all the data

Mini-batch gradient descent vs. stochastic gradient descent

- Why should we use mini-batch?
    - Allows you to have a vectorized implementation
    - Means implementation is much more efficient
    - Can partially parallelize your computation (i.e. do 10 at once)
- A disadvantage of mini-batch gradient descent is the optimization of the parameter *b*
    - But this is often worth it!
- To be honest, stochastic gradient descent and batch gradient descent are just specific forms of batch-gradient descent
    - For mini-batch gradient descent, b is somewhere in between 1 and m and you can try to optimize for it!

Stochastic gradient descent convergence

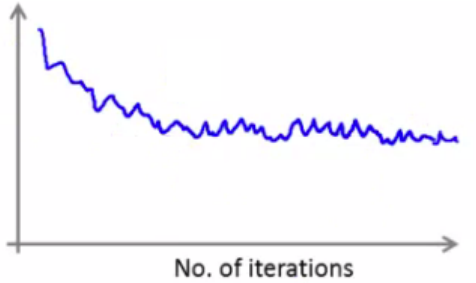
- We now know about stochastic gradient descent
  - But how do you know when it's done!?
  - How do you tune learning rate alpha (α)?

Checking for convergence

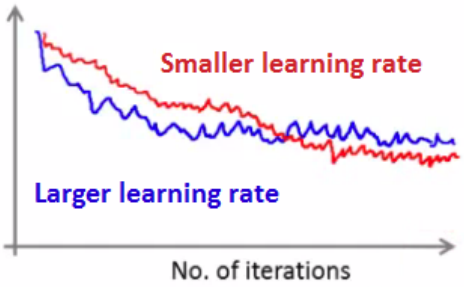
- With batch gradient descent, we could plot cost function vs number of iterations
  - Should decrease on every iteration
  - This works when the training set size was small because we could sum over all examples
    - Doesn't work when you have a massive dataset
  - With stochastic gradient descent
    - We don't want to have to pause the algorithm periodically to do a summation over all data
    - Moreover, the whole point of stochastic gradient descent is to *avoid* those whole-data summations
- For stochastic gradient descent, we have to do something different
  - Take cost function definition

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

    - One half the squared error on a single example
  - While the algorithm is looking at the example (x^i, y^i), but *before* it has updated θ we can compute the cost of the example (cost(θ, (x^i, y^i))
    - i.e. we compute how well the hypothesis is working on the training example
      - Need to do this before we update θ because if we did it after θ was updated the algorithm would be performing a bit better (because we'd have just used (x^i, y^i) to improve θ)
  - To check for the convergence, every 1000 iterations we can plot the costs averaged over the last 1000 examples
    - Gives a running estimate of how well we've done on the last 1000 estimates
    - By looking at the plots we should be able to check convergence is happening
- What do these plots look like
  - In general
    - Might be a bit noisy (1000 examples isn't that much)
  - If you get a figure like this

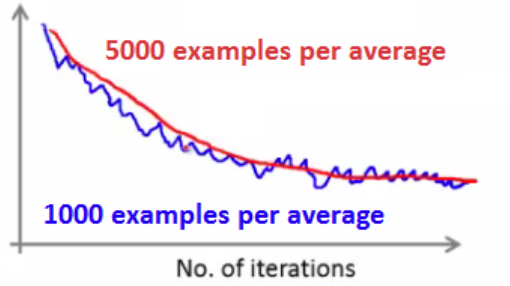


    - That's a pretty decent run
    - Algorithm may have convergence
  - If you use a smaller learning rate you may get an even better final solution



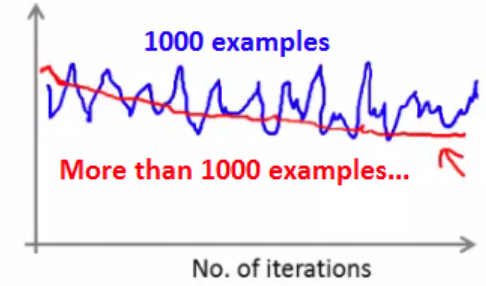
- This is because the parameter oscillate around the global minimum
- A smaller learning rate means smaller oscillations

- If you average over 1000 examples and 5000 examples you may get a smoother curve



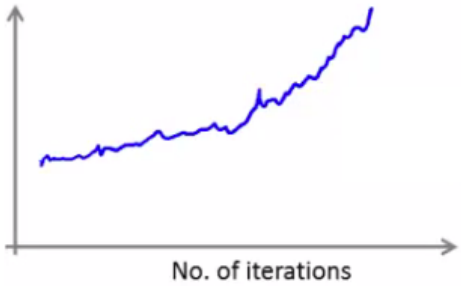
- This disadvantage of a larger average means you get less frequent feedback

- Sometimes you may get a plot that looks like this



- Looks like cost is not decreasing at all
- But if you then increase to averaging over a larger number of examples you do see this general trend
  - Means the blue line was too noisy, and that noise is ironed out by taking a greater number of entires per average
- Of course, it may not decrease, even with a large number

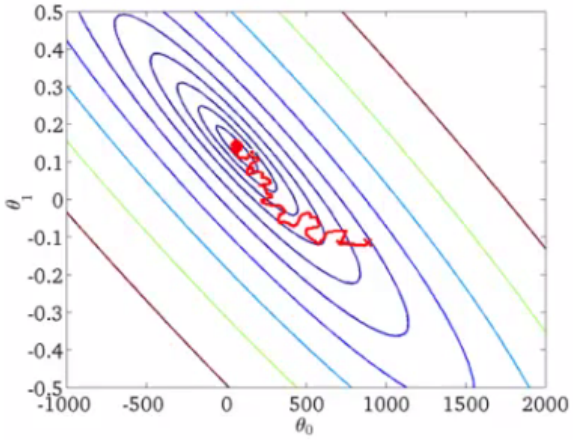
- If you see a curve the looks like its increasing then the algorithm may be displaying divergence



- Should use a smaller learning rate

Learning rate

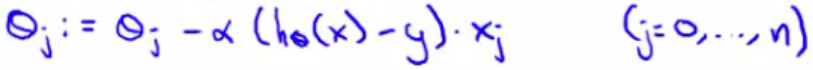
- We saw that with stochastic gradient descent we get this wandering around the minimum
  - In most implementations the learning rate is held constant
- However, if you want to converge to a minimum you can slowly decrease the learning rate over time
  - A classic way of doing this is to calculate α as follows
    α = const1/(iterationNumber + const2)
  - Which means you're guaranteed to converge somewhere
    - You also need to determine const1 and const2
  - BUT if you tune the parameters well, you can get something like this



Online learning

- New setting
  - Allows us to model problems where you have a continuous stream of data you want an algorithm to learn from
  - Similar idea of stochastic gradient descent, in that you do slow updates
  - Web companies use various types of online learning algorithms to learn from traffic
    - Can (for example) learn about user preferences and hence optimize your website
- Example - Shipping service
  - Users come and tell you origin and destination
  - You offer to ship the package for some amount of money ($10 - $50)
  - Based on the price you offer, sometimes the user uses your service (y = 1), sometimes they don't (y = 0)
  - Build an algorithm to optimize what price we offer to the users
    - Capture
      - Information about user
      - Origin and destination
    - Work out
      - What the probability of a user selecting the service is
    - We want to optimize the price
  - To model this probability we have something like
    - p(y = 1|x; θ)
      - Probability that y =1, given x, parameterized by θ
    - Build this model with something like
      - Logistic regression
      - Neural network
  - If you have a website that runs continuously an online learning algorithm would do something like this

        - User comes - is represented as an (x,y) pair where
            - x - feature vector including price we offer, origin, destination
            - y - if they chose to use our service or not
        - The algorithm updates θ using just the (x,y) pair



        - So we basically update all the θ parameters every time we get some new data
    - While in previous examples we might have described the data example as (x^i, y^i) for an online learning problem we discard this idea of a data "set" - instead we have a continuous stream of data so indexing is largely irrelevant as you're not storing the data (although presumably you could store it)
- If you have a major website where you have a massive stream of data then this kind of algorithm is pretty reasonable
    - You're free of the need to deal with all your training data
- If you had a small number of users you could save their data and then run a normal algorithm on a dataset
- An online algorithm can adapt to changing user preferences
    - So over time users may become more price sensitive
    - The algorithm adapts and learns to this
    - So your system is dynamic

Another example - product search

- Run an online store that sells cellphones
    - You have a UI where the user can type in a query like, "Android phone 1080p camera"
    - We want to offer the user 10 phones per query
- How do we do this
    - For each phone and given a specific user query, we create a feature vector (x) which has data like features of the phone, how many words in the user query match the name of the phone, how many words in user query match description of phone
        - Basically how well does the phone match the user query
    - We want to estimate the probability of a user selecting a phone
    - So define
        - y = 1 if a user clicks on a link
        - y = 0 otherwise
    - So we want to learn
        - p(y = 1|x ; θ) <- this is the problem of learning the predicted **click through rate** (CTR)
    - If you can estimate the CTR for any phone we can use this to show the highest probability phones first
    - If we display 10 phones per search, it means for each search we generate 10 training examples of data
        - i.e. user can click through one or more, or none of them, which defines how well the prediction performed
- Other things you can do
    - Special offers to show the user
    - Show news articles - learn what users like
    - Product recommendation
- These problems could have been formulated using standard techniques, but they are the kinds of problems where you have so much data that this is a better way to do things

Map reduce and data parallelism

- Previously spoke about stochastic gradient descent and other algorithms
  - These could be run on one machine
  - Some problems are just too big for one computer
  - Talk here about a different approach called Map Reduce
- Map reduce example
  - We want to do batch gradient descent

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

    - Assume m = 400
      - Normally m would be more like 400 000 000
      - If m is large this is really expensive
  - Split training sets into different subsets
    - So split training set into 4 pieces
  - **Machine 1** : use (x^1, y^1), ..., (x^{100}, y^{100})
    - Uses first quarter of training set
    - Just does the summation for the first 100

$$temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Machine 2: Use $(x^{(101)}, y^{(101)}), \ldots, (x^{(200)}, y^{(200)})$.

$$temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Machine 3: Use $(x^{(201)}, y^{(201)}), \ldots, (x^{(300)}, y^{(300)})$.

$$temp_j^{(3)} = \sum_{i=201}^{300} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Machine 4: Use $(x^{(301)}, y^{(301)}), \ldots, (x^{(400)}, y^{(400)})$.

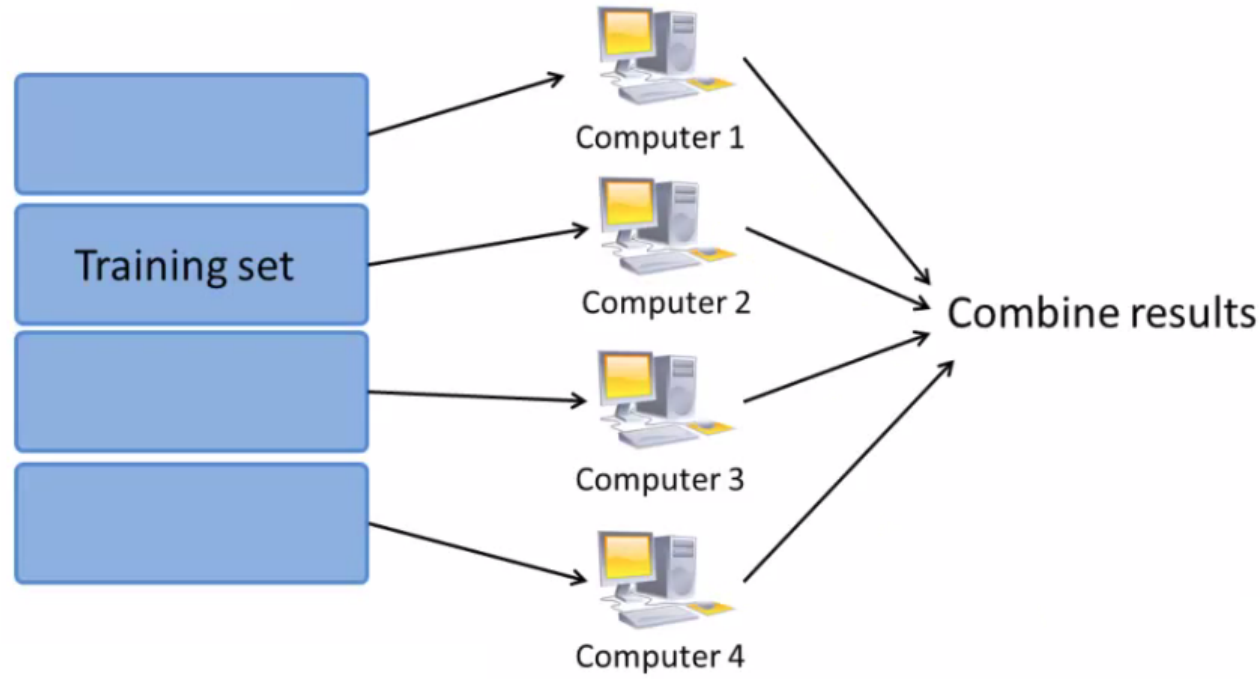
$$temp_j^{(4)} = \sum_{i=301}^{400} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

  - So now we have these four temp values, and each machine does 1/4 of the work
  - Once we've got our temp variables
    - Send to to a centralized master server
    - Put them back together
    - Update θ using

$$\theta_j := \theta_j - \alpha \frac{1}{400} \left(temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)} \right)$$

$$(j = 0, \ldots, n)$$

      - This equation is doing the same as our original batch gradient descent algorithm
- More generally map reduce uses the following scheme (e.g. where you split into 4)



- The bulk of the work in gradient descent is the summation
  - Now, because each of the computers does a quarter of the work at the same time, you get a 4x speedup
  - Of course, in practice, because of network latency, combining results, it's slightly less than 4x, but still good!
- Important thing to ask is
  - "Can algorithm be expressed as computing sums of functions of the training set?"
    - Many algorithms can!
- Another example
  - Using an advanced optimization algorithm with logistic regression
  - $$J_{train}(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) - (1-y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$
    - Need to calculate cost function - see we sum over training set
    - So split training set into x machines, have x machines compute the sum of the value over 1/xth of the data
  - $$\frac{\partial}{\partial \theta_j} J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$
    - These terms are also a sum over the training set
    - So use same approach
- So with these results send temps to central server to deal with combining everything

- More broadly, by taking algorithms which compute sums you can scale them to very large datasets through parallelization
  - Parallelization can come from
    - Multiple machines
    - Multiple CPUs

        - Multiple cores in each CPU
    - So even on a single compute can implement parallelization
- The advantage of thinking about Map Reduce here is because you don't need to worry about network issues
    - It's all internal to the same machine
- Finally caveat/thought

    - Depending on implementation detail, certain numerical linear algebra libraries can automatically parallelize your calculations across multiple cores
    - So, if this is the case and you have a good vectorization implementation you can not worry about local Parallelization and the local libraries sort optimization out for you

Hadoop

- Hadoop is a good open source Map Reduce implementation
- Represents a top-level Apache project develop by a global community of developers
    - Large developer community all over the world
- Written in Java
- Yahoo has been the biggest contributor
    - Pushed a lot early on
- Support now from Cloudera

Interview with Cloudera CEO Mike Olson (2010)

- Seeing a change in big data industry (Twitter, Facebook etc) - relational databases can't scale to the volumes of data being generated
    - ***Q: Where the tech came from?***
        - Early 2000s - Google had too much data to process (and index)
            - Designed and built Map Reduce
                - Buy and mount a load of rack servers
                - Spread the data out among these servers (with some duplication)
                - Now you've stored the data and you have this processing infrastructure spread among the data
                    - Use local CPU to look at local data
                    - Massive data parallelism
                - Published as a paper in 2004
                    - At the time wasn't obvious why it was necessary - didn't support queries, transactions, SQL etc
            - When data was at "human" scale relational databases centralized around a single server was fine
            - But now we're scaling by Moore's law in two ways
                - More data
                - Cheaper to store
- **Q: How do people approach the issues in big data?**

    - People still use relational databases - great if you have predictable queries over structured data
        - Data warehousing still used - long term market
    - But the data people want to work with is becoming more complex and bigger
        - Free text, unstructured data doesn't fit will into tables
        - Do sentiment analysis in SQL isn't really that good
        - So to do new kinds of processing need a new type of architecture

    - Hadoop lets you do *data* processing - not transactional processing - on the big scale
    - Increasingly things like NoSQL is being used
    - Data centers are starting to chose technology which is aimed at a specific problem, rather than trying to shoehorn problems into an ER issue
    - Open source technologies are taking over for developer facing infrastructures and platforms
- **Q: What is Hadoop?**
    - Open source implementation of Map reduce (Apache software)
    - Yahoo invested a lot early on - developed a lot the early progress
    - Is two things
        - **HDFS**
            - Disk on ever server
            - Software infrastructure to spread data
        - **Map reduce**
            - Lets you push code down to the data in parallel
    - As size increases you can just add more servers to scale up
- **Q: What is memcached?**
    - Ubiquitous invisible infrastructure that makes the web run
    - You go to a website, see data being delivered out of a MySQL database
        - BUT, when infrastructure needs to scale querying a disk EVERY time is too much
    - Memcache is a memory layer between disk and web server
        - Cache reads
        - Push writes through incrementally
        - Is the glue that connects a website with a disk-backend
    - Northscale is commercializing this technology
    - New data delivery infrastructure which has pretty wide adoption
- **Q: What is Django?**
    - Open source tool/language
- **Q: What are some of the tool sets being used in data management? What is MySQL drizzle?**
    - Drizzle is a re-implementation of MySQL
        - Team developing Drizzle feels they learned a lot of lessons when building MySQL
        - More modern architecture better targeted at web applications
    - NoSQL
        - Distributed hash tables
        - Idea is instead of a SQL query and fetching a record, go look up something from a store by name
            - Go pull a record by name from a store
        - So now systems being developed to support that like
            - MongoDB
            - CouchDB
    - Hadoop companion projects
        - Hive
            - Lets you use SQL to talk to a Hadoop cluster
        - HBase
            - Sits on top of HDFS
            - Gives you key-value storage on top of HDFS - provides abstraction from distributed system
    - Good time to be working in big data
        - Easy to set up small cluster through cloud system
        - Get a virtual cluster through Rackspace or Cloudera

18: Application Example OCR

Previous Next Index

Problem description and pipeline

- Case study focused around photo OCR
- Three reasons to do this
  - 1) Look at how a **complex system** can be put together
  - 2) The idea of a machine learning **pipeline**
    - What to do next
    - How to do it
  - 3) Some more interesting ideas
    - Applying machine learning to tangible problems
    - **Artificial data synthesis**

What is the photo OCR problem?

- Photo OCR = photo optical character recognition
  - With growth of digital photography, lots of digital pictures
  - One idea which has interested many people is getting computers to understand those photos
  - The photo OCR problem is getting computers to read text in an image
    - Possible applications for this would include
      - Make searching easier (e.g. searching for photos based on words in them)
      - Car navigation
- OCR of documents is a comparatively easy problem
  - From photos it's really hard

OCR pipeline

- 1) Look through image and find text
- 2) Do character segmentation
- 3) Do character classification
- 4) *Optional* some may do spell check after this too
  - We're not focussing on such systems though

- **Pipelines** are common in machine learning
  - Separate modules which may each be a machine learning component or data processing component
- If you're designing a machine learning system, pipeline design is one of the most important questions
  - Performance of pipeline and each module often has a big impact on the overall performance a problem
  - You would often have different engineers working on each module
    - Offers a natural way to divide up the workload

Sliding window image analysis

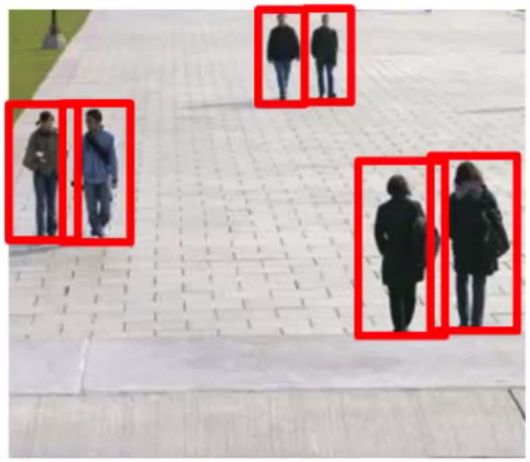
- How do the individual models work?
- Here focus on a sliding windows classifier

- As mentioned, stage 1 is **text detection**
  - Unusual problem in computer vision - different rectangles (which surround text) may have different aspect ratios (aspect ratio being height : width)
    - Text may be short (few words) or long (many words)
    - Tall or short font
    - Text might be straight on
    - Slanted

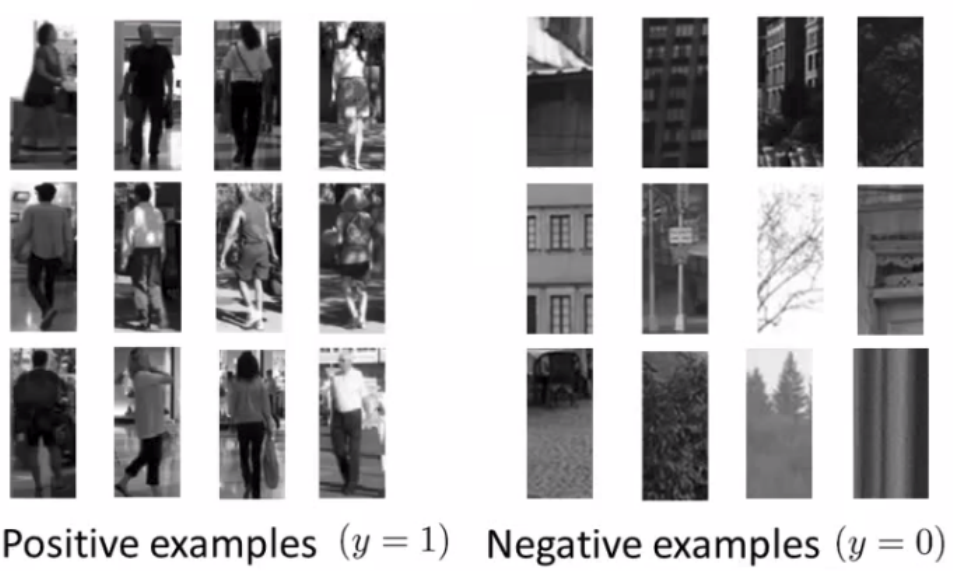
  - Let's start with a simpler example

Pedestrian detection

- Want to take an image and find pedestrians in the image

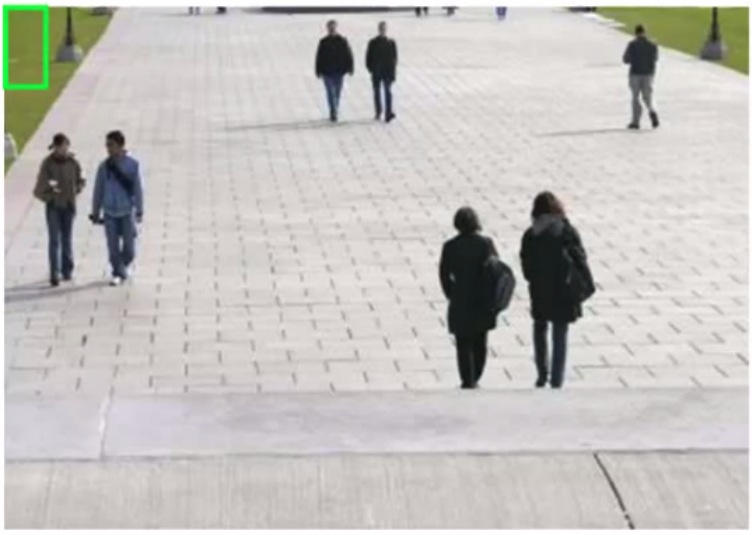


- This is a slightly simpler problem because the aspect ration remains pretty constant
- Building our detection system
  - Have 82 x 36 aspect ratio
    - This is a typical aspect ratio for a standing human
  - Collect training set of positive and negative examples



- Could have 1000 - 10 000 training examples
- Train a neural network to take an image and classify that image as pedestrian or not
  - Gives you a way to train your system

- Now we have a new image - how do we find pedestrians in it?
  - Start by taking a rectangular 82 x 36 patch in the image



    - Run patch through classifier - hopefully in this example it will return y = 0
  - Next slide the rectangle over to the right a little bit and re-run
    - Then slide again
    - The amount you slide each rectangle over is a parameter called the step-size or stride
      - Could use 1 pixel
        - Best, but computationally expensive
      - More commonly 5-8 pixels used
    - So, keep stepping rectangle along all the way to the right
      - Eventually get to the end
    - Then move back to the left hand side but step down a bit too
    - Repeat until you've covered the whole image
  - Now, we initially started with quite a small rectangle
    - So now we can take a larger image patch (of the same aspect ratio)
    - Each time we process the image patch, we're resizing the larger patch to a smaller

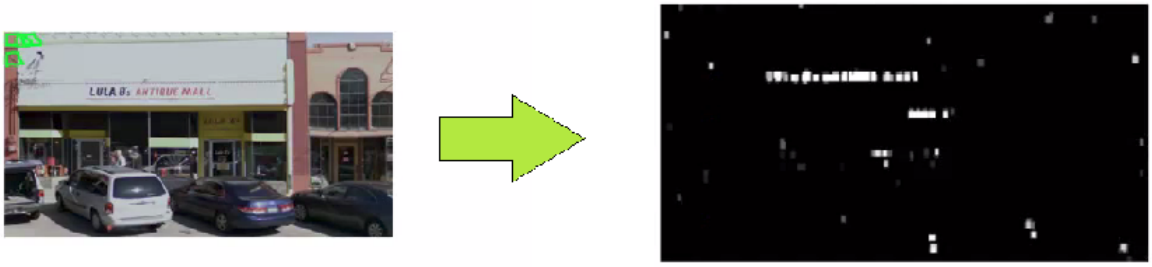
image, then running that smaller image through the classifier
- Hopefully, by changing the patch size and rastering repeatedly across the image, you eventually recognize all the pedestrians in the picture

Text detection example

- Like pedestrian detection, we generate a labeled training set with
  - Positive examples (some kind of text)
  - Negative examples (not text)

Positive examples $(y = 1)$ Negative examples $(y = 0)$

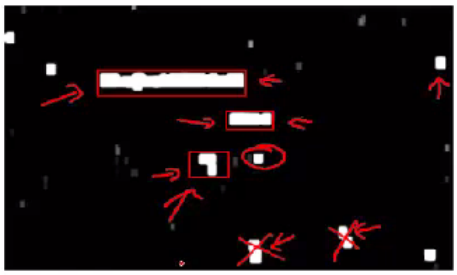
- Having trained the classifier we apply it to an image
  - So, run a sliding window classifier at a fixed rectangle size
  - If you do that end up with something like this



  - White region show where text detection system thinks text is
    - Different shades of gray correspond to probability associated with how sure the classifier is the section contains text
      - Black - no text
      - White - text

        - For text detection, we want to draw rectangles around all the regions where there is text in the image
    - Take classifier output and apply an **expansion algorithm**
        - Takes each of white regions and expands it
        - How do we implement this
            - Say, for every pixel, is it within some distance of a white pixel?
            - If yes then colour it white

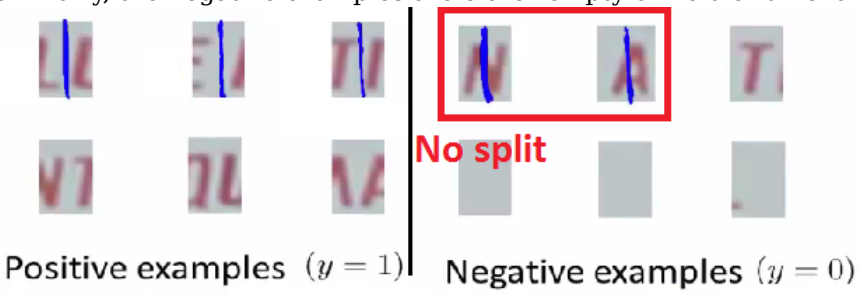
    - Look at connected white regions in the image above
        - Draw rectangles around those which make sense as text (i.e. tall thin boxes don't make sense)



    - This example misses a piece of text on the door because the aspect ratio is wrong
        - Very hard to read

Stage two is character segmentation

- Use supervised learning algorithm
- Look in a defined image patch and decide, is there a split between two characters?
    - So, for example, our first training data item below looks like there is such a split
    - Similarly, the negative examples are either empty or hold a full characters



- We train a classifier to try and classify between positive and negative examples
    - Run that classifier on the regions detected as containing text in the previous section
- Use a 1-dimensional sliding window to move along text regions
    - Does each window snapshot look like the split between two characters?
        - If yes insert a split
        - If not move on
    - So we have something that looks like this

Character classification

- Standard OCR, where you apply standard supervised learning which takes an input and identify which character we decide it is
    - Multi-class characterization problem

Getting lots of data: Artificial data synthesis

- We've seen over and over that one of the most reliable ways to get a high performance machine learning system is to take a low bias algorithm and train on a massive data set
    - Where do we get so much data from
    - In ML artifice data synthesis
        - Doesn't apply to every problem
        - If it applies to your problem can be a great way to generate loads of data
- Two main principles
    - 1) Creating data from scratch
    - 2) If we already have a small labeled training set can we amplify it into a larger training set

Character recognition as an example of data synthesis

- If we go and collect a large labeled data set will look like this
    - Goal is to take an image patch and have the system recognize the character
    - Treat the images as gray-scale (makes it a bit easer)

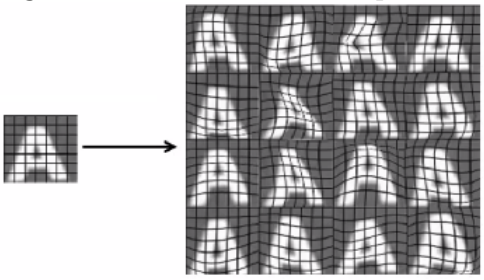
Real data

- How can we amplify this
    - Modern computers often have a big font library
    - If you go to websites, huge free font libraries
    - For more training data, take characters from different fonts, paste these characters again random backgrounds
- After some work, can build a synthetic training set

Synthetic data

- Random background
- Maybe some blurring/distortion filters
- Takes thought and work to make it look realistic
  - If you do a sloppy job this won't help!
  - So unlimited supply of training examples
- This is an example of creating new data from scratch

- Other way is to introduce distortion into existing data
  - e.g. take a character and warp it



  - 16 new examples
  - Allows you amplify existing training set
  - This, again, takes though and insight in terms of deciding how to amplify

Another example: speech recognition

- Learn from audio clip - what were the words
  - Have a labeled training example
  - Introduce audio distortions into the examples
- So only took one example
  - Created lots of new ones!
- When introducing distortion, they should be reasonable relative to the issues your classifier may encounter

Getting more data

- Before creating new data, make sure you have a low bias classifier
  - Plot learning curve
- If not a low bias classifier increase number of features
  - Then create large artificial training set
- Very important question: How much work would it be to get 10x data as we currently have?
  - Often the answer is, "Not that hard"
  - This is often a huge way to improve an algorithm
  - Good question to ask yourself or ask the team
- How many minutes/hours does it take to get a certain number of examples
  - Say we have 1000 examples

    - 10 seconds to label an example
    - So we need another 9000 - 90000 seconds
    - Comes to a few days (25 hours!)
- Crowd sourcing is also a good way to get data
    - Risk or reliability issues
    - Cost
    - Example
        - E.g. Amazon mechanical turks

Ceiling analysis: What part of the pipeline to work on next

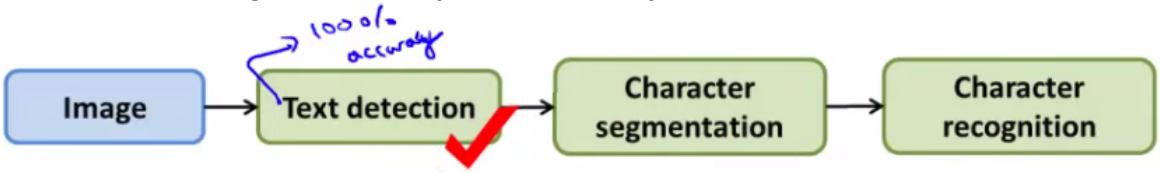
- Through the course repeatedly said one of the most valuable resources is developer time
    - Pick the right thing for you and your team to work on
    - Avoid spending a lot of time to realize the work was pointless in terms of enhancing performance

Photo OCR pipeline

- Three modules
    - Each one could have a small team on it
    - Where should you allocate resources?
- Good to have a single real number as an evaluation metric
    - So, character accuracy for this example
    - Find that our test set has 72% accuracy

Ceiling analysis on our pipeline

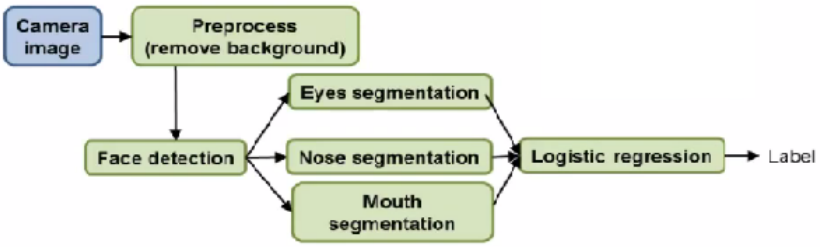
- We go to the first module
    - Mess around with the test set - manually tell the algorithm where the text is
    - Simulate if your text detection system was 100% accurate
        - So we're feeding the character segmentation module with 100% accurate data now
    - How does this change the accuracy of the overall system



    - Accuracy goes up to 89%
- Next do the same for the character segmentation
    - Accuracy goes up to 90% now
- Finally doe the same for character recognition
    - Goes up to 100%
- Having done this we can qualitatively show what the upside to improving each module would be
    - Perfect text detection improves accuracy by 17%!
        - Would bring the biggest gain if we could improve
    - Perfect character segmentation would improve it by 1%
        - Not worth working on
    - Perfect character recognition would improve it by 10%
        - Might be worth working on, depends if it looks easy or not
- The "ceiling" is that each module has a ceiling by which making it perfect would improve the system overall

Other example - face recognition

- NB this is not how it's done in practice



  - Probably more complicated than is used in practice
- How would you do ceiling analysis for this
  - Overall system is 85%
  - Perfect background -> 85.1%
    - Not a crucial step
  - + Perfect face detection -> 91%
    - Most important module to focus on
  - + Perfect eyes ->95%
  - + Perfect Nose -> 96%
  - + Perfect Mouth -> 97%
  - + Perfect logistic regression -> 100%
- Cautionary tale
  - Two engineers spent 18 months improving background pre-processing
    - Turns out had no impact on overall performance
    - Could have saved three years of man power if they'd done ceiling analysis