Stanford

Machine Learning

Andrew Ng - Summer 2015

Contributors: Max Smith

Latest revision: June 27, 2015

Contents

Abstract

This course provides a broad introduction to machine learning, datamining, and statistical pattern recognition. Topics include: (i) Supervised learning (parametric/non-parametric algorithms, support vector machines, kernels, neural networks). (ii) Unsupervised learning (clustering, dimensionality reduction, recommender systems, deep learning). (iii) Best practices in machine learning (bias/variance theory; innovation process in machine learning and AI). The course will also draw from numerous case studies and applications, so that you'll also learn how to apply learning algorithms to building smart robots (perception, control), text understanding (web search, anti-spam), computer vision, medical informatics, audio, database mining, and other areas.

1 Linear Regression with One Variable

Model Representation

- Goal is model labelled data (data which we have the correct output for) to a line
- Notation:

  m = number of training examples

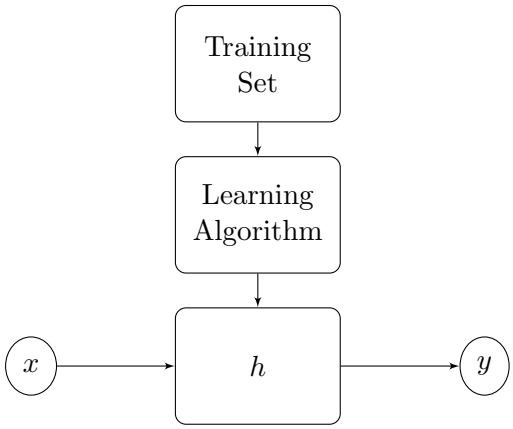
  x = input variable/feature

  y = output variable/feature

  (x, y) = one training example

  $(x^{(i)}, y^{(i)})$ = ith training example (parens indicate index)

- We take a training set, input into a learning algorithm, which returns a hypothesis (h) that models the relationship.



- h maps from x's to y's ($h(x) = y$).
- We need to determine how we want to represent h
- A simple linear model with one variable for h is:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

  , called **univariate linear regression**.

Cost Function

- Given a hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

  $\theta_i's$ = parameters of the model

- We will now discuss how to choose the parameters of our model
- Idea: choose θ_0, θ_1 so that $h_\theta(x)$ is close to y for our training examples (x, y)
- We want to minimize θ_0, θ_1 such that $h(x) - y$ is minimal (reminder: $h(x)$ is the guess at the correct value at y).
- Because we we only are looking to minimize our absolute distance, we square the distance we want to minimize to account for positive and negative differences equally now making our cost function: $(h(x) - y)^2$
- However, we don't want to minimize it for just one example, so we do this for every training example:

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

- To make later math easier, we further refine our formula to be half the average:

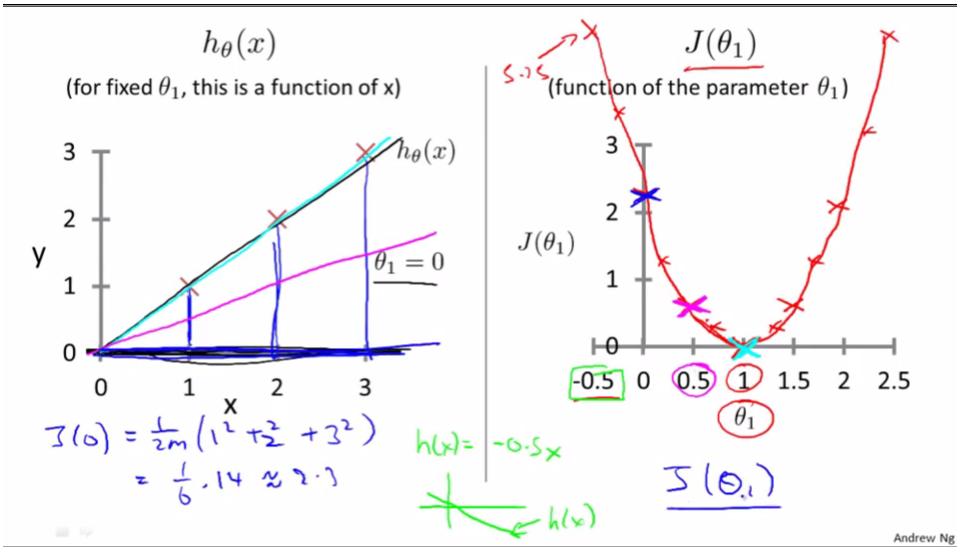
$$\frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

- This function we created is called our **cost** function, as it measures how expensively incorrect our current model is, which we will denote with J.
- The cost function is dependent on the hypothesis parameters, and our goal is to adjust these parameters to minimize the overrall cost of our model:

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

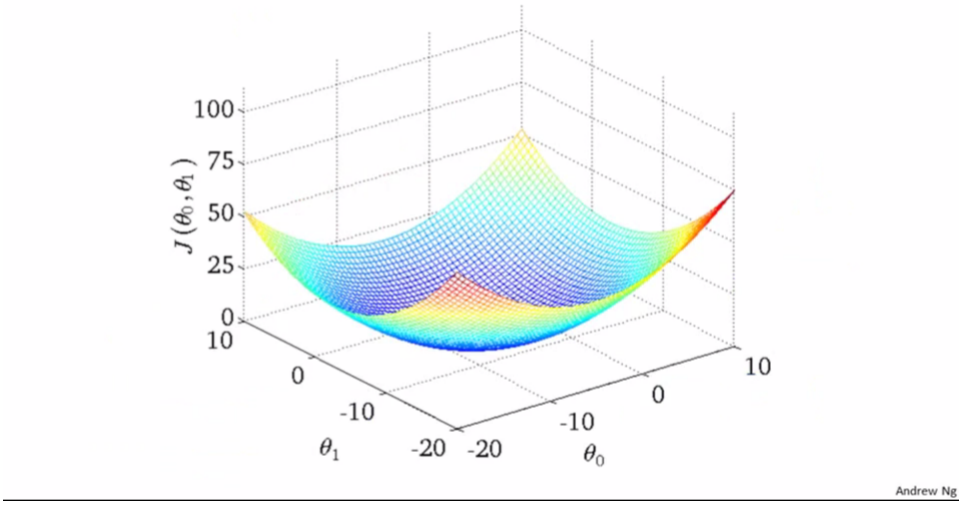
- Now our goal is to minimize J over the variables θ_0, θ_1

Cost Function - Intuition I



This image shows that for varying parameter values, the cost function changes. In this idealistic example there's a global minimum, the goal of minimized cost, that is very easily followed by a hill-climbing style algorithm.

Cost Function - Intuition II



Similarly when you have an additional variable, you want to reach the bottom of this N-dimensional hill (note: not all models will have such a perfect hill).

- The gradient gives the direction of maximumal increase on a surface.
- We will use a negative gradient to find the 'direction' to travel towards the bottom of the hill
- Another common way to represent multidimensional cost functions is through contour plots
-

Gradient Descent

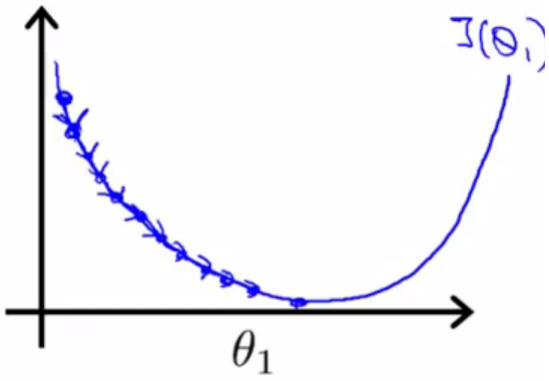
- Given some function $J(\theta_0, \theta_1, \ldots, \theta_n)$ we want to minimize J with respect to $\theta_0, \theta - 1, \ldots, \theta_n$.
- Choose initialize values for the parameters (eg. $\theta_0 = \ldots = \theta_n = 0$)
- Iteratively change $\theta_0, \ldots, \theta_n$ to reduce $J(\theta_0, \ldots, \theta_n)$, until hopefully a minimum is achieved.

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

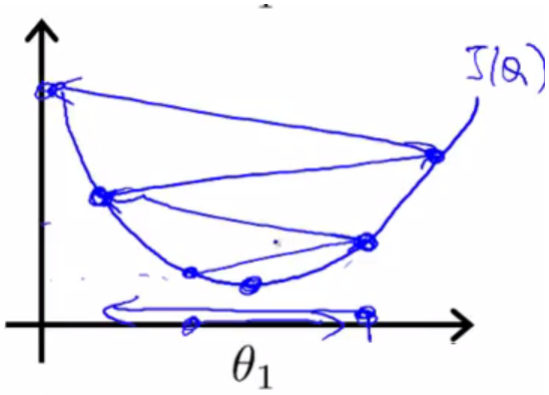
- α is the learning rate, which determines how much change happens in each update
- Ensure simultaneous update, store in temps and then assign.
- An issue with gradient descent is finding local minimums, because you won't be able to find the optimal solution.
-

Gradient Descent Intuition

- To simplify, we will consider the cost funciton with 1 variable ($J(\theta_0)$).
- The negative gradient means that you negative slope, which results in increases with negative slope and dcreases with positive slopes
- If α is too small, gradient descent can be very slow.



- If α is too large, it may fail to converge (not reach minimum).



- If you're already at the local minimum, you will not change your parameters because the gradient is zero.
- You can still converge to a local minimum with a fixed α (learning rate) because as we approach the minimun the gradient descent will automatically take smaller steps.

Gradient Descent for Linear Regression

- Before we continue, we must calculate what the derivative term is:

$$\begin{aligned}\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1) &= \frac{\partial}{\partial\theta_j}\frac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}-y^{(i)}))^2 \\ &= \frac{\partial}{\partial\theta_j}\frac{1}{2m}\sum_{i=1}^{m}(\theta_0+\theta_1x^{(i)}-y^{(i)})^2\end{aligned}$$

– In our linear model we have $j = 0, 1$; therefore, we can simplify our equation further for each case of j:

$$j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)} - y^{(i)})$$

$$j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)} - y^{(i)}) x^{(i)}$$

– Linear regression will always have a convex function for it's cost; namely, it is bowl shaped and doesn't have any local min besides the global - always finds best solution.

– **Batch:** each step of gradient descent uses all the training examples

– Gradient descent scales better than normal equations, which is an advanced linear algebra technique that finds the parameters in closed forms - no iterations.

2 Linear Regression with Multiple Variables

Multiple Features

– Instead of just one feature (x), we known multiple features ($x_1, \ldots, x_n$). eg. size, number of bedrooms, number of floors, age of home.

– $x_j^{(i)}$: value of feature j in i^{th} training example

– Now our hypothesis must account for multiple features:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \ldots \theta_n x_n$$

– Again we define $x_0 = 1$ to simplify future math ($x_0^{(i)} = 1$).

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}, \; \theta = \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

– Transposing the θ vector given our assumption for $x_0^{(i)}$ allows us to simplify our hypothesis into:

$$h_\theta(x) = \theta^T x$$

Gradient Descent for Multiple Variables

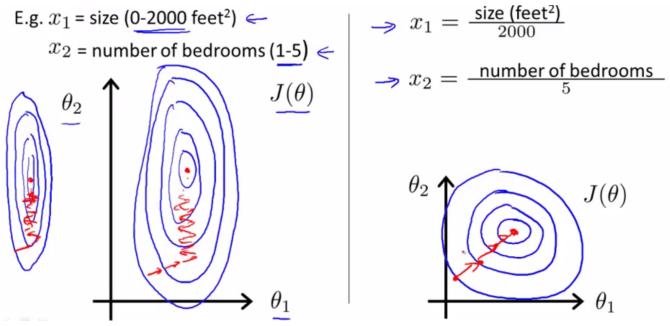
– Repeat until convergence ($j = 0, \ldots, n$):

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

– This is a valid generalization of the previous formula because of our base case $x_0^{(i)} = 1$

Gradient Descent in Practice I - Featuer Scaling

- **Feature scaling:** if features are on similar scales then we converge more quickly
- Your parameters will oscillate along the larger ranged parameter making it's way much slower towards the center (in the case of two variables); whereas, if both axis were equal then you don't have a worst case to fret about



- Typically, we want to scale each feature into approximately a $-1 \leq x_i \leq 1$ range (same order of magnitude).
- **Mean normalization:** replacing x_i with $x_i - \mu_i$ to make features have approximately zero mean (does not apply to $x_0 = 1$).
- Combining mean normalization and feature scaling we assign $x_i := \frac{x_i - \mu_i}{range_i}$

Gradient Descent in Practice II - Learning Rate

- To ensure gradient descent is working correctly, plot the cost function against the number of iterations. It should converge towards 0, decreasing at every iteration.
- The number of iterations required can vary widely for different applications
- You can create an automatic convergence test to ensure appropriately ending of gradient descent by checking if the difference between two iterations ϵ is below a threshold.
- If there is any increase in slope, use a smaller α
- For sufficiently small α, $J(\theta)$ should decrease on every iteration
- But if α is too small, gradient descent an be slow to converge
- To choose α try: $\ldots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, \ldots$

Features and Polynomial Regression

- Suppose we have a housing price prediction: $h(x) = \theta_0 + \theta_1(frontage) + \theta_2(depth)$
- We can define a new feature $(area) = (frontage)(depth)$, that we can use in a new hypothesis $h(x) = \theta_0 + \theta_1(area)$
- We can map these hypothesis of more complex features into a linear regression problem:

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \qquad = \theta_0 + \theta_1(size) + \theta_2(size)^2 + \theta_3(size)^3$$

- If your features are like those chosen, then feature scaling is very important
- There are many more choices for modifications to our features (such as: $\sqrt{}$).
- Trying new features can allow you to have a more appropriate model

Normal Equations

- The **normal equation** allows us to solve for θ analytically (without iterations)
- Intuition: $J(\theta) = a\theta^2 + b\theta + c$ In previous calculus classes you would find the minimum by taking the derivative set equal to 0 and solving for θ.
- This can be extended with partial fractions and solving for every $\theta_j \in \theta$.

$$\frac{\partial}{\partial \theta_j} J(\theta) = \ldots = 0 \text{ (for every} j)$$

- We construct a matrix from the features and a vector from the solutions as so (n features, m examples):

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \cdots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{(m)} & x_1^{(m)} & \ldots x_n^{(m)} & \end{bmatrix} \in \mathbb{R}^{m \times (n+1)}$$

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times 1}$$

- We can then represent the θ by the **normal equation**

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

- X is entitled the **design matrix**
- Normal equation does not perform well with a large n due to the computation $(X^T X)^{-1} \in \ltimes \times \ltimes$ which is typically $\mathcal{O}(n^3)$

3 Logistic Regression

Multiple Features

- A potential binary classification solution is to make the binary decision based on a threshold. eg. threshold = 0.5:

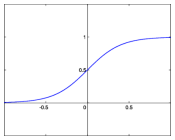
$$\begin{aligned} h(x) &\geq 0.5 && \text{predict } y = 1 \\ h(x) &< 0.5 && \text{predict } y = 0 \end{aligned}$$

- We want a classifier that results in $0 \leq h \leq 1$, as we only have 2 outcomes 0, 1

Hypothesis Representation

- **Sigmoid/Logistic function**:

$$g(z) = \frac{1}{1 + e^{-z}}$$



- Note: there are horizontal asymptotes at 0 and 1.
- We will modify our original hypothesis to now be: $h(x) = g(\theta^T x)$, where g is the previously defined sigmoid function
- Interpretation of hypothesis output:

$$h_\theta(x) = \text{ estimated probability that } y = 1 \text{ on input } x$$

- Eg. $h(x) = 0.7 \rightarrow 70\%$ chance of tumor being malignant
- $h(x) = p(y = 1|x; \theta)$ is another way of defining this.
- $p(y = 0|x; \theta) + p(y = 1|x; \theta) = 1$

Decision Boundary

- Suppose we predict $y = 1$ if $h(x) \geq 0.5$. Graphically, we can seee that this is the same as predicting 1 when $\theta^T x \geq 0$
- **Decision Boundary**: region where $h(x)$ is equal to the threshold (the line that seperates 0 predictions vs 1 predictions).
- This concept can be expanded with the higher powered functions, to result in non-linear decision boundaries

$$h(x) = g(\theta_0 + \theta_1 x + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$$

$$\text{Predict } y = 1 \text{ if } -1 + x_1^2 + x_2^2 \geq 0$$

Cost Function

- How do we choose/fit the parameters θ?
- We abstract our linear regression cost function to be:

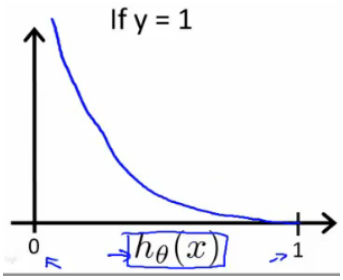
$$j(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} (h(x^{(i)} - y^{(i)}))^2 = \frac{1}{m} \sum_{i=1}^{m} \text{cost}(h(x^{(i)}, y)$$

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

- However, this cost function is non-convex for logistic regression, which doesn't allow us to run gradient descent (no garuntee of global minimum reached).

– Let our cost function for logistic regression now be defined as:

$$\text{cost}(h(x), y) = \begin{cases} -\log(h(x)) & y = 1 \\ -\log(1 - h(x)) & y = 0 \end{cases}$$



– This allows us to have no cost when we were correct at our guess, but have increasingly greater cost the more wrong we were.

Simplified Cost Function and Gradient Descent

– We can simplify the cost function to a single formula:

$$\text{cost}(h(x), y) = -y \log(h(x)) - (1 - y) \log(1 - h(x)$$

– This formula works because when $y = 1$ the portion of the equation that is relevant for 0 becomes 0 via $(1 - y)$

– Similarly for $y = 0$.

– Note: $y \in 0, 1$ always.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{cost}(h(x^{(i)}), y^{(i)}) \qquad = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log h(x^i) + (1 - y(i)) \log(1 - h(x(i))) \tag{1}$$

– Note the negative sign was pulled out of the summation and brought in front of the summation

– To implement gradient descent we must first fit the parameters θ to the model by minimizing $J(\theta)$

– Then we can make a predition given the new x using: $h(x) = \frac{1}{1+e^{-\theta^T x}}$

– We again minimize our cost function using gradient descent, where:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)} - y(i)) x_j^{(i)}$$

– To ensure that the learning rate α is set properly, remember to plot the cost function $(J(\theta))$ as a function of number of iterations and make sure $J(\theta)$ is decreasing on every iteration

Advanced Optimization

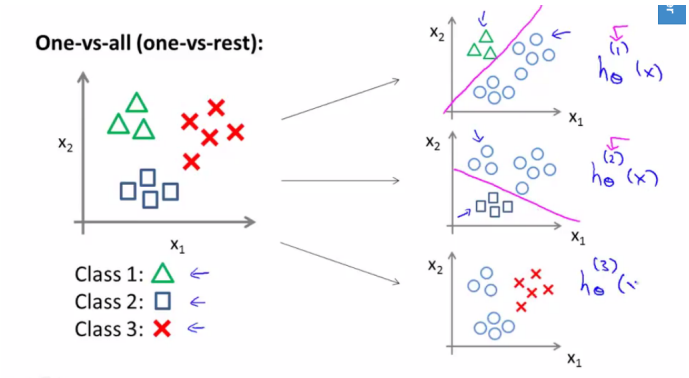
- Optimization algorithm: has the goal of minimizing a cost function $J(\theta)$.
- Given θ we have code that can compute:
  - $J(\theta)$ (to monitor convergence)
  - $\frac{\partial}{\partial \theta_j} J(\theta)$ for $(j = 0, 1, \ldots, n)$
- Gradient descent repeats the following until convergence:

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

- Gradient descent isn't the only optimization algorithm:
  - Conjugate gradient
  - BFGS
  - L-BFGS
- They have the advantages:
  - No need to manually pick α
  - Often faster than gradient descent
- And the disadvantages:
  - More complex
- Don't implement these yourself, use package implementation

Multiclass Classification: One-vs-All

- Example of **multiclass classification**: email foldering/tagging: work, friends, family, hobby, etc.
- In **one-vs-all** we compare each classification against all other possibilities.

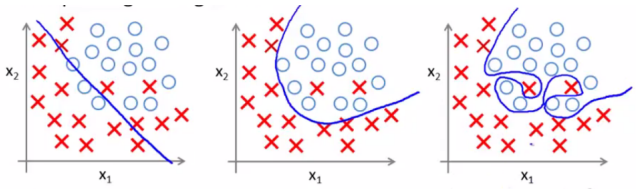


- This gives us a classifier for each class
- Each classifier estimates the probability that the value is that particular class
- This allows us to guess the particular class by taking the representative class of the maximum probability classifier across all classifiers

4 Regularization

The Problem of Overfitting

- **Underfitting**: when a model cannot capture the underlying trend of the data ("high bias")
- **Overfitting**: when a model captures the noise of the data ("high variance")
- **Generalize**: fails to fit to new examples
- Overfitting's poor generalization results in low costs not always being correct



- If we have too many features for very little data, overfitting can easily become a big problem.
- Options:
  - Reduce number of features
    - Manually select which features to keep
    - Model selection algorithm (later in course)
  - Regularization
    - Keep all the features, but reduce magnitude/values of parameters θ_j
    - Works well when we have a lot of features, each of which contributes a bit to predicting y

Cost Function

- Having smaller values for parameters $\theta_0, \theta_1, \ldots, \theta_n$
  - "Simpler" hypothesis
  - Less prone to overfitting
- To exemplify lets consider the housing scenario:
  - Features: $x_1, \ldots, x_100$
  - Paremeters: $\theta_0, \ldots, \theta_100$
  - We don't know which ones are complex to shrink, so we modify the cost function to shrink every parameter

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

  - We don't penalize θ_0 by convention, because it's a constant and makes very little difference
- λ is called the regularization parameter. It controls the trade off of fitting the training set well and keeping the parameters small and simple to prevent overfitting
- If λ is set to be very large we will penalize all the paramters extremely highly resulting which will result in all by the constant being close to zero (fits to a horizontal line). "Underfit"

Regularized Linear Regression

- Using the new regularized linear regression cost function from the previous section we can now update our gradient descent algorithm to encorporate this modification:

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

$$\theta_j := \theta_j - \alpha (\frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y(i))x) j^{(i)} + \frac{\lambda}{m} \theta_j)$$

- The θ_j $(j = 1, \ldots, n)$ term can also be written:

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

- The term $(1 - \alpha \frac{\lambda}{m})$ is typically < 1, so this results in shrinking θ_j by multiplying by a value < 1, and then performing the same gradient descent function
- We also had the normal equation to solve the same problem, and it can also be updated for regularization:

$$\theta = (X^T X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ldots & \\ & & & 1 \end{bmatrix}) X^T y$$

- Suppose $m \leq n$ then $(X^T X)$ will be non-invertible/singular
- Regularization will take care of this flaw so long as $\lambda > 0$

Regularized Logistic Regression

- We can also regularize logistic regression in a similar manner

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

$$\theta_j := \theta_j - \alpha (\frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y(i))x) j^{(i)} + \frac{\lambda}{m} \theta_j)$$

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

5 Neural Networks: Representation

Non-linear Hypothesis

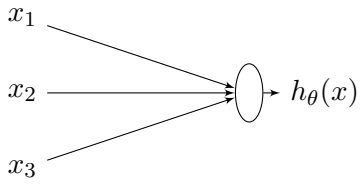
- Suppose you have a housing classification problem with different features $x_1, \ldots, x_1 00$ and if we were to include all quadratic terms for linear classification there would be an enourmous number of terms (5000 features).
- Using linear classifiers has an extremely bad asymptotic complexity (n is typically large)
- For example computer vision problems are $\mathcal{O}(n^2)$
- Neural networks turn out to be a much better way to solve these style problems

Neurons and the Brain

- Neural networks origins are in algorithms that try to mimic the brain
- **"One learning algorithm hypothesis"**: x cortex can learn whatever is hooked up to it
  - Auditory cortex can learn to see
  - Somatosensory cortex (touch) can learn to see
  - There is one algorithm that can teach anything to do any function

Model Representation I

- Neural networks work by simulating the neurons in the brain
- Has "input wires" dendrite
- Has "output wires" axon
- Neuron is a computational unit that takes in inputs and produces an output
- Neurons communicate with little spikes of electricity through their axons, which another neuron can receiv with its dendrite
- In an artificial neural network we model a neuron as a logistic unit:



$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}, h(x) = \frac{1}{1 + e^{-\theta^T x}}$$

  - Here the arrows coming from the x are the input wires
  - The neuron does the computation
  - Finally the output comes out
- The x_0 is called the **bias unit** and is sometimes omitted because it's constant.
- **Activation function**: defines the output of that node given an input or set of inputs
- "weights" are synonomous with parameters of the model
- Neural networks are groups of neurons strung together

  TODO: Neural Network

$$a_1^{(2)} = g(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3)$$

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

- **Input layer**: first layer of inputted values
- **Output layer**: the final layer that calculates the output value
- **Hidden layer**: the center layers that don't have known outputs (not input or output layer)
- $a_i^{(j)}$ = "activation" of unit i in layer j
- $\theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$
- If a network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$

Model Representation II

- Consider the previous neural network, where we performed the 4 large equations to calculate the output; we will modify the notation to be of the form:

$$a_1^{(2)} = g(z_1^{(2)})$$
$$a_2^{(2)} = g(z_2^{(2)})$$
$$a_3^{(2)} = g(z_3^{(2)})$$

- Let us define:

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

- This allows us to perform vectorized calculations:

$$z^{(2)} = \theta^{(1)}x$$
$$a^{(2)} = g(z^{(2)})$$

- If we instead consider the input layers to be the first layer of activation $a^{(1)} = x$, we may redefine our formulas:

$$z^{(2)} = \theta^{(1)}a^{(1)}$$
$$a^{(2)} = g(z^{(2)})$$

- We can account for any bias units by including $a_0^{(k)} = 1$

$$h(x) = a^{(3)} = g(z^{(3)}) = g(\theta^{(2)}a^{(2)})$$

- **Forward propagation**: information only moves in one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes

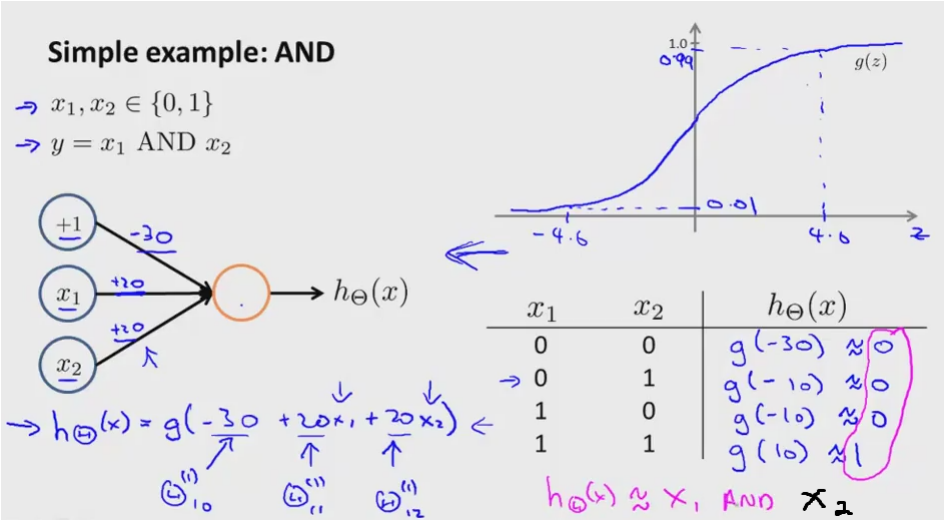
- If you cover part of the neural network, only showing the last two layers, you'll notice that it's just logistic regression
- However instead of using the original features $x_1, \ldots, x_n$, they're using the new features it learned on its own $a_1, \ldots, a_n$
- This allows you to use better features than if you were constrained to only your own features, the network has the ability to learn any new features it wants
- The **network architecture** is how the neurons are laid out and connected

Examples and Intuitions I

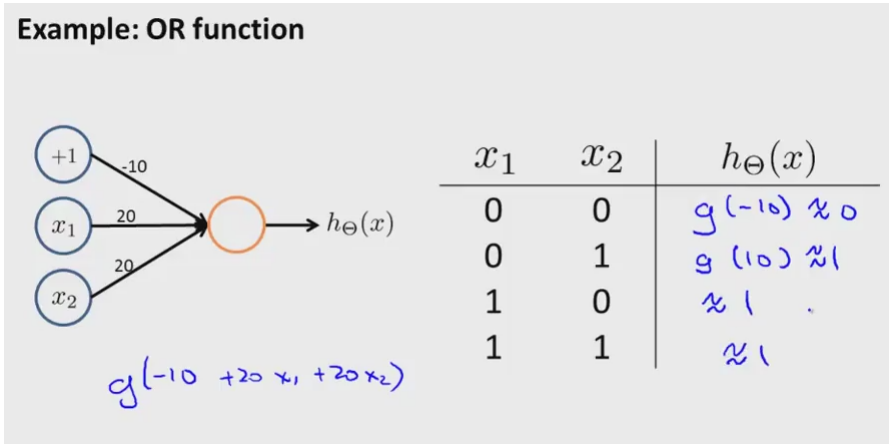
- Non-linear classification example: XOR/XNOR, is difficult to to model

  TODO Graph

- For example we can model simple logical operations with logistic regression:



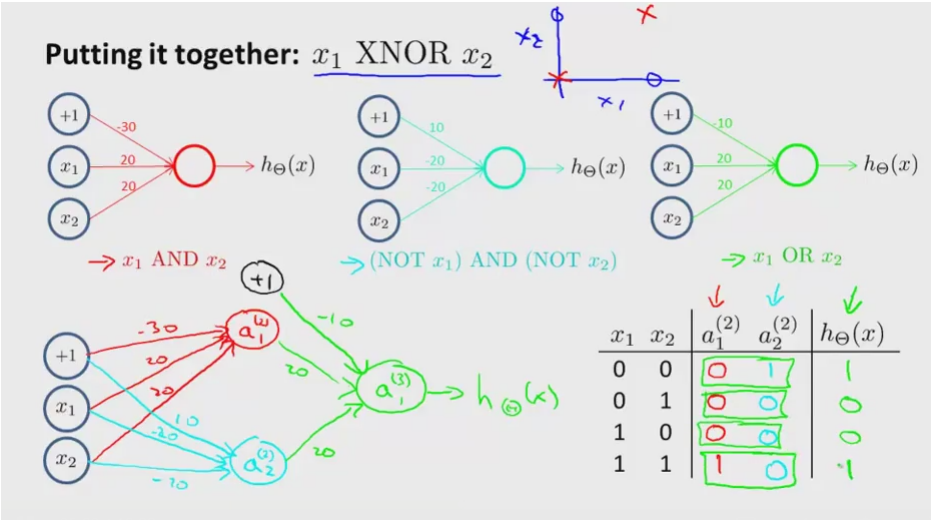
x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$



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

Examples and Intuitions II

- For models that have non linear boundaries, using a multi-layered network is the best way to approximate the model, for example:



- Each layer of the neural network is able to compute even more complex features
- The last layer makes the prediction about the correct classification

Multiclass Classification

- Multiclassification is an extension of a one-vs-all
- We want to have the same output units as classes $h(x) \in \mathbb{R}^k$
- Each is a classifier for each class, ie. Is it a '1,' is it a '2,' etc..
-

6 Neural Networks: Learning

Cost Function

- Suppose we have:
  - Training sets: $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
  - $m =$ number of training examples
  - $L =$ total number of layers in network
  - $s_l =$ number of units (not counting bias unit) in layer l
  - $k =$ number of classes
- Binary Classification:
  - $y = 0$ or 1

  - 1 output unit $h(x) \in \mathbb{R}$
  - $s_l = 1$
  - $k = 1$
- Multi-class Classification (K classes)
  - $y \in \mathbb{R}^K$
  - K output units
  - $h(x) \in \mathbb{R}^K$
  - $s_l = K, (K \geq 3)$
- New cost function:

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

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

Backpropagation Algorithm

- In order to perform gradient descent we need to compute $J(\Theta)$, $\frac{\partial}{\partial \Theta_{ij}}{}^{(l)} J(\Theta)$
- We have already defined $J(\Theta)$, so we just need the partial derivatives.
- Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l
- An example is for each output unit (layer $L = 4$): $\delta_j^{(4)} = a_j^{(4)} - y_j$
- We will comput the δ for every layer of the neural network

$$\delta^{(n)} = (\Theta^{(3)})^T \delta^{(n+1)} . \times g'(z^{(n)})$$

- Where $.\times$ is element-wise multiplication
- There is no erro associated with the input layer
- The name backpropagation, comes from calculating the error from the back towards the beginning of the network
- Suppose we have a training set of m examples
- We will set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)
- This will be used to compute the partial derivatives of the cost function

  TODO: turn into pseudocode

  For $i = 1$ to m: Set $a^{(1)} = x^{(i)}$ Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$ Using $y^{(i)}$, comput $\delta^{(L)} = a^{(L)} - y^{(i)}$ Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
- It's possible to vectorize the final update: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$

– After completing the loop we calculate:

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

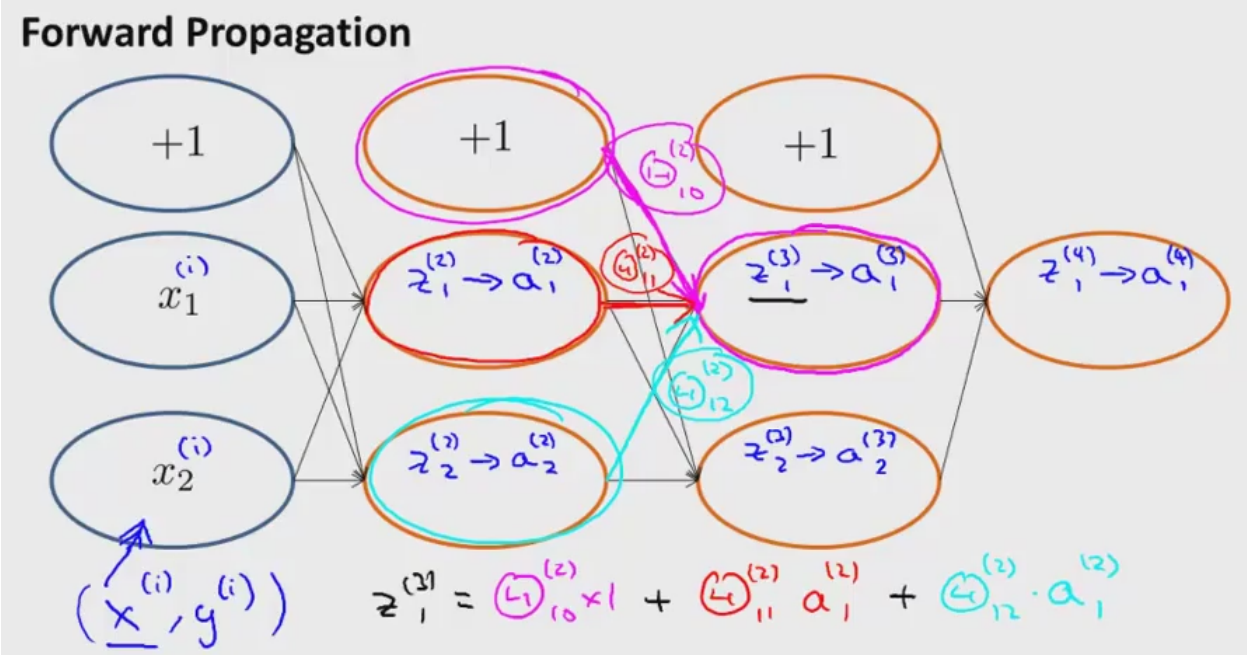
$$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} \text{ if } j = 0$$

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

Backpropagation Intuition

– Let us first take another look at forward propagation

– We first feed an input into the input layer $x^{(i)}$, then we calculate the second layer $z_1^{(2)} \rightarrow a_1^{(2)}$, similarly for the next 2 layers

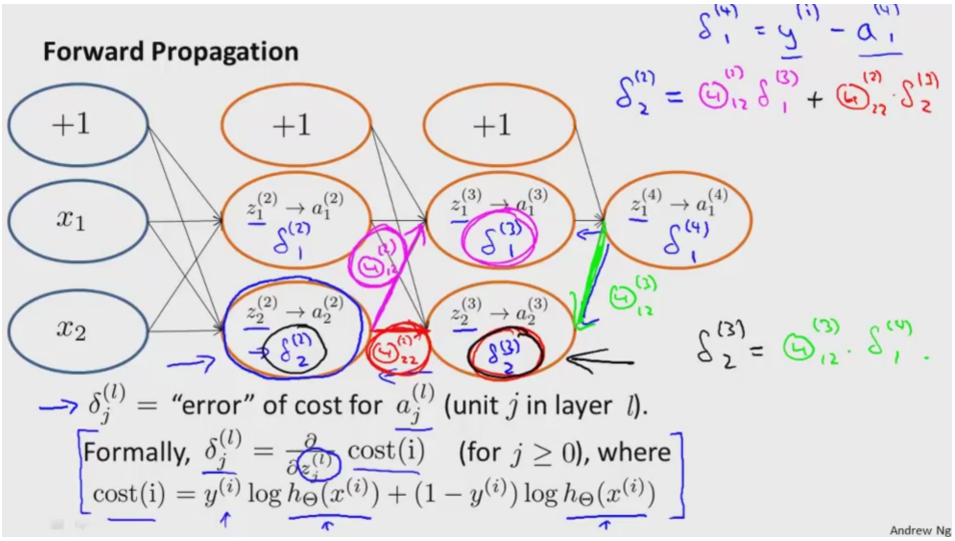


– If you focus on a single example, the case of 1 output unit, and ignoring regularization ($\lambda = 0$):

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

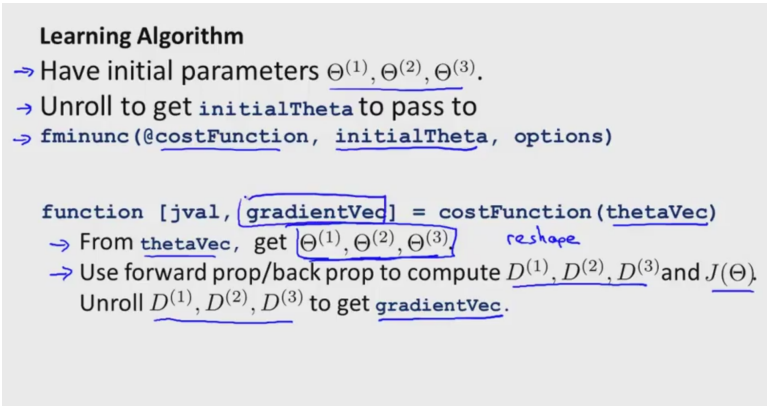
– Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}}\text{cost}(i)$ for $j \geq 0$

– Backpropagation looks identical, but backwards:



Implementation Note: Unrolling Parameters

- Our previous use of minfunc and costFunction in octave assumed that θ and the return would be vectors
- You can unroll elements into large vector: thetaVec = [Theta1(:); Theta2(:); Theta3(:)];
- You can also change them back into the matrix using reshape: Theta1 = reshape(thetaVec(1:size),dim1, dim2);



Gradient Checking

- Suppose you have a function $J(\Theta)$, and we want to estimate the derivative at $\theta \in \mathbb{R}$.
- We consider the points $\theta - \epsilon$ and $\theta + \epsilon$, and draw a line through these points and use the slope of this line as an approximation
- This gives our approximation as:

$$\frac{d}{d\Theta} J(\Theta) = \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

- A good value for epsilon is: $\epsilon = 10^{-4}$
- Now consider $\theta \in \mathbb{R}^n$

- We can calculate the derivative for each element the same as we did for the single variable. Holding the other variables constant
- We then can compare this approximate derivative to the backpropagation to check that they're approximately the same
- Don't do gradient checking once you start learning, because it will be very slow
- j

Random Initialization

- Initializing all the parameters in a neural network, does not work like it did in our previous algorithms
- After each update, parameters corresponding to inputs going into each of two hidden units are identical
- This results in not being able to develop more complex features
- Instead, we will now initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
- The goal is "symmetry breaking"

Putting it Together

- We must pick a network architecture (connectivity pattern between neurons)
- Reasonable default: 1 hidden layer, or if ¿1 hidden layer, have same number of hidden units in every layer (usually the more the better)
- Training a neural network:
  - Randomly initialize weights
  - Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
  - Impelement code to cmpute cost function $J(\Theta)$
  - Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
  - Perform forward propagation and backpropagation using each example
  - Use gradient checking to compare partial derivatives computed using backpropagation vs. using numerical estimate of gradient of J. Then disable gradient checking code.
  - Use gradient descent or advanced optimization method with backpropagation to try and minimize J as a function of parameters Θ
- $J(\Theta)$ is non-convex, and can get stuck in local optimum

7 Advice for Applying Machine Learning

Deciding What to Try Next

- Suppose ou've trained your model, but it is largely error prone. What should you try next?
  - Get more training examples

  - Try smaller sets of features
  - Try getting additional features
  - Try adding polynomial features
  - Try decreasing λ
  - Try increasing λ
- **Diagnostic**: A test that you can run to gain insight what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance
- Diagnostics can take time to implement, but oding so can be a very good use of your time

Evaluating a Hypothesis

- **Overfitting**: fails to generalize to new example not in training set
- How do we determine if it overfits?
- Suppose we have data set $(x^{(i)}, y^{(i)})$, we will split the data into two sets: **training set**, and **test set**
- We will typically assign 70% to be the training set, and the remainder 30% to be the test set
- m_{test} = number of test example
- Training/testing procedure for linear regression:
  - Learn parameter θ from training data (minimizing training error J(θ))
  - Compute test set error:

$$J_{test}(\theta_{training}) = \frac{1}{2m_{test}} \sum_{i=1}^{m)test} (h(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

- Training/testing procedure for logistic regression:
  - Learn parameter θg from trainin data
  - Compute test set error:

$$J_{test}(\theta_{training}) = \frac{-1}{m_{test}} \sum_{i=1}^{m)test} y_{test}^{(i)} \log h(x_{test}^{(i)}) + (1 - y_{test}^{(i)}) \log h(x_{test}^{(i)})$$

  - Misclassification error (0/1 misclassification error):

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

$$\text{Test error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \text{err}(h(x_{test}^{(i)}, y^{(i)})$$

Model Selection and Train/Validation/Test Sets

- If you consider adding another parameter to your model that is: d =degree of polynomial

$$h(x;d) = \sum_{i=0}^{d} \theta_i x^i$$

- If we denote the parameters for each respctive model as $\theta^{(d)}$, we can then consider $J_{test}(\theta^{(d)}$ for each hypothesis
- Suppose we choose a model $\theta^{(5)}$, the problem with this system is it is likely to be an optimistic estimate of generalization error; namely, we fit the new parameter d to the test set
- It is no longer fair to reuse the test set on this model to test it, because it's already favoring it
- Now we will split up the dataset into 3 pieces: **training set**, **cross validation set (cv)**, **test set**
- A typical split is 60%-20%-20%, favoring the training set
- m_{cv} = number of cross validation examples
- Training error:

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

- Cross Validation error

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

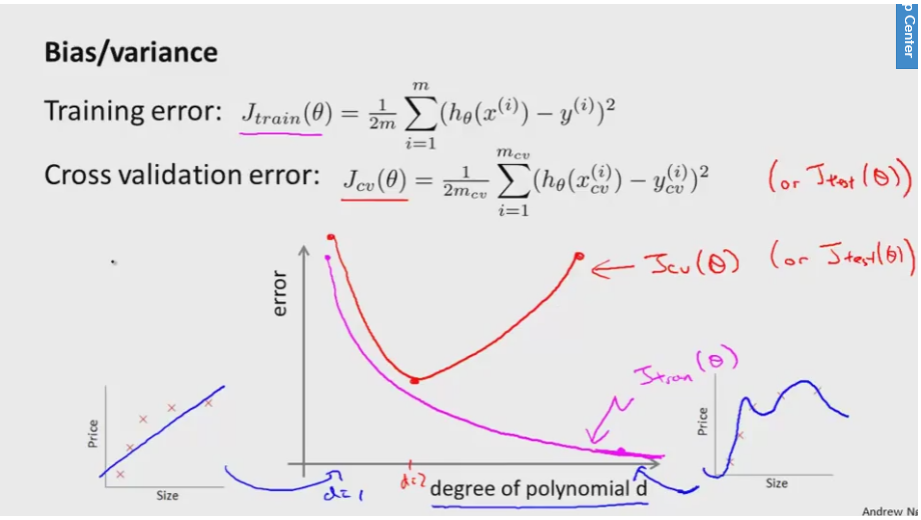
- Test error

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

- Now instead of the test set, we will use the cross validation set to select the model
- We will select the model based on the minimum of $J_{cv}(\theta^{(d)})$
- j

Diagnosing Bias vs. Variance

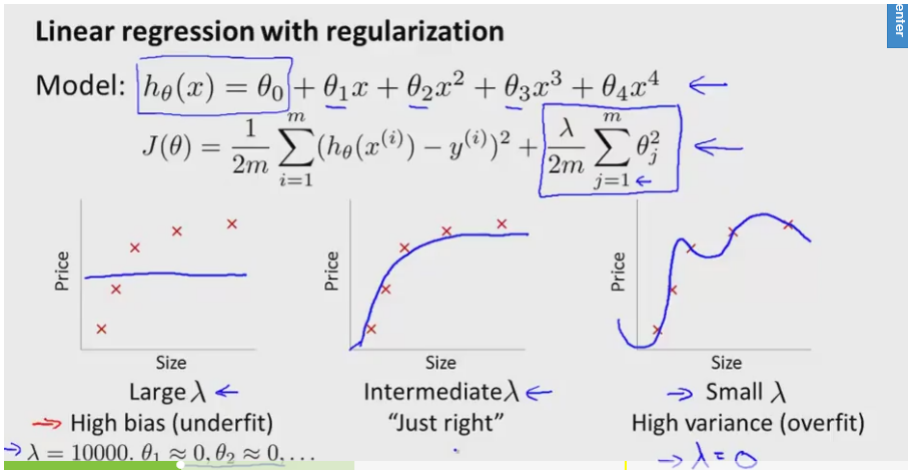
- Underfitting/overfitting problems
- Overfit = high variance
- Underfit = high bias



- High bias is shown on the left, where you have a low order polynomial with a large error
- High variance is shown on the right, where you have a high decree polynomial with a large error
- Your model suffers from high bias (underfit) problem if the training error is high and the cv is approximately the same.
- Your model suffers from high variance (overfit) if the training is low (fit well) while the cv error is much greater than the training error

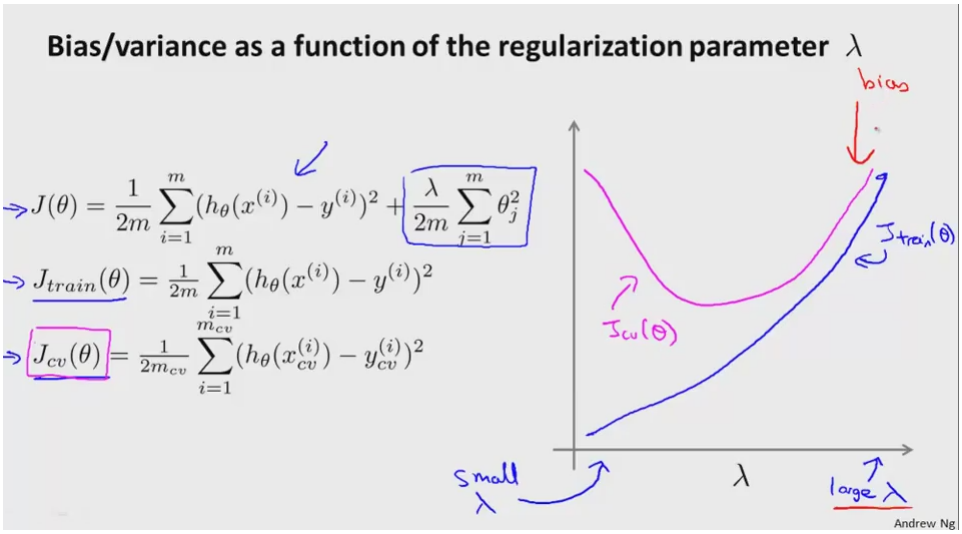
Regularization and Bias/Variance

- In regularization we have a regularization term with a variable λ



- When you have a large λ all parameters are heavily penalized, so only the constant term remains
- Contrasting, when we have a small λ we maintain our very high variance overfitting
- We want to find the "just right" value
- Choosing the regularization parameter λ

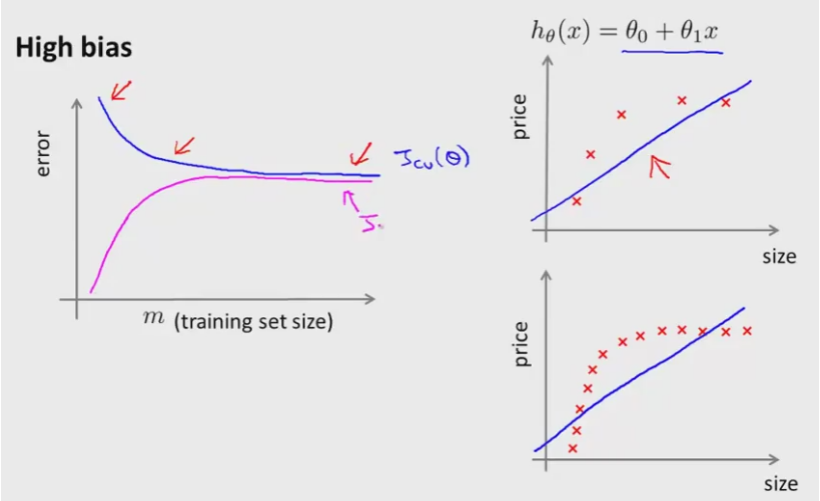
- Try multiples of $\lambda = 0, 0.01, 0.02, 0.04, 0.08, \ldots, 10$
- Let the models of each value of λ be $\theta^{(n)}$
- Choose the smallest $J_{cv}(\theta(n))$



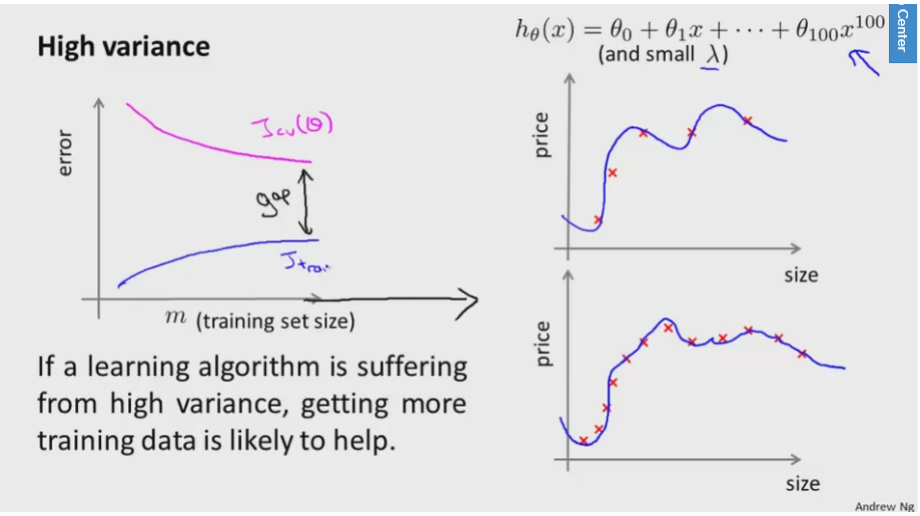
- For small λ you have little penalty and are able to fit your data well
- The cross validation error is very high when λ is small because you fit the too perfectly to other data set

Learning Curves

- Plot J_{train} or J_{cv} against m (training set size)
- To do this we will deliberatly limit our training set size to complete the graph at lower points
- Suppose you have a high bais, If we were to increase the training set size you end up with a similar straight line
- The cv error will plateau out because the line can't conform any better to the data
- The training set also has a similar error because it's unable to also conform to the data
- It has a high bias because both errors are high
- If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.



- Suppose now you have high variance, we will fit the data very well; however, it will largely overfit the data
- Since we can very easily fit the data our training error will remain low
- However, the cross validation error will remain high because it's too perfectly fitted for other data points
- The variance occurs do to this large gap in error between cv and training
- If a learning algorithm is suffering from high variance, getting more training data is likely to help.



Deciding What to Do Next Revisited

- What should you try next?
  - Get more training examples → fixes high variance
  - Try smaller sets of features → fixes high variance
  - Try getting additional features → fixes high bias (typically)
  - Try adding polynomial features → fixes hih bias
  - Try decreasing λ → fixes high bias (less penalty for complex features)

  - Try increasing $\lambda \rightarrow$ fixes high variance (more penalty for complex features)
- "Small" neural networks: fewer parameters; more prone to underfitting; computationally cheaper

  TODO: Draw
- "Large" neural network: more parameters; more prone to overfitting; computationally more expensive; use regularization to address overfitting

  TODO: Draw
- Try using the training/cv/test sets to determine the best choice for number of hidden layers

8 Machine Learning System Design

Prioritizing What to Work On

- Building a spam Classifier:
  - Supervised learning problem.
  - x =features of email
  - y = spam (1) or not spam (0)
  - Featurs x: choose 100 words indicative of spam/not spam
  - $\vec{x} = 1$ when the corresponding word appears, and 0 otherwise (eg. andrew is the first word, and it's in the email $x[0] = 1$)
  - Note: in practice, take most frequently occuring n words (10000 to 50000) in training set, rather than manually pick 100 words
- How to spend our time to make it have low error?
  - Collect lots of data
    - eg. "honeypot" project
  - Develop sophisticated features, for example based on email routing information (from email header)
  - Develop sophisticated features for message body, eg. should "discount" and "discounts" be treated as the same word? How about "deal" and "Dealer"? Features about punctuation?
  - Develop sophisticated algorithm to detect misspellings (eg. m0rtgages, med1cine, w4tches)

Error Analysis

- Recommended approach:
  - Start with a simple algorithm that you can implement quickly. Implement it and test it on our cross-validation data
  - Plot learning curves to decide if more data, more features, etc. are likely to help.
  - Error analysis: Manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on
- This allows evidence to decide our decisions, and not gut feelings
- Manually categorize errors when doing analysis:

  - What type of error is it
  - What cues (features) you think would have helped the algorithm classify them correctly
- The importance of numerical evaluation:
- Should discount/discounts/discounted/discounting be treated as the same word?
- Can use "stemming" software (eg. Porter stemmer)
- Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try and see if it works (hopefully this can be done quickly)
- Then we wil need numerical evaluation (eg. cross validation erro) of algorithm's performance with and without stemming
- Comparing the error between these two, you can easily decide whether or not to use stemming

Error Metrics for Skewed Classes

- Consider training a logistic regression model $h(x)$ ($y = 1$ if cancer, $y = 0$ otherwise)
- We discover that we got 1% error on test set (99% correct)
- However, only 0.50% of patients have cancer
- If we always predicted $y = 0$, we would have better error (0.5%)
- **Skewed classes**: where we have much greater number of examples of one class then the others
- This makes classification accuracy unclear when quality changes in classifier

		Actual Class	
		1	0
Predicted Class	1	True Positive	False Positive
	0	False Negative	True Negative

- **Precision/Recall**:
- **Precision**: $\frac{\text{True Positives}}{\#PredictedPositives}$ (first row)
- **Recall**: $\frac{\text{True positive}}{\#\text{ Actual Positives}}$ (first col)
- $y = 1$ in presence of rare class that we want to detect

Trading Off Precision and Recall

- Continueing the cancer classification problem, with a logistic regression ($y = 1$ has cancer).
- Suppose we want to predict $y = 1$ (cancer) only if very confident
- One way to do this is to set the threshold for predicted $y = 1$ when $h(x)$ is very high ($h(x) \geq 0.7$)

  - Higher precision
  - Lower recall
- Suppose we want to avoid missing too many cases of cancer (avoid false negatives)
  - We may now instead set the threshold very low ($h(x) < 0.3$)
  - High recall, lower precision
  - Catching more cancer individuals, but more incorrect
- How to compare precision/recall numbers?
- Suppose we have 3 different algorithms:

	Precision (P)	Recall (R)
Algorithm 1	0.5	0.4
Algorithm 2	07	0.1
Algorithm 3	0.02	1.0

- Averaging lends itself to Alg 3, which is clearly a bad choice (always predicts $y = 1$), so this is a bad choice
- **F score**: $F_1 = 2\frac{PR}{P+R}$
- There are many different scores to compare precision/recall, but this numeric is the typical beginner level

Data For Machine Learning

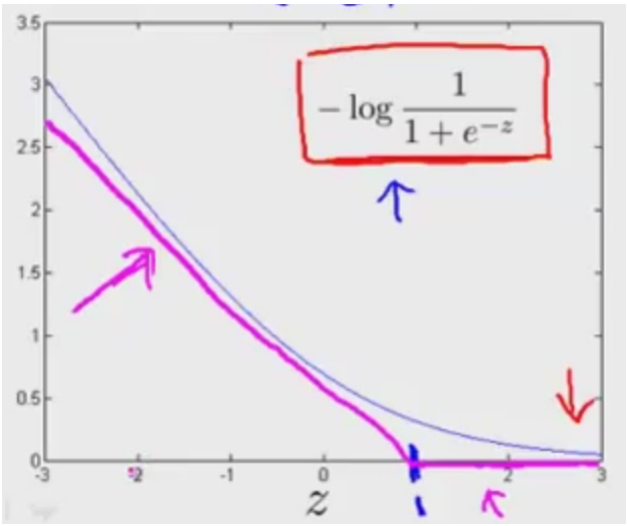
- "It's not who has the best algorithm that wins. It's who has the most data"
- Assume feature $x \in \mathbb{R}^{n+1}$ has sufficient information to predict y accurately.
- Example: For breakfast I ate _ eggs. (predict correct form of: to, too, two)
- Counterexample: predict housing price from only size (feet^2) and no othe features (don't have enough information to accurately predict the price)
- Useful test: Given the input x, can a human expert confidently predict y? (given the features for the model)
- Supose we use a learning algorithm with many parameters (eg. logistic regression/linear regression with many features; neural network with many hidden units)
- J_{train} should be small due to the low bias of these algorithms
- And using our very large training set (unlikely to overfit, low variance) $\rightarrow J_{train} = J_{test}$
- These result in J_{test} being very small

9 Support Vector Machines

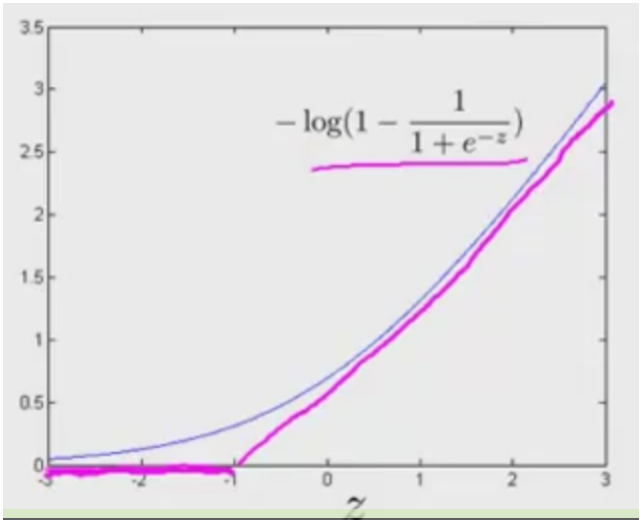
Optimization Objective

- In logistic regression we had the sigmoid activation function, and the hypthosis: $h(x) = \frac{1}{1+e^{-\theta^T x}}$
- If we have an example where $y = 1$, we want $h(x) = 1$, this implies $\theta^T x \gg 0$

- Conversly, if $y = 0$, we want $h(x) = 0 \rightarrow \theta^T x \ll 0$
- When $y = 1$ only the first term of the cost function wil matter $(-y \log \frac{1}{1+e^{-\theta^T x}})$
- If we plot this as a function of $z = \theta^T x$ we get the following curve



- The purple line will be our SVM's cost when $y = 1$, which is made up of two line segments
- Similarly for when $y = 0$ (second term of cost function) we have the same following graphs:



- We will denote the SVM's cost when $y = 0$ to be $\text{cost}_1(z)$, similarly when $y = 0$, $\text{cost}_0(z)$
- Logistic regression's cost function:

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

- Support vector machine's cost function:

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

- We should be able to remove the constant, since the minimization problem will take care of reducing it:

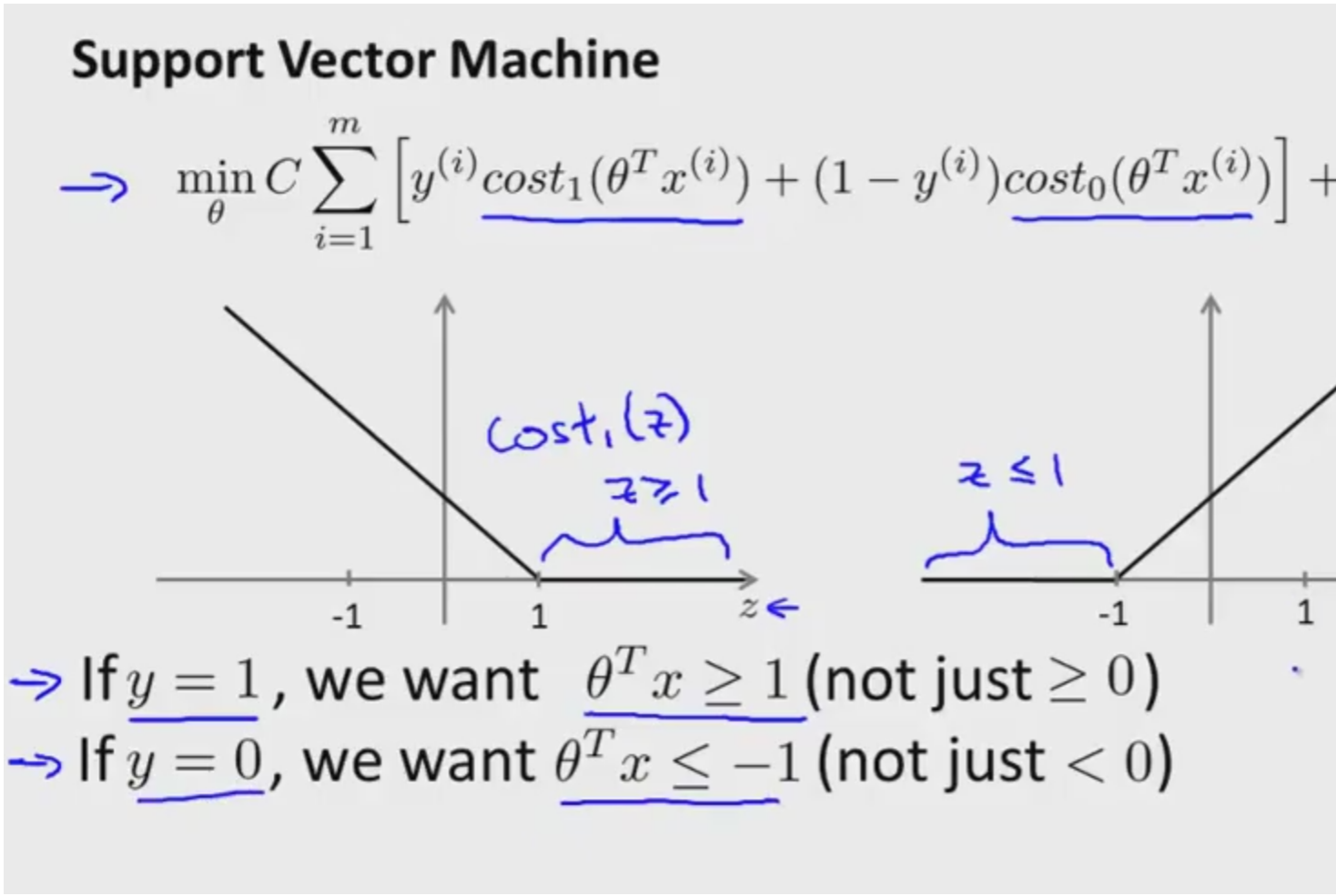
$$\min_\theta \left[\sum_{i=1}^{m} y^{(i)}\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2$$

- Also, for SVM we will use a different parameter to control the weight of the penalty term: $A+\lambda B \rightarrow CA+B$, where C is our new term
- Having learned the parameters θ, this is the hypothesis for the SVM:

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

Large Margin Intuition

- -



- To make these cost functions small, we want:

$$\theta^T x \geq 1 \text{If } y = 1$$
$$\theta^T x \leq -1 \text{If} y = 0$$

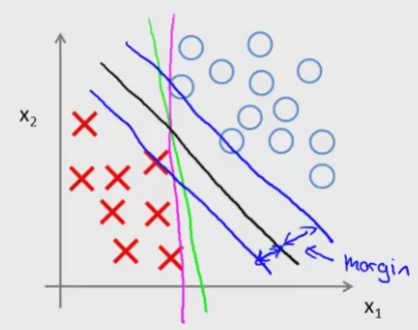
- However, if you just barely get the example right (0 instead of +/- 1), where you stlil have some cost
- This builds an extra safety margin factor, still predicting the correct result but there's another "level" where you will have no cost
- If C is very large, and we want our first cost term to be 0, so our resulting cost function is:

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

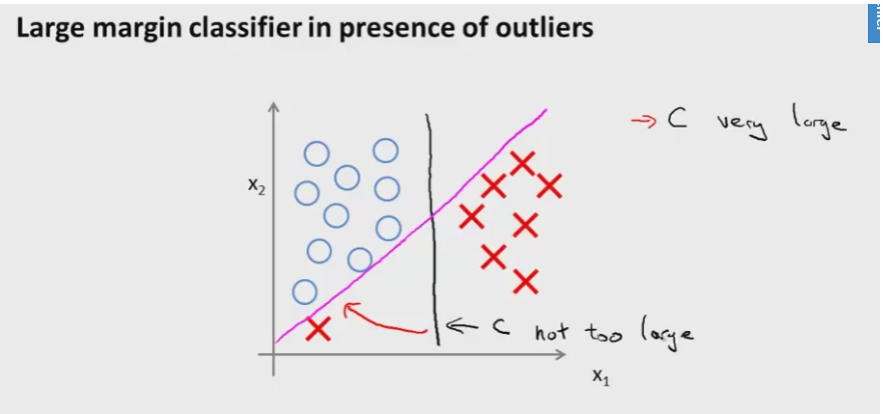
- This is subject to the constraint:

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

- When you solve this optimization problem, you get a linearly separable example sets (can be divided by lines into their classes).



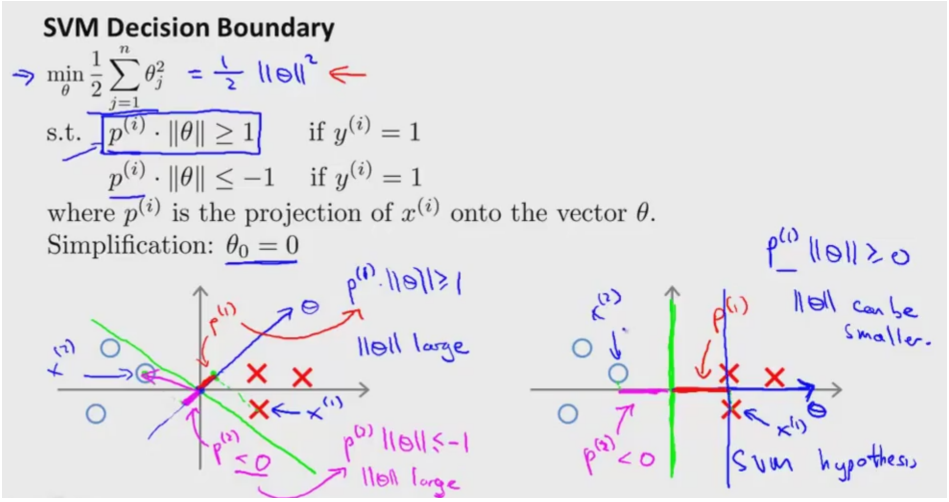
- There exists many different lines (green/purple) that can fit this division; however, the svm will find the black (most natural looking) dividing line
- This line has the largest margin from the classes (the blue lines)
- The distance from the predicted line and the nearest item is the margin
- The svm is called a **large margin classifier** because it finds the largest margin for its model
- Consider adding an outlier as shown below, that the large margin would not be the 'best' idea. If C was very large, this is how the svm will handle the outlier. However, it wasn't, you'd still have hte black decision boundar



Mathematics Behind Large Margin Classification

– In the case of 2 features, where $\theta_0 = 0$

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



– The margin works by trying to get the projectionss of the $y = 1$ examples to be large positive values (θ points towards them, so they have large projections)

– This also results in very large negative values for the $y = 0$

– The $\theta_0 = 0$ assumption, allows this simplification where decision boundary fits to the origin (the ideas proven here work $\theta \neq 0$ as well)

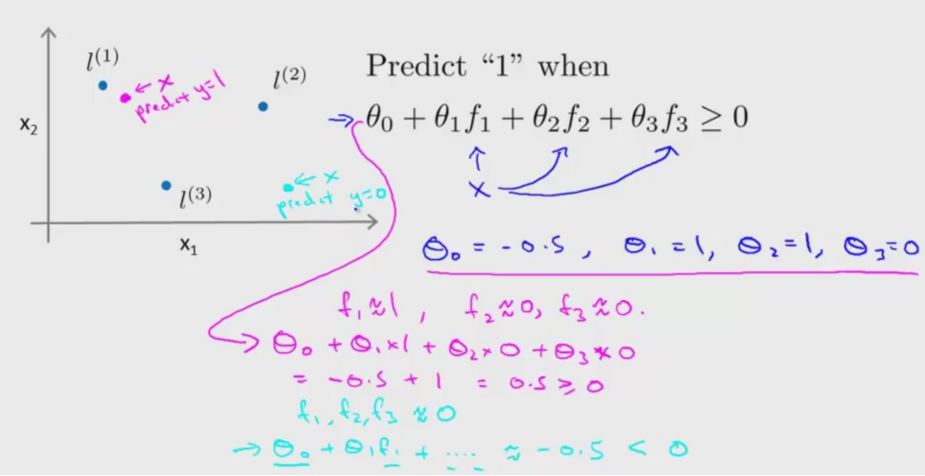
Kernels I

– One way to find a complex non-linear decision boundary is complex polynomials

– Another way to denote this is:

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \dots$$

- In this case:
$$f_1 = x_1, f_2 = x_2, f_3 = x_1 x_2, f_4 = x_1^2, \ldots$$
- Is there a different/better chocie of the features $f_1, f_2, \ldots$?
- Given x, compute new feature depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(2)}$
- One possible feature: $f_1 = \text{similarity}(x, l^{(1)} = \exp(-frac||x - l^{(2)}||^2 2\sigma^2)$
- For this example, we will also define our second & third feature similarly: $f_n = \text{similarity}(x, l^{(n)})$
- This similarity function is called the **gaussian kernels**
- The idea of the similarity function is called the **kernel**
- If $x \approx l^{(1)}$, then $f_1 =\approx \exp(-\frac{0^2}{2\sigma}) \approx 1$
- If x is far from $l^{(1)}$ then $f_1 \approx 0$
- So these measure how close x is to the landmarks
- You can consider a guassian curve on the 3rd dimension, where the closer you get to the landmark, the higher up the hill (the corresponding value of the feature)
- As you increase σ^2 the value falls away much slower (higher variance tolerated)
- Example:



Kernels II

- Where do we get $l^{(1)}, l^{(2)}, \ldots$?
- Given m training examples, we choose $l^{(1)}, \ldots, l^{(m)} = x^{(m)}$
- And our features: $f_m = \text{similarity}(x, l^{(m)})$

- We can now represent our training example i as a stacked vector all of all of the new features:

$$f^{(i)} = \begin{bmatrix} f_0^{(i)} \\ \vdots \\ f_m^{(i)} \end{bmatrix}$$

- SVM with Kernels
  - Hypothesis: Given x, compute features $f \in \mathbb{R}^{m+1}$
  - Predict $y = 1$ if $\theta^T f \geq 0$
  - Training, by solving:

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

  - Note: we are ignoring θ_0 in our second term
- SVM parameters:
  - One of the choices you need to make is what value of C you would like to use
  - Large C: lower bias, high variance
  - Small C: higher bias, low variance
  - We also need to choice σ^2
  - Large σ^2: featurs f_i vary more smoothly (varies slowly). Higher bias, lower variance.
  - Small σ^2: features f_i vary less smoothly (sharper change). Lower bias, higher variance
  -

Using an SVM

- When using software packages to computer svm you need to specify:
  - Choice of parameter C
  - Choice of kernel (similarity function)
  - Choice of parameter σ^2
- **Linear Kernel**: gives standard linear classifier
- Why might you choose a gaussian kernel? If $x \in \mathbb{R}^n$, n small andor large. Then the gaussian kernel can help with this very complex models
- If it asks you to provide a function to compute the kernel function, **perform feature scaling before using the kernel**
- Only kernels that satsify the condition "Mercer's Theorem" to make sure SVM packages' optimizations run correctly, and do not diverge
- Many off-the-shelf kernels available:

  - Polynomial kernel: $k(x, l; c, p) = (x^T l + c)^p$,
  - More esoteric: string kernel, chi-square kernel, histogram intersection kernel, ...
- Use the cross-validation set to decide on a kernel
- Many svm packages already ahve built-in multi-in multi-class classification unctionality. Otherwie, use one-vs-all method.
- Logistic regression vs. svm
  - If the number of training examples (m) is greater than the number of features (n), the typical approach is logistic regression over svm
  - If n is small, and m is intermediate ($n = 1 - 1000, m = 10 - 10000$) use svm with guassian kernel
  - If n is small and m is large, create/add more features, then use logistic regression or svm without a kernel
  - Neural network likely to work well for most of these settings, but may be slower to train
- SVM is convex, so you will always find the global optimum

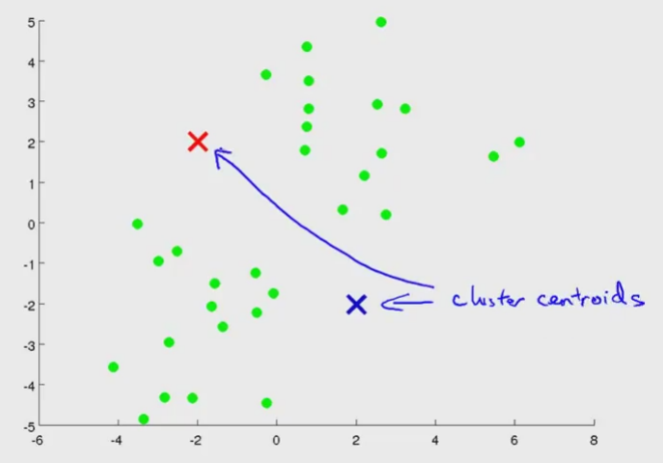
10 Clustering

Unsupervised Learning: Introduction

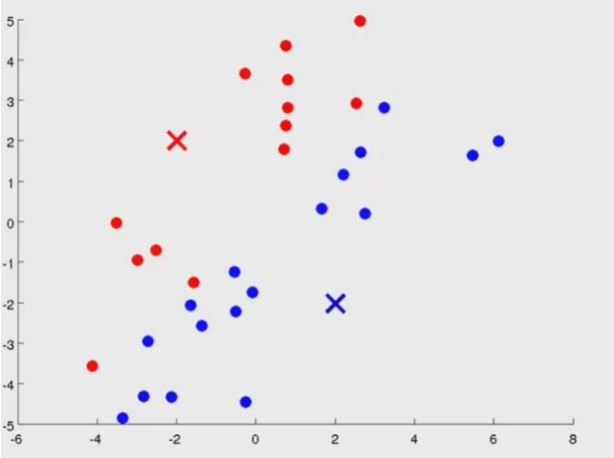
- In supervised learning we're given a training set $\{(x^{(m)}, y^{(m)})\}$, we try and find the decision boundary
- In **unsupervised learning** we're given a training set $\{x^{(m)}\}$, give it to an algorithm and try and find structure
- An example is a clustering algorithm, which finds data that close together
- Applications of clusterin: market segmentation, social network analysis, organize computing clusters, astronomical data analysis

K-Means Algorithm

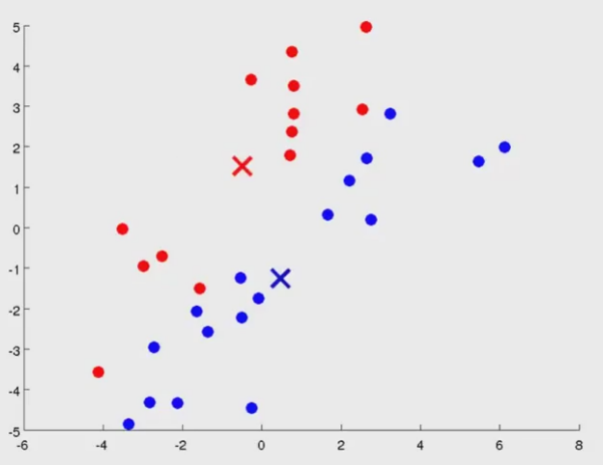
- Randomly initialize two points, called the **cluster centroids**



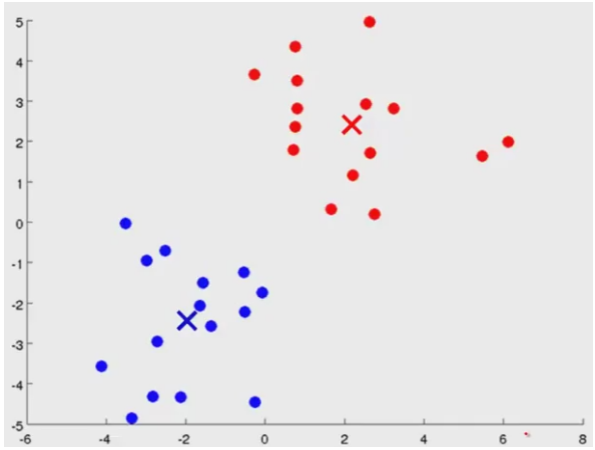
- You should have a cluster centroid for each group you want
- There's two steps: cluster assignment, and move centroid
- **Cluster assignment**: go through each example, and depending on which cluster it is closer to it will assign it to one of the centroids



- **Move centroid**: move the centroids to the mean of all the same groupped points



- Then you repeat this process until the movement of the centroids



- **K-means Algorithm**
  - Input:
    - K (number of clusters)
    - Training set $\{x^{(1)}, \ldots, x^{(m)}\}$
  - $x^{(i)} \in \mathbb{R}^n$ (drop $x_0 = 1$ convention)
  - Randomly initialize K cluster centroids $\mu_1, \ldots, \mu_k \in \mathbb{R}^n$
  - Repeat: TODO: pseudo-code markup
    - for $i = 1$ to m
      - $c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$ ($\min_k ||x^{(i)} - \mu_k||^2$)
    - for $k = 1$ to K
      - μ_k := average (mean) of points assigned to cluster k

Optimization Objective

- $\mu_c(i)$:= cluster centroid of cluster to which example $x^{(i)}$ has been asigned ($x^{(i)} \to 5$, $c^{(i)} = 5$, $\mu_{c^{(i)}} = \mu_5$)
- Optimzation Objective:

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

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

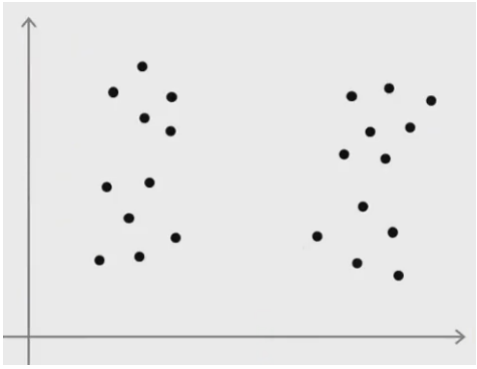
- The cluster assignment step, can be shown to be this minimization process of the cost function to the group assignment holding the centroids constant
- The moving centroids minimize with respect to the centroids

Random Initialization

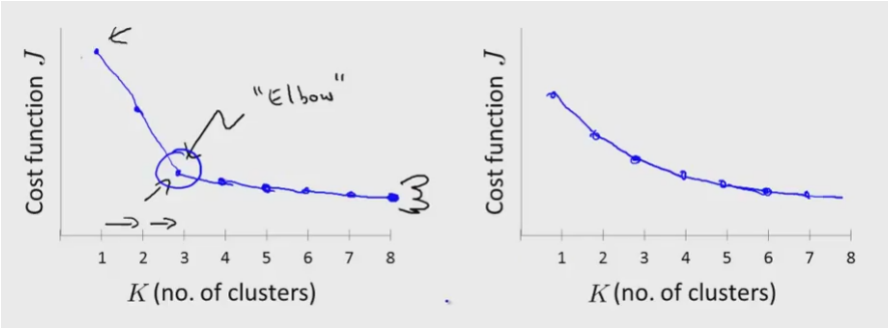
- Instead of random centroids, randomly pick K training examples, and set $\mu_1, \ldots, \mu_k$ be these K examples
- There exists local optimum, to prevent this we may run the entire K-means algorithm, and choose the lowest cost solution
- If $K = 2 - 10$, doing local random initializations to find optimal, works well; however, if K is larger then this tends to fall apart

Choosing the Number of Clusters

- The most common way is to manually choose it based on observations.
- There exists situations, where it's ambiguous the true number of clusters:



- The **elbow method** is to run K-means many times and vary the number of clusters.

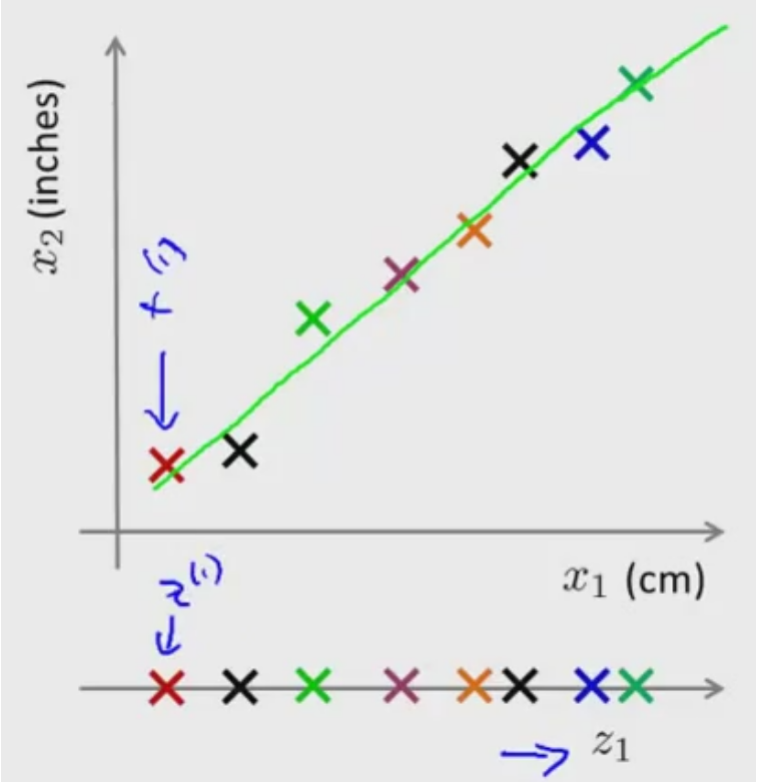


- The method looks at the plot, and takes the "elbow" as the correct value
- The elbow method isn't typically used, because the curve can be just as ambiguous
- "It's worth a shot, but don't have high expectations"
- Sometimes, you're running K-means to get clusters to ue for some later/downstream purpose. Evaluate K-means based on a metric for how well it performs for that later purpose
-

11 Dimensionality Reduction

Motivation I: Data Compression

- Lets say we are given a data set, where 2 features are redundant (inches and cm), and we want to reduce the set to only account for 1 copy of this feature
- We can think of this reduction, as specifing the distance along the line that these dimensions chart



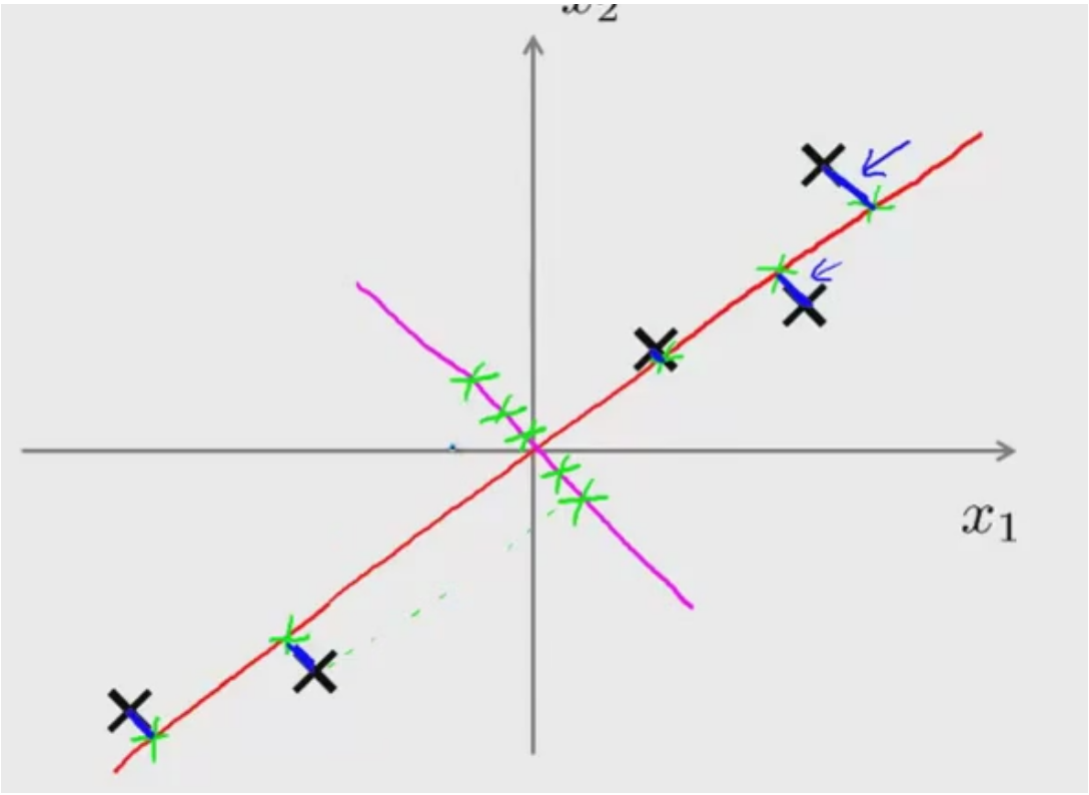
- Here z_1 represents both x_1 and x_2
- This compression can happen from m-dimension to n-dimension ($m > n$), and is not limited to just one case

Motivation II: Visualization

- For a lot of problems, we want some way to visualize and understand the data
- For example given $x^{(i)} \in \mathbb{R}^{50}$, using dimensionality reduction, we come up with a different feature representation $Z^{(i)} \in \mathbb{R}^2$ where these are some measure of the rest of the dimensions. Now we can chart this in on a plane.

Principal Component Analysis Problem Formulation

- PCA finds a lower dimensional surface on which to project the data, which minizes the projection error (distance from point to surface)



- There are bad choices (purple line) which have a very large error, and are not a good idea to reduce to, which is why we want to keep the error near zero
- **PCA**: Reduce from n-dimension to k-dimension: Find k vectors $u^{(1)}, \ldots, u^{(k)}$ onto which to project the data, so as to minimze the projection error

Principal Component Analysis Algorithm

- Data preprocessing
  - Training set: $x^{(1)}, \ldots, x^{(m)}$
  - Preprocessing (feature scaling/mean normalization);

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

  - Replace $x_j^{(i)}$ with $x_j - \mu_j$
  - If different features on different scales (eg., x_1 =size of house, x_2 = number of bedrooms), scale features to have comparable range of values
- Reduce data from n-dimensions to k-dimensions
- Compute "Covariance matrix":

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

- Compute eigenvectos of matrix σ: $[U, S, V] = svd(Sigma)$;
- The U matrix will have columns of the principal components $[u^{(i)}]$, since $U \in \mathbb{R}^{n \times n}$ we only need to take the first k columns to reduce

$$x \in \mathbb{R}^n \rightarrow z \in \mathbb{R}^k$$
$$z = [\vec{u^{(1)}}, \ldots, \vec{u^{(k)}}]^T x$$

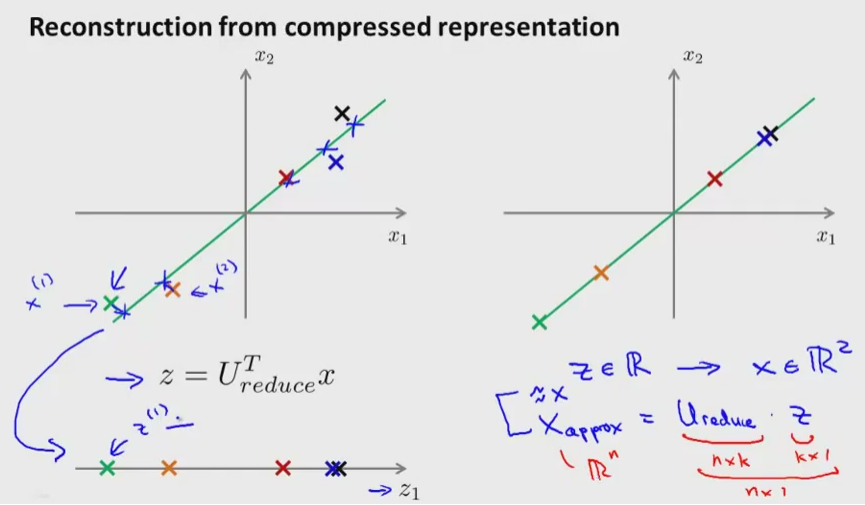
Choosing the Number of Principal Components

- Average squared projection error: $\frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - x_{approx}^{(i)}||^2$
- Total variation in the data: $\frac{1}{m} \sum_{i=1}^{m} ||x^{(i)}||^2$
- Typicall, choose k to be the smallest value so that: $\frac{\text{Average squared proj error}}{\text{Total variation in data}} \leq 0.01$
- "99% of variance is retained"
- Algorithm:
  - Try PCA with $k = 1 : n$
  - Compute $U_{reduce}, z^{(1)}, \ldots, z^{(m)}, x_{approx}^{(1)}, \ldots, x_{approx}^{(m)}$
  - Check if: ≤ 0.01
- When computing $[U, S, V]$, where S is the diagonal matrix of eigenvalues
- For a given k:

$$1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1} nS_{ii}} \leq 0.01$$

- This will give us the variability fraction like above

Reconstrction from Compressed Representation



Advice for Applying PCA

- Suppose we have $(x^{(m)}, y^{(m)})$, $x^{(i)} \in \mathbb{R}^{10000}$ like in computer vision algorithms, running a learning algorithm will be very slow
- First extract inputs:
  - Unlabeled dataset: $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^{10000}$
  - We convert via PCA into: $z^{(1)}, \ldots, z^{(m)} \in \mathbb{R}^{1000}$
- Now we have: $(z^{(1)}, y^{(1)})$, because our training examples are now represented with a lower dimension input
- Then we will learn our hypothesis: $h(z)$
- You should only reduce the training set, not the cv or test sets
- Bad use of PCA: To prevent overfitting:
  - Use $z^{(i)}$ instead of $x^{(i)}$ to reduce the number of features to $k < n$
  - Thus, fewer features, less likely to overfit
  - This might work OK, but isn't a good way to address overfitting. User regularization instead.
- Consider, doing the whole thing without using PCA
- Before implementing PCA, first try running whatever you want to do with the original/raw data $x^{(i)}$. Only if that doesn't do what you want, then implement PCA and consider using $z^{(i)}$.

12 Anomaly Detection

Problem Motivation

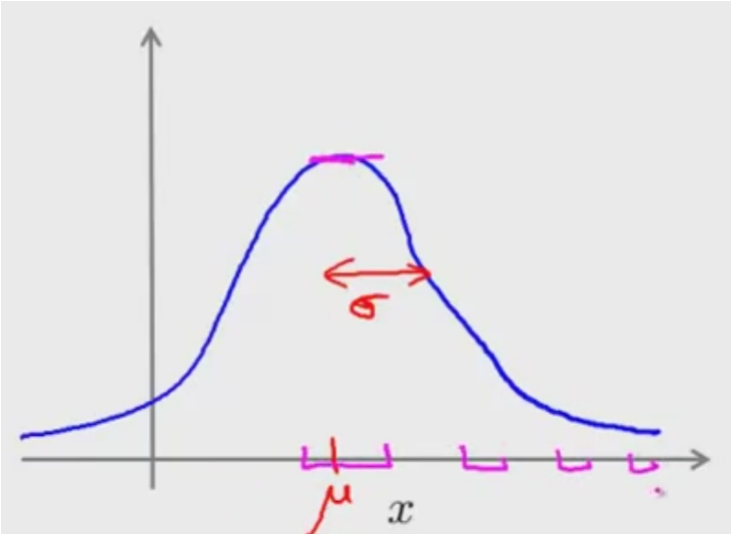
- Imagine you're an aircraft engine manufacturing, and you measure multiple features $\{x^{(1)}, \ldots, x^{(m)}\}$ off your engines
- Lets say given a new engine x_{test}, is this aircraft engine anomalous in any way. Should it undergo further testing?
- If $p(x_{test}) < \epsilon \rightarrow$ flag anomaly
- Else $p(x_{test}) \geq \epsilon$ okay
- Applications:
  - Fraud detection
    - $x^{(i)}$ features of user i's activities
    - Model $p(x)$ from data
    - Identify unusual users by checking which have $p(x) < \epsilon$
  - Manufacturing
  - Monitoring computers in a ata center

Gaussian Distribution

- Say $x \in \mathbb{R}$. If x is a distributed Gaussian with mean μ, variance σ^2.

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

- It will look like a bell shaped curve, with peak at μ:



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

-

Algorithm

- Density estimation
  - Training set $\{x^{(m)}\} \in \mathbb{R}^n$

$$p(x) = p(x_1; \mu_1, \sigma_1^2) \dots p(x_n; \mu_n, \sigma_n^2)$$

  - This corresponds to an independnce assumption (not necessary to use this algorithm)
- Algorithm:
  - Choose features x_i that you think might be indicative of anomalous examples
  - Fit parameters $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$

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

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

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

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

  - Anomaly if $p(x) < \epsilon$

Developing and Evaluating an Anomaly Detection System

- When developing a learning algorithm (choosing features, etc.), making decisions is much easier if we have a way of evaluating our learning algorithm
- Assume we have some labeled data, of anomalous and non-anomalous examples ($y = 0$ if normal, $y = 1$ if anomalous).
- Training set: $x^{(1)}, \ldots, x^{(m)}$ (assume normal examples/not anomalous)
- Cross validation set: $\{(x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})}\}$
- Test set: $\{x_{test}^{(m_{test})}, y_{test}^{(m_{test})}\}$
- Given the aircraft example: 10000 good (normal) engines, 20 flawed engines (anomalous)
  - Training set: 6000 good engines ($y = 0$)
  - CV: 2000 good engines, 10 anomalous ($y = 1$)
  - Tset: 2000 good, 10 anomalous
- Algorithnm evaluation:
  - Fit model $p(x)$ on training set $\{x^{(m)}\}$
  - On a cross validation/test example x, predict:

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

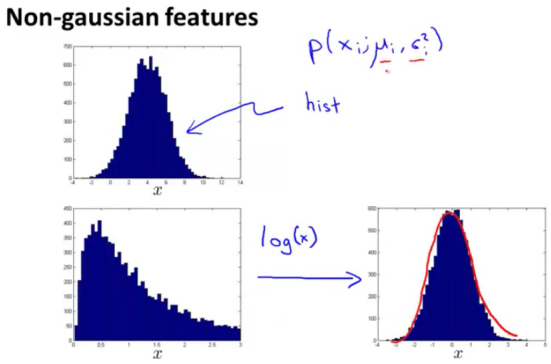
  - Possible evaluation metrics:
    - True positives, false positive, false negative, true negative
    - Precision/Recall
    - F_1 score
  - Can also use cross validation set to choose parameter ϵ

Anomaly Detection vs. Supervised Learning

- Anomaly Detection
  - Very small number of positive examples ($y = 1$). (0-20 is common).
  - Large number of negatives ($y = 0$) examples
  - Many differet "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far
- Supervised Learning:
  - Large number of positive and negative examples
  - Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely tob e similar to ones in training set
- In anomaly detection, we typically have very small postivie examples, so there's not enough to learn about them

Choosing What Features to Use

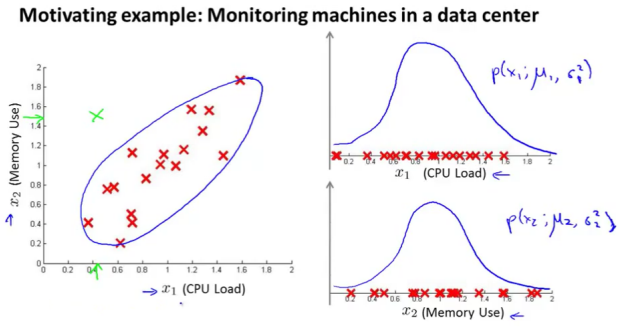
- When plotting the probabilities from the anomaly detection we hope to get a gaussian distribution:



- However, sometimes we get a skewed distribution, and it's best to attempt to transform (eg. $\log x_i + c \to x_i$) this data into a guassian distribution.
- Error analysis for anomaly detection
  - Want $p(x)$ large for normal examples x, and small for anomalous examples
  - Most common problem: $p(x)$ is comparable (say, both large) for normal and anomalous examples
  - One way to help pull out the anomaly, is to come up with another feature that makes it easier to distinguish anomalies
- Choose features that might take on unusally large or small values in the vent of an anomaly (eg. network traffic, CPU load)

Multivariate Gaussian Distribution

- Lets say we have unlabelled data:

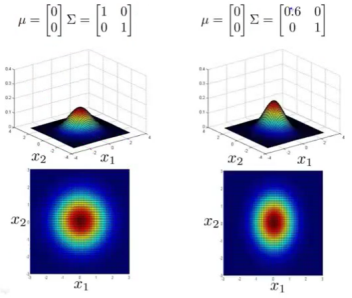


[–]

- Where we have an anomaly (green)
- This anomaly, doesn't appear to be an outlier in either 1D plot
- This keeps a 'circular' regions of probability
- $x \in \mathbb{R}^n$. Don't model $p(x_1), \ldots$ separately. Model $p(x)$ all in one go.

- Parameters: $\mu \in \mathbb{R}^n$, $\sigma \in \mathbb{R}^{n \times n}$ (covariance matrix)

$$p(x; \mu, \sigma) = \frac{1}{(2u\pi)^{n/2}|\sigma|^{0.5}} \exp(-0.5(x-\mu)^T \sigma^{-1}(x-\mu))$$



[–]

- As you decrease σ you get a smaller steeper curve, in the direction of the col/row you varied
- μ similarly models the point where the curve is centered
- You can also model correlactions with the off-diagonal entries

[–]

Anomaly Detection using the Multivariate Gaussian Distribution

- Given a training set: $\{x^{(m)}\} \in \mathbb{R}^n$

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

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

- Fit model $p(x)$ by setting these previous parameters
- Given a new example x, compute: $p(x; \mu, \sigma^2)$ Flag an anomaly if $p(x) < \epsilon$
- The original product model of Gaussian, corresponds to multivariate Gaussian where the distribution is constrained so that the contours are axis aligned
- Namely, σ has 0's on the off-diagonal entries
- When would you use each?
- **Original Model**
  - Manually create features to capture anomalies where x_1, x_2 takes unusual combinations of values (eg. $x_3 = x_1/x_2$)
  - If you're willing to spend the time creating these features, this model will work fine
  - Computationally cheaper (alternatively, scalles better to large n)
  - OK even if m (training set size) is small

- **Multivariate Gaussian**
  - Automatically captures correlations between features
  - Computationally more expensive
  - Must have $m > n$, or else σ is non-inverible
  - σ may be non-invertible if there are redundant features ($x_1 = x_2, x_3 = x_4 + x_5$), linearly dependent.

13 Recommender Systems

Problem Formulation

- Example: predicting movie ratings. Want to recommend movies that people will enjoy.
- User rates movies using zero-to-five stars

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	?

- Notation
  - n_u = no. users
  - n_m = no. movies
  - $r(i, j) = 1$ if user j has rated movie i
  - $y^{(i,j)}$ = rating given by user j to movie i (defined only if $r(i, j) = 1$)
- We want to predict what the '?' will be in our table, namely, how much they will like the movie
- In our data, we can see Alice and Bob like romance movies, and not so much action movies. We want to be able to learn from this and recommend movies intelligently based on this sort of information

Content Based Recommendations

- Say we define features based on different recommendation categories:

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	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1
Swords vs. karate	0	0	5	?	0	0.9

- With this dat we have: $n_u = 4, n_m = 5$. We also include our intercept term: $x_0 = 1$

$$x^{(1)} = \begin{bmatrix} 1 \\ 0.9 \\ 0 \end{bmatrix}$$

- For each user j, learn a paramter $\theta^{(j)} \in \mathbb{R}^3$. Predict user j as rating movei i with $(\theta^{(j)})^T x^{(i)}$ stars.
- Suppose we want to want to guess what Alice will think of "Cute puppies of love":

$$x^{(3)} = \begin{bmatrix} 1 \\ 0.99 \\ 0 \end{bmatrix}, \theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, (\theta^{(1)})^T x^{(3)} = 4.95$$

- We will discuss how we got θ later.
- We would predict that Alice would enjoy the movie
- Problem formulation:
  - $r(i,j) = 1$ if user j has rated movie i (0 otherwise)
  - $y^{(i,j)}$ = rating by user j on movie i (if defined)
  - $\theta^{(j)}$ = parameter vector for user j
  - $x^{(i)}$ = feature vector for movei i
  - For user j, movie i, predicted rating: $(\theta^{(j)})^T x^{(i)}$
  - $m^{(j)}$ = no. of movies rated by user j
  - To learn $\theta^{(j)}$ (paramter for user j):

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

  - To learn $\theta^{(1)}, \ldots, \theta^{(n_u)}$:

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

  - Gradient descent update:

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

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

- Since $m^{(j)}$ is a constant, you may exclude it to simplify the math

Collaborative Filtering

- One of the movies has a feature vector $x^{(1)}$, we can infer from the opinions (the θ) of the individuals the type of movie we are looking at
- Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, to learn $x^{(i)}$:

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

- Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$ to learn $x^{(1)}, \ldots, x^{(n_m)}$:

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

- Given $x^{(n_m)}$ (and omvie ratings), can estimate $\theta^{(n_u)}$
- Given $\theta^{(n_u)}$, can estimate $x^{(n_m)}$
- Which do we want to do first?
- Guess $\theta \to x \to \theta \to x \to \ldots$
- This will cause your features to converge via **collaborative filtering**

Collaborative Filtering Algorithm

- Collaborative filtering optimization objective:

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

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

- Algorithm:
  - Initialize $x^{(n_m)}, \theta^{(n_u)}$ to small random values
  - Minimize $J(\ldots)$ using gradient descent (or an advanced optimization algorithm) eg. for every $j = 1, \ldots, n_u, i = 1, \ldots n_m$:

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

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

  - For a user with parameters θ and a movie with (learned) features x, predict a star rating of $\theta^T x$

Vectorization - Low Rank Matrix Factorization

- Lets group the movie data into a matrix:

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

- This is what we were previously writing sa $y^{(i,j)}$
- Predicted ratings:

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

- We can denote vectorize this via:

$$X = \begin{bmatrix} - & (x^{(1)})^T & - \\ - & (x^{(2)})^T & - \\ & \vdots & \\ - & (x^{(n_m)})^T & - \end{bmatrix}, \Theta = \begin{bmatrix} - & (\theta^{(1)})^T & - \\ - & (\theta^{(2)})^T & - \\ & \vdots & \\ - & (\theta^{(n_u)})^T & - \end{bmatrix}$$

$$X\Theta^T$$

- This algorithm is also called **low rank matrix factorization**
- This matrix has a mathematical property called **low rank matrix**
- Finding related movies:
  - For each product i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$
  - Potential features: $x_1 =$ romance, $x_2 =$ action, ...
  - It's hard to truely interpret for a human in practice
  - How to find movies j related to movie i?
  - $||x^{(i)} - x^{(j)}||$ is small $\rightarrow$ movie j and i are "similar"

Implementaiton Detail - Mean Normalization

- Consider a user who has not rated any movies
- $n = 2, \theta^{(5)} \in \mathbb{R}^2$, but he has no movies which are rated
- So only the regularization term afffects $\theta^{(5)}$, since the first term in the cost function is unknown
- This will result in: $\theta^{(5)} = \vec{0}$
- This assumption of ranking all movies with a 0, doesn't seem very likely

- For **mean normalization** we will compute the average rating as so:

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

- Then we perform: $Y := Y - \mu$
- We will use these mean normalized movie ratings to learn $\theta^{(j)}, x^{(i)}$
- For user j, on movie i predict:

$$(\theta^{(j)})^T x^{(i)} + \mu_i$$

- This will result in our new user to have average ratings as a guess

14 Large Scale Machine Learning

Learning with Large Datasets

- We want to use larger datasets because it can significantly improve our models
- Learning with large datasets comes with its own problems, typically computational times
- For example: gradient descent's update term is $\mathcal{O}(m)$, for each update of θ_j
- Why not randomly choose a subset of data to train on before investing effort into creating software for large datasets
- If you plot can already show that $J_{cv} \approx J_{train}$ then you're already set, so there's no need to scale

Stochastic Gradient Descent

- Linear regression with gradient descent:

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

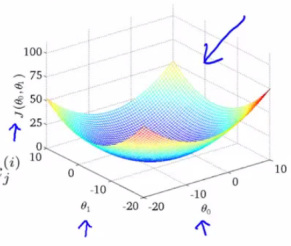
$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Repeat {

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

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

}



- The ideas we will present are fully general, but we will use linear regression to exemplify
- The previous version of gradient descent we used is called **batch gradient descent**, because we're looking at a batch of examples

- We will redefine:
$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h(x^i) - y^{(i)})^2$$
- We can now rewrite:
$$J_{train} = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$
- Stochastic gradient descent:
  - Randomly suffle dataset
  - Repeat{
    for i=1,. . .,m{
    $\theta_j := \theta_j - \alpha(h(x^{(i)}) - y^{(i)})x_j^{(i)}$ for $j = 0, \ldots, n$
    }
    }
- It's scanning through the training examples, and take a gradient descent step with the cost of just the first training example, repeating to fit a little towards each training example
- This outter loop allows multiple movements through the data set
- Batch gradient descent would make a reasonable move towards the center; however, stochastic will move more randomly as it conforms to each example
- It wanders around the global minimum, not necessarily finding it in the end; however, this shouldn't matter too much in most models in pratice
- How many times should we repeat the outer loop? 1-10x may be enough depending on the training set

Mini-Batch Gradient Descent

- Batch gradient descent: Use all m examples in each iteration
- Stochastic gradient descent: USe 1 example in each iteration
- Mini-batch gradinet descent: Use b examples in each iteration
- b = mini-batch size (Typical range: [2, 100])
- This changes our update towards:
$$\theta_j := \theta_j - \alpha\frac{1}{b}\sum_{k=i}^{i+b-1}(h(x^{(k)}) - y^{(k)}x_j^{(k)}$$
$$i := i + b$$
-

Stochastic Gradient Descent Convergence

- For batch gradient descent, we would plot J_{train} as a function of the number of iterations, ensuring that it remained decreasing
- However with stochastic, you can't check the same cost function, because it uses the slow technique that it's designed to avoid
- Instead during learning, compute $cost(\theta, (x^{(i)}, y^{(i)}))$ before updating θ using $(x^{(i)}, y^{(i)})$
- Now every so many interations (say 1000) iterations, plot *cost* averaged over the last 1000 examples processed by the algorithm
- These plots may be noisy
- As you increase the number of iterations you average, you may be able to achieve a smoother curve, but you get very little feedback
- Somtiems the cost won't decrease, but when increasing the iterations you may get more intution that is hidden by the noise
- If the curve diverges, use a smaller learning rate α
- Learning rate α is typically held constant. It can slowly decrease over time if we want θ to converge
- This may not be worthwhile since you have to tune more paramteres

Online Learning

- If you have a continuous stream of data is use an online learning algorithm to use the prefences from the stream of data to optimize some of the decisions on the site
- Considering a shipping service webstie where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping services ($y = 1$), and sometimes not ($y = 0$)
- Features x capture properties of user, of origin/destination and asking price. We want to learn $p(y = 1|x; \theta)$ to optimize price
- Repeat forever { Get (x, y) corresponding to user. Update θ using just the new example (x, y): $\theta_j := \theta_j - \alpha(h(x) - y)x_j$ for $j = 0, \ldots, n$ }
- We are discarding the notion of a fix training set, so no longer using (i) notation
- This algorithm can adapt to changing user preferences
- Another example: Product search (learning to search)
  - User searches for "android phone 1080p camera"
  - Have 100 phones in store. Will return 10 results.
  - x = features of phone, how many words in user query match name of phne, how many words in query match description of phone, etc.
  - $y = 1$ if user clicks on link. $y = 0$ otherwise

  - Learn $p(y = 1|x; \theta)$
  - Learning the CTR (click-through-rate)
  - Use to show user the 10 phones they're most likely to click on
- Other examples: choosing special offers to show user; customized selection of news articles; product recommendations ...

Map Reduce and Data Parallelism

- We will split the training set into different subsets, and process each portion on a different computer
- After each machine as computed a temporary new θ_j and combine them

$$\theta_j := \theta_j - \alpha \frac{1}{m}(temp_j^{(1)} + \ldots + temp_j^{(\ldots)} \text{ for } j = 0, \ldots, n$$

- Many learning algorithms can be expressed as computing sums of functiosn over the training set
- You can also use this on multi-core machines
- Some linear algebra libraries already do multi-core computations

15 Application Example: Photo OCR

Problem Descritpion and Pipeline

- **Photo OCR**: photo optical character recognition
- How do we get computers to understand the text that appears in images
- Photo OCR pipeline
  - Text detection (bounding box)
  - Character segmentation (segment each character within box)
  - Character classification (on each segment)
- These 'pipelines' are common, where each component may be its own machine learning component (not necessary)
- This pipeline can be one of the most important design decision

Sliding Windows

- Supervised leanring for pedestrain detection:
  - $x =$ pixels in 82x36 image patches
  - Positive examples ($y = 1$), negative ($y = 0$)
  - Where positive examples are images containing a pedestrian
  - Train a classifier to recognize pedestrians

- To check if if a pedestrian in our image, we will take the same bounding box and slide it across the first row of this size on our image
- The amount of pixels you shift the image over in this slide is called the **step-size/stride**
- Now we scale this sliding box for different sized objects of interest
- We can apply this concept to detect boxes that contain some sort of character
- We can expand our guessed areas, and then draw bounding rectangles over areas that have reasonable resolutions for words
- We may also need to train a classifier to classify between letters that are split/partial, to find the individual characters in the words

Getting Lots of Data and Artificial Data

- For character recognition, we can make fake data by taking font from the internet and overlaying it on a random background iamge
- You can apply different distortions to get even more sophisticated examples
- Another way is by taking already obtained data, and introducing distortions
- For another example you can alter audio for speech recognition by overlaying backgrounds
- Distortion introduced should be representation of the type of noise/distortions in test set
- Usually does not help to add purely random/meaningless noise to your data
- Discussion on getting more data:
  - Make sure you have a low bias classifier before expending the effort (Plot learning curves). eg. keep increasing the number of features/number of hidden units in neural network until you have a low bias classifier
  - "How much work would it be to get 10x as much data as we currently have?"
    - Artificial data synthesis
    - Collect/label it yourself (no. of hours?)
    - "Crowd source" (eg. amazon mechanical turk)

Ceiling Analysis - What Part of the Pipeline to Work on Next

- What part of the pipeline should you spend the most time trying to improve?
- In order to make decisions on developing the system, it will be helpful to have a single real-number evaluation of the system
- Feed perfect inputs into each pipeline stage and see the total improvement via this metric
- The larger the jump, the more you should focuso on that segment of the pipeline