Coursera Machine Learning by Andrew Ng

Ignavier Ng
zyingaa@connect.ust.hk

Description

- This note attempts to integrate all important knowledges taught in Coursera Machine Learning course into a single PDF file

Notation

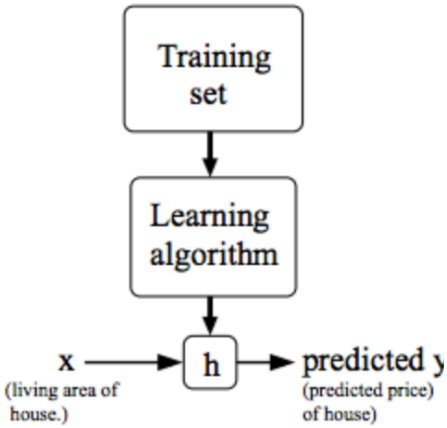
- x's = "input" variable / features
- y's = "output" variable / "target" variable
- m = Number of training examples
- n = $|x^{(i)}|$ (number of features)
- (x, y) = one training example
- $(x^{(i)}, y^{(i)})$ = i^{th} training example, $x^{(i)} \epsilon \mathbb{R}^n$ $(or\ \mathbb{R}^{n+1})$, $y^{(i)} \epsilon \mathbb{R}$
- $X = \begin{bmatrix} (x^{(1)})^T \\ \dots \\ (x^{(m)})^T \end{bmatrix}$, $Y = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(m)} \end{bmatrix}$

Machine Learning

* Content in this part focus in supervised learning, but is also applied in unsupervised learning

Hypothesis

- Given a training set, aim to learn function $h: X \rightarrow Y$ so that $h(x)$ is a "good" predictor (hypothesis) for the corresponding value of y



Underfitting (or high bias)

- Applied to both linear and logistic regression
- Happens when the hypothesis function h maps poorly to the trend of the data
- Usually caused by a function that is too simple or uses too few features
- Solution: add more features

Overfitting (or high variance)

- https://www.quora.com/Can-overfitting-occur-on-an-unsupervised-learning-algorithm
- Applied to both linear and logistic regression
- Happens when the hypothesis function h fits the available data (too well) but does not generalize well to predict new data
- Usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data
- Solution:
    - Reduce number of features
        - Manually select which features to keep
        - Use a model selection algorithm
    - Regularization
        - Keep all features, but reduce the magnitude of parameters θ_j
        - Works well when we have a lot of slightly useful features

Cost function

- Measure the accuracy of hypothesis function
- Average difference of all the results of the hypothesis with inputs from x's and the actual output y's
- Square error function / Mean squared error (most commonly used cost function):

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

  * The function is halved as convenience during computation of gradient descent

Regularization

- Formula of regularized cost function:

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

  * λ is regularization parameter which determines inflation of the cost of theta parameters
  * By convention, θ_0 is usually not regularized

- Intuition: Using above cost function with extra summation, we can smooth the output of our hypothesis function to reduce overfitting
- However, if we introduce too much regularization, we can underfit the training set and lead to worse performance even for examples not in the training set

Gradient descent

- Method to optimize θ
- Intuition: Slope of the tangent is derivative at that point and give us a direction to move forwards. We make steps down the cost function in the direction with the steepest descent

- **Batch gradient descent**
  - Use all m examples in each iteration
  - Ideal case to be applied: convex function (because it can be susceptible to local minima)
  - Algorithm (regularized version):

    Repeat until convergence {

    Simultaneously update θ_0 *and* θ_j *for* $j = 1, \dots, n$*:*

    $$\theta_0 \coloneqq \theta_0 - \alpha \frac{\partial}{\partial \theta_0} \sum_{i=1}^{m} Cost(\theta, (x^{(i)}, y^{(i)})) \quad [\theta_0 \text{ is not regularized}]$$

    $$\theta_j \coloneqq \theta_j - \alpha [\frac{\partial}{\partial \theta_j} \sum_{i=1}^{m} Cost(\theta, (x^{(i)}, y^{(i)})) + \frac{\lambda}{m} \theta_j]$$

    }

    * Substitute $\lambda = 0$ for non-regularized version
  - α is called learning rate
    - if α is too small, gradient descent can be slow
    - if α is too large, gradient descent can overshoot the minimum, may fail to converge, or even diverge
  - α is held constant. As we approach a local minimum, gradient descent will automatically take smaller steps
  - Convergence checking: Plot $J_{train}(\theta)$ as function of number of iterations, make sure it is decreasing (or at least non-increasing) with every iteration

- **Stochastic gradient descent**
  - Use 1 example in each iteration
  - Ideal case to be applied: convex function
  - Runs much faster than batch gradient descent when m is very large
  - Algorithm (regularized version):

    Randomly shuffle (reorder) training data

    Repeat until convergence {

    for $i = 1, \dots, m$ *{*

    Simultaneously update θ_0 *and* θ_j *for* $j = 1, \dots, n$*:*

    $$\theta_0 \coloneqq \theta_0 - \alpha \frac{\partial}{\partial \theta_0} Cost(\theta, (x^{(i)}, y^{(i)})) \quad [\theta_0 \text{ is not regularized}]$$

    $$\theta_j \coloneqq \theta_j - \alpha [\frac{\partial}{\partial \theta_j} Cost(\theta, (x^{(i)}, y^{(i)})) + \frac{\lambda}{m} \theta_j]$$

    }

    }
  - α is (typically) held constant. But can slowly decrease α over time if we want θ to converge

    $$\alpha = \frac{c_1}{iterationNumber + c_2}$$
  - Convergence checking:
    - During learning, compute $Cost(\theta, (x^{(i)}, y^{(i)}))$ before updating θ using $(x^{(i)}, y^{(i)})$
    - Every p (e.g. $p = 1000$) iterations, plot $Cost(\theta, (x^{(i)}, y^{(i)}))$ averaged over the last p examples processed by algorithms, make sure the trend is decreasing
  - Stochastic gradient descent usually moves the parameters in the direction of the global minimum, but not always because it often oscillates around continuously in some region close to the global minimum and ends up in local minimum

- But in practice this isn't a problem as long as the parameters end up in some region close to global minimum
- With smaller α, sometimes we can get slightly better (sometimes negligible) θ, because the oscillation will be smaller

* With well-tuned $\alpha, J_{train}(\theta)$ should decrease in each iteration of batch gradient descent but NOT necessarily stochastic gradient descent

- **Mini-batch gradient descent**
  - Use b example (e.g. $b = 10$ *or* $2\sim100$) in each iteration where $b|m$
  - Algorithm (regularized version):

Repeat until convergence {

for $i = 1, 1+b, 1+2b, \dots, m-b+1$ *{*

Simultaneously update θ_0 *and* θ_j *for* $j = 1, \dots, n$*:*

$$\theta_0 \coloneqq \theta_0 - \alpha \frac{\partial}{\partial \theta_0} \frac{1}{b} \sum_{k=i}^{i+b-1} Cost(\theta, (x^{(i)}, y^{(i)}))$$ *[θ_0 isn't regularized]*

$$\theta_j \coloneqq \theta_j - \alpha \frac{\partial}{\partial \theta_j} \frac{1}{b} \sum_{k=i}^{i+b-1} Cost(\theta, (x^{(i)}, y^{(i)})) + \frac{\lambda}{m} \theta_j]$$

}

}

  - Disadvantage: An extra parameter b which takes time to deal with
  - Advantage: Works somewhat in between batch and stochastic gradient descent. Runs even faster than both of them with good vectorization

- Techniques to choose α correctly
  - Automatic convergence test (not encouraged)
    - Declare convergence if *J(θ)* decreases by less than a constant in one iteration
    - Disadvantage: hard to choose appropriate threshold value
  - Plot $J_{train}(\theta)$ as function of number of iterations, make sure it is decreasing (or at least non-increasing) with every iteration **(encouraged)**
  - Suggestion to choose α: 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, …

Advanced Optimization

- Can be used in both linear and logistic regression
- Highly encouraged if the data set is large
- More sophisticated and faster algorithm to optimize θ:
  - Conjugate gradient
  - BFGS
  - L-BFGS

- Algorithm:

(1) For a given input value θ, provide a function that evaluates $J(\theta)$ and $\frac{\partial}{\partial \theta_j} J(\theta_j)$

```
function [jVal, gradient] = costFunction(theta)
     jVal = [_code to compute J(theta)_];
     gradient = [_code to compute derivative of J(theta)_];
end
```

(2) Use MATLAB's "*fminunc()*" or "*fmincg()*" optimization algorithm along with the "*optimset()*" function that creates an object containing the options we want to send to "*fminunc()*"

* *fmincg()* is more efficient for dealing with a large number of parameter

```
options = optimset('GradObj', 'on', 'MaxIter', 100);
initialTheta = zeros(n+1, 1);
[optTheta, functionVal, exitFlag]= fminunc(@costFunction,…
                                   initialTheta, options);0
```

Online learning

- With continuous stream of data, online learning algorithm can be effective
- Can adapt to changing user preferences
- Allows us to learn from continuous stream of data (Since we use each example once then no longer process it again)
- Algorithm (non-regularized):

Repeat forever {

Get (x, y) *corresponding to user*

Simultaneously update θ_j *for* $j = 0, \dots, n$*:*

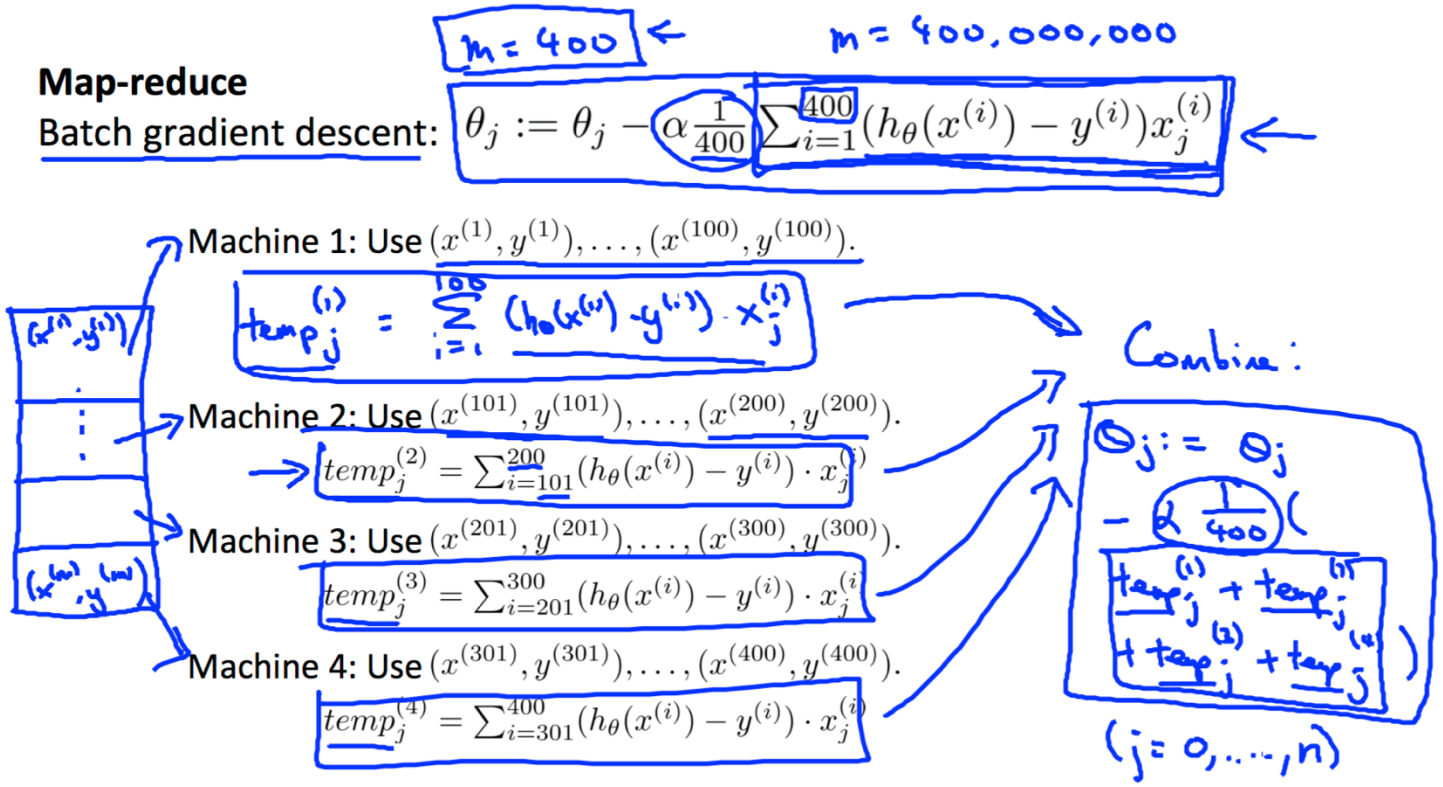
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} Cost(\theta, (x, y))$$

}

Map-reduce and data parallelism

- For learning algorithm which can be expressed as computing sums of function over training set, the computation of sum can be shared by n computers parallelly
- For n computers, if there were no network latencies and no costs of the network communications to send the data back and forth, we can potentially get up to a n times speed up
- In practice, because of network latencies, the overhead of combining the results afterwards and other factors, we get slightly less than n times speed up

- There are multiple processing cores in computers nowadays. So even in a single computer, we can split the training sets into pieces and send the training set to different cores within a single computer by using Map-reduce. As using Map-reduce within a single machine, we don't have to worry about network latency
- Some numerical linear algebra libraries can automatically parallelize their linear algebra operations across multiple cores within the machine



Supervised Learning

Properties

- Given training set (datasets (input) and their "correct answer" (output))

Can be used to solve:

- Regression problem
    - Predict results within a continuous output (real valued output)
    - Map input variables to some continuous function

- Classification problem
    - Predict results in a discrete output (discrete valued output)
    - Map input variables into discrete categories
    - Can use regression on classification problem as well, just that we need to set threshold classifier (not encouraged)

Multivariate linear regression

- Form: $h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x$
  * For convenience, we assume $x_0^{(i)} = 1$ for $i = 1, \dots, m$

- Target (regularized version): $\min_{\theta_0,\dots,\theta_n} \frac{1}{2m}[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2]$

- Gradient descent algorithm (regularized version) to compute $\theta_0, \dots, \theta_n$

  Repeat until convergence {

  Simultaneously update θ_0 and θ_j for $j = 1, \dots, n$:

  $$\theta_0 := \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad [\theta_0 \text{ is not regularized}]$$

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

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

  or use vectorized implementation

  $$\theta := (1 - \frac{\alpha}{m}\lambda)\theta - \frac{\alpha}{m}X^T(X\theta - \vec{y})$$

  $$\theta_0 := \theta_0 + \frac{\alpha}{m}\lambda\theta_0 \quad [\theta_0 \text{ is not regularized}]$$

  }

  * Intuition: We must have $(1 - \frac{\alpha}{m}\lambda) < 1$, which is intuitively equivalent to reducing value of θ_j by some amount on every update

- We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven

- Techniques to speed up gradient descent
  - **Feature scaling**
    - Divide the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1.
  - **Mean normalization**
    - Subtract the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero
  - Formula: $x_i = \frac{x_i - \mu_i}{s_i}$
  - s_i is either range of values (max - min) standard deviation

- Polynomial regression
  - We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form)

- **Normal equation** (for linear regression)
  - Method to solve for θ analytically
  - No need to do feature scaling

  - Add a first column of all 1's into *X* before implementing the formula
  - Formula (regularized version):

    $\theta = (X^T X + \lambda L)^{-1} X^T y, L = diag(0,1,1,\dots,1)$ *[θ_0 is not regularized]*
  - Derivation: http://eli.thegreenplace.net/2014/derivation-of-the-normal-equation-for-linear-regression/

- For non-regularized version, how if $X^T X$ is non invertible?
  - Reason: X is linearly dependent, due to redundant features, where two features are closely related
  - Solution: Reduce features / use `pinv` instead of `inv` in MATLAB / regularization

- For regularized version, after adding λL, $X^T X + \lambda L$ must be invertible
  - Proof: http://stats.stackexchange.com/questions/243304/how-to-prove-this-regularized-matrix-is-invertible/243307

- Difference between gradient descent and normal equation

Gradient descent	**Normal equation**
Need to choose α	No need to choose α
Needs many iterations	No need to iterate
$O(kn^2)$	$O(n^3)$ - slow in calculating inverse
Works well when n is large	Slow if n is large

Logistic regression

- Sigmoid function / Logistic function:

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

- $g(z)$ maps any real number to (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification

Binary classification (Based on logistic regression)

- $h_\theta(x)$ gives us the probability that output is 1

$$h_\theta(x) = P(y = 1|x; \theta) = 1 - P(y = 0|x; \theta)$$
$$P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1$$

- Decision boundary
  - We have:

$$h_\theta(x) = g(\theta^T x) \geq 0.5 \text{ when } \theta^T x \geq 0$$
$$h_\theta(x) = g(\theta^T x) \leq 0.5 \text{ when } \theta^T x \leq 0$$

  - Therefore:

$$\theta^T x \geq 0 \Rightarrow y = 1$$
$$\theta^T x \leq 0 \Rightarrow y = 0$$

- Cost function and its generalization
  - We cannot use the same cost function that we use for linear regression because logistic function will cause the output to be wavy, causing many local optima, which may not be convex function

  - Writing cost function in this way guarantees that $J(\theta)$ is convex for logistic regression

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

$$Cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & y = 1 \\ -\log(1 - h_\theta(x)) & y = 0 \end{cases}$$
$$= -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x)) \quad [Generalized]$$

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

* Derivation of cost function using maximum-likelihood estimation: http://math.stackexchange.com/questions/886555/deriving-cost-function-using-mle-why-use-log-function

* Regularized version:

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

  - Vectorized implementation

$$h = g(X\theta)$$
$$J(\theta) = -\frac{1}{m}(y^T \log(h) + (1 - y^T) \log(1 - h))$$

- Gradient descent algorithm to compute $\theta_0, \dots, \theta_n$:
  - Identical with algorithm in multivariate linear regression
  - Same formula and method to choose learning rate
  - Feature scaling and mean normalization can be use

  * Regularized version

Repeat until convergence {

Simultaneously update θ_0 *and* θ_j *for* $j = 1, \dots, n$*:*

$$\theta_0 \coloneqq \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad [\theta_0 \text{ is not regularized}]$$

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

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

or use vectorized implementation

$$\theta \coloneqq (1 - \frac{\alpha}{m}\lambda)\theta - \frac{\alpha}{m}X^T(g(X\theta) - \vec{y})$$

$$\theta_0 \coloneqq \theta_0 + \frac{\alpha}{m}\lambda\theta_0 \quad [\theta_0 \text{ is not regularized}]$$

}

Multiclass classification (Based on logistic regression and binary classification)

- Since $y \in \{0,1, \dots, n\}$, we divide our problem into $n + 1$ (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes

- Algorithm:
  (1) Choose one class and lump all the others into a single second class
  (2) Apply binary logistic regression, use the hypothesis returning the highest value as our prediction
  (3) Repeat step (2) for every cases

- For $i \in \{0,1, \dots, n\}$, $h_\theta^{(i)}(x)$ gives us the probability that output is i

$$h_\theta^{(i)}(x) = P(y = i|x; \theta)$$

$$Prediction_\theta = \max_{i \in \{0,1,\dots,n\}} h_\theta^{(i)}(x)$$

** $\sum_{i=0}^{n} P(y = i|x; \theta)$ does not (necessarily) equal 1

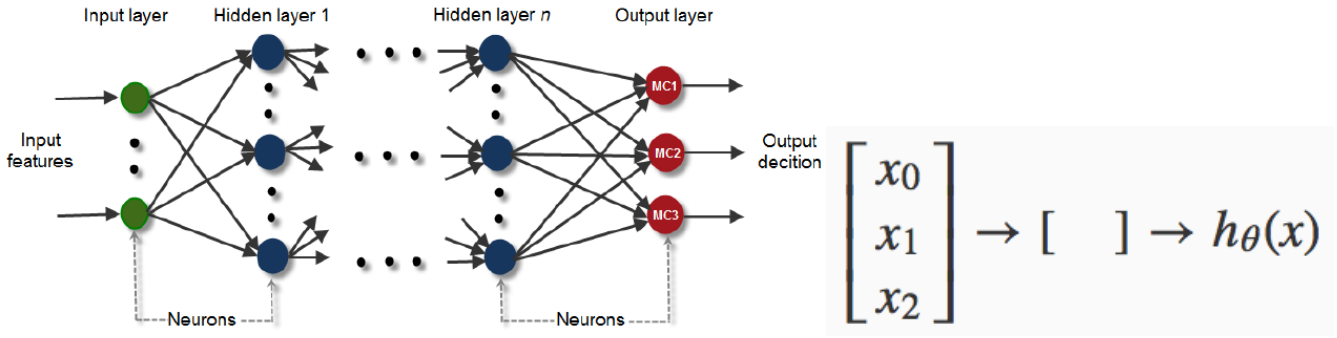
- Cost function (without regularization) (similar to binary classification):

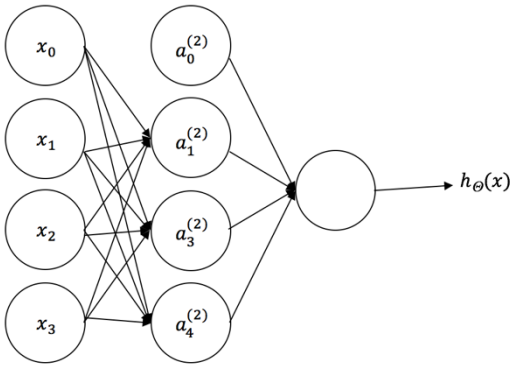
$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \log\left(Prediction_\theta(x^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - Prediction_\theta(x^{(i)})\right)]$$

Neural Network

- Background
  - Origin: algorithm that tries to mimic the neuron in brain
  - Was widely used in 80s and early 90s, popularity diminished in late 90s
  - State-of-the-art technique for many applications
  - Computationally expensive
  - The "one learning algorithm" hypothesis
    - Neural rewiring experiment
    - Different cortex learns to do different stuff
  - Non-linear (complex) hypothesis classifier

- Properties
  - Neurons are computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**)
  - Dendrites are the input features $x_1, \dots, x_n$, and output is the result of hypothesis function
  - x_0 input node is called "bias unit", which is always equal to 1
  - We use the same logistic function (sometimes it is called as sigmoid (logistic) **activation** function, where "theta" parameters are sometimes called "weights")





- Neural network model and **forward propagation**:

$a_i^{(j)}$ = activation of unit i in layer j, where x_0 and $a_0^{(j)}$ are bias units, always equal to 1

$\Theta^{(j)}$= matrix of weights controlling function mapping from layer j to layer $j+1$

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

  - Generalized implementation: $a_i^{(j)} = g(\Theta_i^{(j-1)}a^{(j-1)})$
  - Vectorized implementation: $a^{(j)} = g\left(z^{(j)}\right) = g(\Theta^{(j-1)}a^{(j-1)})$
  - If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$, where $+1$ comes from addition of bias nodes
  - The output nodes will not include bias nodes, while input nodes will
  - Remember to include bias node in every iteration

- Example of using neural network as logic gate

Let's set our first theta matrix as:

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$$

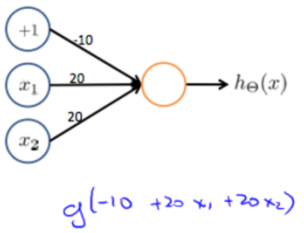
This will cause the output of our hypothesis to only be positive if both x_1 and x_2 are 1. In other words:

$$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$$

$x_1 = 0$ *and* $x_2 = 0$ *then* $g(-30) \approx 0$
$x_1 = 0$ *and* $x_2 = 1$ *then* $g(-10) \approx 0$
$x_1 = 1$ *and* $x_2 = 0$ *then* $g(-10) \approx 0$
$x_1 = 1$ *and* $x_2 = 1$ *then* $g(10) \approx 1$

So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either x_1 is true or x_2 is true, or both:

Example: OR function



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

- Architecture of neural network
  - Number of input units = dimension of features $x^{(i)}$
  - Number of output units = number of classes
  - Number of hidden units per layer = usually the more the better (must balance with cost of computation as it increases with more hidden units)
  - Default: 1 hidden layer, if >1 hidden layer, it is recommended to have same number of units in every hidden layer
    * Usually bias unit is not counted in number of unit in each layer

- Multiclass classification
  - To classify data into multiple classes, let our hypothesis function return a vector of values
  - Example: Classify images into one of three categories

    (1) Define our set of resulting classes as y, each $y^{(i)}$ represents a different category

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

    (2) Setup:

$$\begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ \cdots \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ \cdots \end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ h_\Theta(x)_3 \end{bmatrix}$$

    (3) Train neural network to obtain $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \ldots$

    (4) Substitute testing data x, the resulting hypothesis should be approximately one of:

$$h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \; or \; \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \; or \; \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$, or search for largest value in vector

- Training of neural network
  * Notation:
    - L = total number of layers in network
    - s_l = total number of units (without counting bias unit) in layer l
    - K = total number of output units / classes
    - Suppose given training sets $\{(\vec{x}^{(1)}, \vec{y}^{(1)}), \dots, (\vec{x}^{(m)}, \vec{y}^{(m)})\}$
      [Symbol of vector is used here to prevent confusion]

  (1) Randomly initialize the weights (parameter)
    - Initializing to the same values will prevent symmetry breaking
    - Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon_l, \epsilon_l]$
    - In each layer l, a good choice for ϵ_l is $\sqrt{\frac{6}{s_l + s_{l+1}}}$
    - Implementation:
      - If the dimensions of `Theta1` is $a \times b$, `Theta2` is $m \times n$

```
Theta1 = rand(a,b) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(m,n) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

  Before (2): Set $\Delta_{i,j}^{(l)} \coloneqq 0$ for all (i, j, l)

  (2) For $t = 1^{st}$ to m^{th} training data:

  a. Implement **forward propagation** to compute $h_\theta(\vec{x}^{(t)})$ and store it
     - Remember to include bias unit in every layer

  b. Implement **backward propagation** to compute partial derivatives
     - Computing direction of gradient
       i. Set $\vec{a}^{(1)} = \vec{x}^{(t)}$
       ii. Perform **forward propagation** to compute $\vec{a}^{(l)}$ for $l = 2,3, \dots, L$ $[h_\theta(\vec{x}^{(t)}) = \vec{a}^{(L)}]$
       iii. Set $\vec{\delta}^{(L)} \coloneqq \vec{a}^{(L)} - \vec{y}^{(t)}$
       iv. For $l = L - 1$ to 2, set $\vec{\delta}^{(l)} \coloneqq \left(\left(\Theta^{(l)}\right)^T \vec{\delta}^{(l+1)}\right) .* g'(\vec{z}^{(l)})$

$$= \left(\left(\Theta^{(l)}\right)^T \vec{\delta}^{(l+1)}\right) .* \vec{a}^{(l)} .* (1 - \vec{a}^{(l)})$$

         - $.*$ is elementwise multiplication
         - Remember to skip/remove $\vec{\delta}_0^{(l+1)}$ before performing $.*$
         - Do not compute $\vec{\delta}^{(0)}$ because it is in first layer (input layer) and we do not want to make any changes
         - Derivation: http://stats.stackexchange.com/questions/94387/how-to-derive-errors-in-neural-network-with-the-backpropagation-algorithm

       v. For $l = 1$ to $L - 1$, set $\Delta_{i,j}^{(l)} \coloneqq \Delta_{i,j}^{(l)} + \vec{a}_j^{(l)} \vec{\delta}_i^{(l+1)}$ for all (i, j)
         - Vectorized implementation: $\Delta^{(l)} \coloneqq \Delta^{(l)} + \vec{\delta}^{(l+1)} (\vec{a}^{(l)})^T$

After (2): Set $\frac{\partial}{\partial \Theta_{i,j}^{(l)}} = D_{i,j}^{(l)} \coloneqq \begin{cases} \frac{1}{m} \Delta_{i,j}^{(l)} & j = 0 \\ \frac{1}{m} (\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)}) & j \neq 0 \end{cases}$

* First column of $\Theta_{i,j}^{(l)}$ is used for bias units and should not be regularized

(3) Implement the cost function (using $h_\theta(\vec{x}^{(t)})$ calculated in (2))

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

* double sum = sum of logistic regression costs calculated for each cell in the output layer
* triple sum = sum of squares of all the individual Θs in the entire network
* should not include columns (first column) corresponding to bias units in triple sum

(4) Use gradient checking to ensure that backpropagation works. Then **disable** gradient checking

- Rationale:

$$\frac{\partial}{\partial\Theta_j} \approx \frac{J(\Theta_1, \dots, \Theta_j + \epsilon, \dots, \Theta_n) - J(\Theta_1, \dots, \Theta_j - \epsilon, \dots, \Theta_n)}{2\epsilon}$$

- Implementation:

```
epsilon = 1e-4;
for i = 1:n
     thetaPlus = theta;
     thetaPlus(i) += epsilon;
     thetaMinus = theta;
     thetaMinus(i) -= epsilon;
     gradApprox(i) = (J(thetaPlus) - …
                      J(thetaMinus))/(2*epsilon)
end;
```

  * Value for epsilon cannot be too small, to avoid numerical problems
- Each dimension of Θ requires two evaluations of cost function and is expensive
  - Solution: Create a small random neural network model and dataset (with a relatively small number of input units and hidden units), thus having a relatively small number of parameters

(5) Use gradient descent or built-in optimization function to minimize the cost function with the weights in theta

- Suppose we have $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \dots$ and $D^{(1)}, D^{(2)}, D^{(3)}, \dots$ and cost function
- For gradient descent, we can directly substitute these elements
- For built-in optimization function such as "`fminunc()`", we need to "unroll" all the elements into a long vector

```
thetaVector = [Theta1(:); Theta2(:); Theta3(:)]
deltaVector = [D1(:); D2(:); D3(:)]
```

- If the dimensions of `Theta1` is $a \times b$, `Theta2` is $m \times n$, `Theta3` is $y \times z$

```
Theta1 = reshape(thetaVector(1:ab),a,b)
Theta2 = reshape(thetaVector(ab+1:ab+mn),m,n)
Theta3 = reshape(thetaVector(ab+mn+1:ab+mn+yz),y,z)
```

* Ideally, we want $h_\Theta(x^{(i)}) \approx y^{(i)}$ to minimize the cost function
* $J(\Theta)$ is not convex so we can end up in a local minimum [but the result is usually good enough]

- Understanding neural network
  - Visualize what the representations captured by the hidden units
  - Given a particular hidden unit, one way to visualize what it computes is to find an input x that will cause it to activate (that is, to have an activation value $a_i^{(l)}$ close to 1)

Support Vector Machines and Kernels

(Refer to http://cs229.stanford.edu/notes/cs229-notes3.pdf)

- SVM
  - Offers computational advantage → only take support vectors into computation
  - Does not generalize to other algorithms
  - Convex → will always find global optimum
  - Hypothesis
    - $h_\theta(x)$ outputs 0 or 1 instead of probability of getting 1

      $h_\theta(x) = \begin{cases} 1 & \theta^T x \geq 0 \\ 0 & otherwise \end{cases}$
  - Decision boundary

    $$\min_\theta C \sum_{0=1}^{m} [y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$
    - Whenever $y^{(i)} = 1$, $\theta^T x^{(i)} \geq 1$
    - Whenever $y^{(i)} = 0$, $\theta^T x^{(i)} \leq -1$
    - C is equivalent to $\frac{1}{\lambda}$
    - Large C: Lower bias, high variance
    - Small C: Higher bias, low variance

- Kernel (Replace dot product in SVM with specific kernel)
  - Necessary and sufficient condition for valid kernel:
    - Corresponding kernel matrix is symmetric positive semi-definite
  - Gaussian kernel → to calculate similarity between two vectors x and $l^{(i)}$

    $f_i = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$
    - σ^2 determines how fast the similarity metric decreases (to 0)
    - Large σ^2: Features vary more smoothly. Higher bias, lower variance
    - Small σ^2: Features vary less smoothly. Lower bias, higher variance
    - MUST perform **feature scaling** before using Gaussian kernel
    - Tends to run slower
  - Some other kernels:
    - Polynomial kernel
    - String kernel
    - Chi-square kernel
    - Histogram intersection kernel

- Use SVM software package (e.g. liblinear, libsvm) to solve for parameters θ
  - Need to specify choice of C and kernel
  - No kernel → linear kernel
  - Most SVM software packages (including svmTrain.m) automatically add the extra feature $x_0 = 1$ and take care of learning the intercept term θ_0. So when passing training data to the SVM software, there is no need to add $x_0 = 1$ ourselves. In MATLAB, the code should work with training examples $x \in R^N$ rather than $x \in R^{N+1}$

- Logistic regression vs SVM
  - n is large (relative to m) → logistic regression or SVM with linear kernel
  - n is small, m is intermediate → SVM with Gaussian kernel
  - n is small, m is large → Add more features, then use logistic regression or SVM with linear kernel
  - Neural network is likely to work well for most of these settings, but may be slower to train

Unsupervised learning

Properties

- Given datasets (input) without “correct answer” (output)

Can be used to solve:

- Problems with little or no idea what our results should look like
- Derive structure from data where we don't necessarily know the effect of the variables
    - by clustering the data based on relationships among the variables in the data
    - there is no feedback based on the prediction results.
- e.g. cocktail party algorithm

K-means algorithm

- Input:
  - K (number of clusters)
  - Training set $\{x^{(1)}, \dots, x^{(m)}\}$, $x^{(i)} \in \mathbb{R}^n$ (drop $x_0 = 1$ convention)
- $c^{(i)}$ = index of cluster $(1,2, \dots, K)$ to which example $x^{(i)}$ is currently assigned
- μ_k = cluster centroid k ($\mu_k \in \mathbb{R}^n$)
- $\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned

- Optimization objective:

$$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k) = \frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \mu_{c^{(i)}}\right\|^2 \quad \text{[Cost function, distortion]}$$

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

- Algorithm:

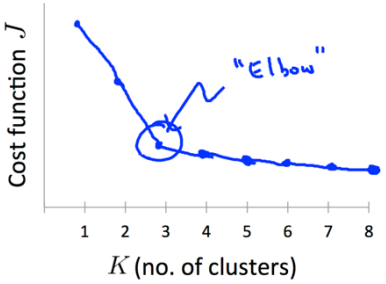
Random initialize K cluster centroids $\mu_1, \dots, \mu_k \in \mathbb{R}^n$
Repeat {
  for $i = 1$ *to* m*:*
    $c^{(i)} :=$ *index (from* 1 *to* K*) of cluster centroid closest to* $x^{(i)}$
    [Cluster assignment step: $\min_{c^{(1)},\dots,c^{(m)}} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k)$*]*
  for $k = 1$ *to* K*:*
    $\mu_k :=$ *average (mean) of points assigned to cluster* k
    [Centroid moving step: $\min_{\mu_1,\dots,\mu_k} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k)$*]*
}

* This algorithm will possibly fall into local optima because $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k)$ is non-convex
* $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k)$ must **decrease** over each iterations

- Solution: Repeat the algorithm for p (e.g. $p = 100$ times)
* The lower the number of cluster, the large (better) the difference this solution will help

for $i = 1$ *to* p *{*
  Randomly initialize K-means
  Run K-means to get $c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k$
  Compute cost function (distortion):
    $J_i(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k)$
}
Pick clustering method i: $\underset{i}{\operatorname{argmin}}\, J_i(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k)$

- Method to choose K:
  - Elbow method
  - Run K-means to get clusters for some later/downstream purpose. Evaluate K-means based on a metric for how well it performs for that later purpose



Principal Component Analysis (PCA)

- Reduce from n-dimension ($x^{(i)} \in \mathbb{R}^n$) to k-dimension ($z^{(i)} \in \mathbb{R}^k$): Find k vectors $u^{(1)}, \dots, u^{(k)}$ onto which to project the data, so as to minimize the projection error
- PCA projection can be thought of as a rotation that selects the view maximizing the spread of data
- Perform dimensionality reduction —> reduce the redundancy of data
- PCA is not linear regression

- **PCA Algorithm** (Do not add bias term $x_0 = 1$ in X):
  (1) Mean normalization and (optionally) feature scaling
  (2) Compute covariance matrix, $\sum = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)})(x^{(i)})^T$

```
Sigma = (1/m)* X' * X;
```

  (3) Compute eigenvector of $\sum$:

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

  * `SVD` is more numerically stable than `eig`

  We get:

$$U = [u^{(1)} \quad \dots \quad u^{(n)}] \in \mathbb{R}^{n \times n}$$
$$S = diag(S_{11}, S_{22}, \dots, S_{nn})$$

  Compute $z^{(i)} \in \mathbb{R}^k$:

$$U_{Reduce} = [u^{(1)} \quad \dots \quad u^{(k)}] \in \mathbb{R}^{n \times k}$$
$$z^{(i)} = {U_{Reduce}}^T x^{(i)}$$

  Vectorized implementation:

```
Ureduce = U(:, 1:k);
z = Ureduce' * X;
```

- **Reconstruction** from compressed representation:

$$x_{Approx}^{(i)} = U_{Reduce}\, z^{(i)}$$

- Method to choosing k (Number of principal components)
  - Average squared projection error:

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

  - Total variation in the data:

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

  - We choose a threshold percentage p as the **percentage of retained variance**
  - $p \in [0.95, 0.99]$ is the most common range of values
  - Choose k to be smallest value such that:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\| x^{(i)} - x_{Approx}^{(i)} \right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\left\| x^{(i)} \right\|^2} \leq 1 - p$$

    * Equivalent form:

$$1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \leq 1 - p \;\Leftrightarrow\; \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \geq p$$

  - To retain high percentage of variance, we can often reduce dimension of data significantly and still retain most variance due to high correlation between many features in most real life data

- Application
  - Speedup supervised learning algorithm
    - Training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$
    - **Define** mapping from every $x^{(i)}$ to $z^{(i)}$
    - New training set $\{(z^{(1)}, y^{(1)}), \dots, (z^{(m)}, y^{(m)})\}$
    - This mapping can be **applied** to $x_{cv}^{(i)}$ and $x_{test}^{(i)}$
  - Compression
    - Reduce memory/disk needed to store data
  - Visualization
    - 2D graph (k = 2) or 3D graph (k = 3)
    - Visualizing datasets in 3 dimensions or greater can be cumbersome
    - It is often desirable to display the data in 2D some information will be lost

- Bad use of PCA
  - Prevent overfitting by reducing $x^{(i)}$ to $z^{(i)}$ by having less features
    - This might work OK, but isn't a good way to address overfitting
    - Because PCA will throw away much valuable data (e.g. the labelled y)
    - Should use regularization instead because it takes y into account

- Remark: Before reducing dimension of data using PCA, we should first consider not using PCA on original data. Only if there is reason to believe that doesn't work, and the learning algorithm ends up running too slowly, or only if the memory requirement or the disk space requirement is too large, then we only want to compress the representation

Anomaly detection

- http://cs229.stanford.edu/notes/cs229-notes2.pdf

- **Gaussian distribution** (normal distribution)
  - A special case of multivariate Gaussian distribution
  - $x \sim N(\mu, \sigma^2)$
  - $P(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x-\mu)^2}{2\sigma^2})$
  - Algorithm (Training set $\{x^{(1)}, \dots, x^{(m)}\}$):
    (1) Choose features x_i that might be indicative of anomalous examples
    (2) Fit parameters $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$
    $$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)},\ \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu)^2$$
    * In machine learning, we use denominator as m instead of $m-1$
    (3) Compute $P(x)$ for new example x
    $$P(x) = \prod_{j=1}^{n} P(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2})$$
    * Theoretically this is an independent assumption between each feature, but the algorithm works fine even if they are not all independent
    Output: Flag x as anomaly if $P(x) < \varepsilon$

  - Evaluation of algorithm
    - Fit model $P(x)$ on training set
    - On a CV/test set example x, predict
      $$y = \begin{cases} 1 & if\ P(x) < \varepsilon\ (anomaly) \\ 0 & if\ P(x) \geq \varepsilon\ (normal) \end{cases}$$
    - Possible evaluation metrics: $F_1\ score$
    - Can use CV set to choose appropriate ε by maximizing $F_1\ score$

  - Remarks:
    - Case 1: Given only a few anomalous examples and many non-anomalous examples
      - Use non-anomalous data in training set to fit parameters in step (2)
      - Use anomalous data in CV and test set
    - Case 2: Given some labeled data and much unlabeled data
      - Assume unlabeled data is non-anomalous, because the number of non-anomalous data should be far larger than anomalous data
      - Use unlabeled data in training set to fit parameters in step (2)
      - Use labeled data in CV and test set

  - For non-Guassian features x, we can transform it into Gaussian features by:
    - $x' = (c_1 x + c_2)^{c_3}$
    - $x' = log(c_1 x + c_2)$

  - Error analysis for anomaly detection
    - Want $P(x)$ large for normal examples x and small for anomalous examples x
    - Most common problems: $P(x)$ is comparable (say, both large) for normal and anomalous examples
    - Choose features that might take on unusually large or small values in the event of an anomaly

- **Multivariate Gaussian distribution**
  - $x \sim N(\mu, \sum)$
  - $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \in \mathbb{R}^n$
  - $\sum = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)} - \mu)(x^{(i)} - \mu)^T \in \mathbb{R}^{n \times n}$ is covariance matrix
  - $P(x; \mu, \sum) = \frac{1}{(2\pi)^{0.5n}|\sum|^{0.5}} \exp(-\frac{1}{2}(x - \mu)^T \sum^{-1}(x - \mu))$

  - Algorithm (Training set $\{x^{(1)}, \ldots, x^{(m)}\}$):
    (1) Choose features x_i that might be indicative of anomalous examples
    (2) Compute μ and $\sum$
    (3) Compute $P(x; \mu, \sum)$ for new example x
    Output: Flag x as anomaly if $P(x; \mu, \sum) \quad < \varepsilon$

- Difference between Gaussian distribution and multivariate Gaussian distribution

Gaussian distribution	**Multivariate Gaussian distribution**
Manually create features to capture anomalies (by combination of different features)	Automatically capture correlations between features
Computationally cheaper (scales better to large dataset)	Computationally expensive
OK even if m is small	Must have $m > n$ or else $\sum$ is non-invertible

- Difference between anomaly detection and supervised learning

Anomaly detection	**Supervised learning**
Very small number of positive examples ($y = 1$) (0-20 is common) Large number of negative examples ($y = 0$)	Large number of positive and negative examples
Many different "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	Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set
Fraud detection, manufacturing, monitoring machines in data center	Email spam classification, weather prediction, cancer classification

Machine learning system design

Machine learning pipeline

- A system with many stages/components, several of which may use machine learning, or sometimes it may not be a machine learning component but to have a set of modules that act one after another on some piece of data in order to produce the output you want

Training, validation and test

- A learning algorithm fits a training set well does not mean it is a good hypothesis. It could over fit and so our predictions on the test set could be poor
- The error of hypothesis as measured on the training set will be lower than the error on any other data set
- One way to break down our dataset into the three sets is:
  - Training set: 60%
  - Cross validation set: 20%
  - Test set: 20%

- For error of linear regression:

$$Validation\ Error = J_{CV}(\Theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\Theta\left(x_{CV}^{(i)}\right) - y_{CV}^{(i)})^2$$

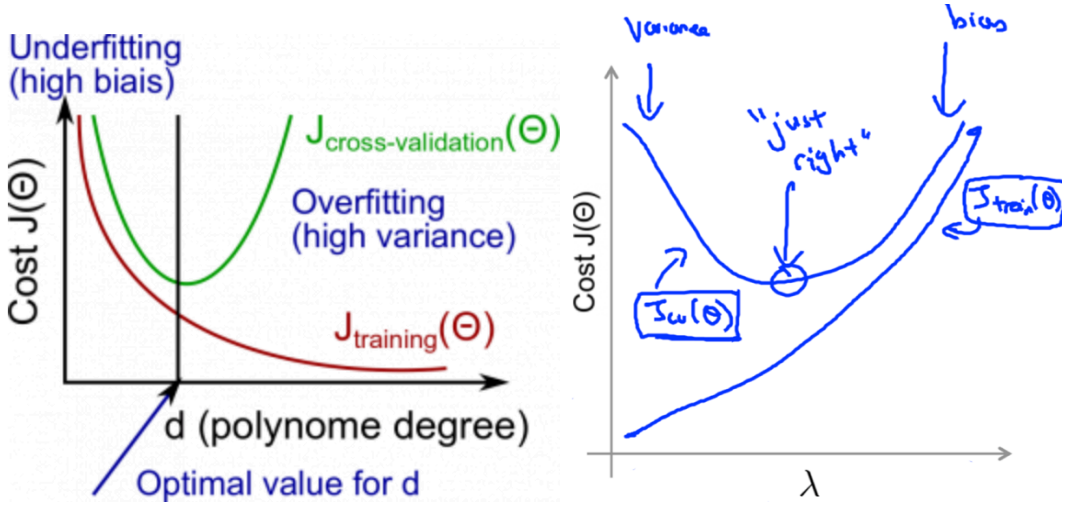
- For misclassification error of logistic classifier:

$$err(h_\Theta(x), y) = \begin{cases} 1 & if\ [h_\Theta(x) \geq 0.5\ and\ y = 0]\ or\ [h_\Theta(x) < 0.5\ and\ y = 1] \\ 0 & otherwise \end{cases}$$

$$Validation\ Error = J_{CV}(\Theta) = \frac{1}{m_{cv}} \sum_{i=1}^{m_{cv}} err\left(h_\Theta\left(x_{CV}^{(i)}\right), y_{CV}^{(i)}\right)$$

  * The same formula applies for test error
- Due to randomness in the training and validation splits of the dataset, the cross validation error can sometimes be lower than the training error

Bias/Variance

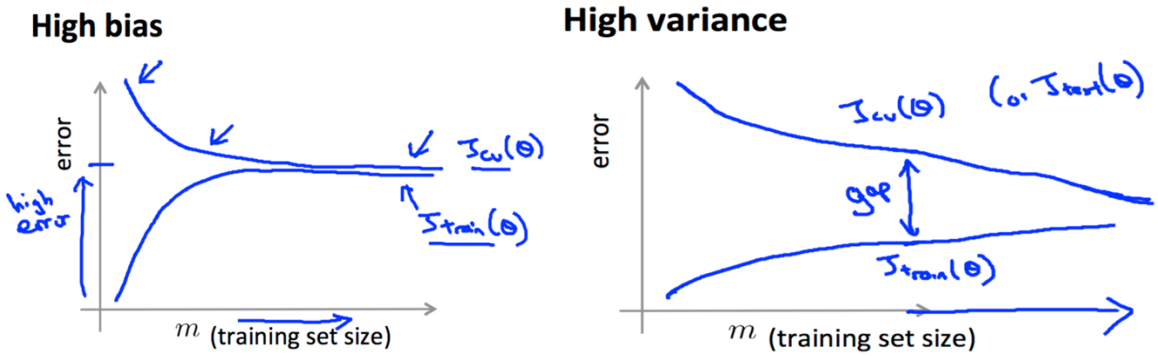


- Degree of polynomial and bias/variance
  - Ideally, we want to find a golden mean between high bias and high variance
  - High bias
    - $J_{CV}(\Theta) \approx J_{train}(\Theta)$
    - Both $J_{CV}(\Theta)$ and $J_{train}(\Theta)$ will be high
  - High variance
    - $J_{CV}(\Theta) \gg J_{train}(\Theta)$
    - $J_{CV}(\Theta)$ will be high
    - $J_{train}(\Theta)$ will be low

- Regularization and bias/variance
  - Not helpful for θ of low dimension
  - As λ increases, our fit becomes more rigid.
  - As λ approaches 0, we tend to over overfit the data

- Model selection (Degree of polynomial and regularization parameter):
  (0) Split dataset into training, cross validation and test data
  * If data set comes in sorted order, must shuffle before splitting
  (1) Create a list of lambdas, i.e. $\lambda \in \{0, 0.01, 0.02, 0.04, \dots, 5.12, 10.24\}$
  (2) Create a set of models with different degrees or any other variants
  (3) Iterate through the λs and for each λ go through all the models to learn some Θ
  (4) Learn the parameter Θ for the model selected, using $J_{train}(\Theta)$ with the λ selected
  (5) Compute the <u>train error</u> using the learned Θ (computed with λ) on the $J_{train}(\Theta)$ **without** regularization or $\lambda = 0$
  (6) Compute the <u>cross validation error</u> using the learned Θ (computed with λ) on the $J_{CV}(\Theta)$ **without** regularization or $\lambda = 0$
  (7) Select the best combo that produces the lowest error on the cross validation set
  (8) Using the best combo Θ and λ, apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem
  * This way, the degree of the polynomial d and λ has not been trained using the test set
  * $J_{CV}(\Theta)$ is expected to be lower than $J_{test}(\Theta)$ because two extra parameters (d and λ) has been fit to cross validation set

- **Learning curve**



  - High variance: $J_{CV}(\Theta) \gg J_{train}(\Theta)$
  - If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.
  - If a learning algorithm is suffering from high variance, getting more training data is likely to help
  - In practice, especially for small training sets, when plotting learning curves to, it is helpful to average (e.g. 50 times) across multiple sets of randomly selected examples to determine the training error and cross validation error

- Debugging a machine learning system with bias/variance
  - Get more training examples → fixes high variance
  - Try smaller sets of features → fixes high variance
  - Get additional features → fixes high bias
  - Add polynomial features → fixes high bias
  - Decrease λ → fixes high variance
  - Increase λ → fixes high bias

- Neural network and bias/variance
  - Small neural network
    - Computationally cheaper
    - Fewer parameters
    - More prone to underfitting
  - Large neural network
    - Computationally expensive
    - More parameters
    - More prone to overfitting
  - Usually, use larger neural network and more regularization to adjust overfitting is more effective than using small neural network
  - Default: 1 hidden layer, if want more hidden layers, try for 1, 2, 3… and calculate $J_{CV}(\Theta)$

Recommended approach (Identifying error)

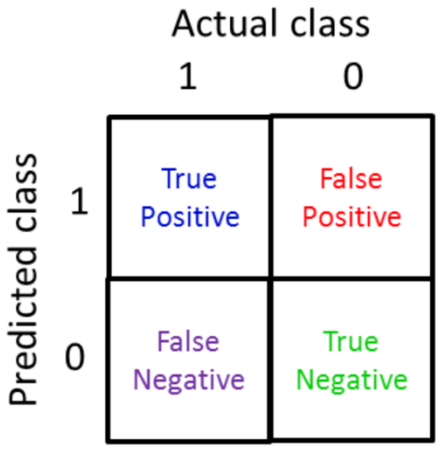
(1) Start with simple (quick) algorithm. Implement and test it on cross validation data [> 24 hours]
(2) Plot learning curves to decide if more data, more features, etc. are likely to help
(3) **Error analysis**: Manually examine the examples (in cross validation set) that algorithm made errors on. Try to spot any systematic trend in what type of examples it is making errors on
* Error analysis should be performed on cross validation set instead of test set
Reason: If we develop new features by examining test set, we may end up choosing features that works well specifically for test set, so $J_{test}(\Theta)$ is no longer a good estimate of how well we generalize to new examples

Error metric

- Single rolled number evaluation metric is no longer a good indicator
- $Precision,\ P = \frac{True\ Pos}{Predicted\ Pos} = \frac{True\ Pos}{True\ Pos + False\ Pos}$
- $Recall,\ R = \frac{True\ Pos}{Actual\ Pos} = \frac{True\ Pos}{True\ Pos + False\ Neg}$
- In logistic classifier:
  - Predict 1 if $h_\theta(x) \geq threshold$
  - Predict 0 if $h_\theta(x) < threshold$
- High threshold → high precision, low recall
- Low threshold → low precision, high recall

Performance of machine learning algorithm

- ~~$Accuracy = \frac{True\ Pos + True\ Neg}{Total\ Num}$~~ [Not a good indicator]
- ~~$Average = \frac{P+R}{2}$~~ [Not a good indicator]
- $F_1\ Score = \frac{PR}{P+R}$
- A good classifier should have both high precision and high recall on the cross validation set
- For logistic classifier, we can try a range of threshold value, evaluate them on cross validation set and choose one that maximizes the $F_1\ Score$



Ceiling analysis

- In machine learning pipelines, assume 100% accuracy in previous pipelines to figure out the upper bound of accuracy up to current pipeline
- The differences between 100% accuracy and actual accuracy help us to figure out which pipeline we should spend most time to improve

Big data rationale

- Using a very large training set makes it unlikely for model to overfit the training data
- More training data only helps when both followings are true:
    - We train a learning algorithm with a large number of parameters
      (That is able to learn/represent fairly complex functions)
    - The features x contain sufficient information to predict y accurately
      (One way to verify this is if a human expert on the domain can confidently predict y when given only x)
- One of the most reliable ways to get a high performance machine learning system is to take a low bias learning algorithm (plot learning curves) and to train it on a massive training set
    - E.g. keep increasing the number of features/number of hidden units in neural network until we get a low bias classifier.
- Often ask: "How much work would it be to get 10x as much data as currently have?"
    - Artificial data synthesis
    - Collect/label manually
    - Crowd source (e.g. Amazon Mechanical Turk)

Artificial data synthesis

- Introduce distortion within representation of the type of noise/distortions in the test set
- Usually does not help to add purely random/meaningless noise to data

Application of Supervised Learning

Recommender systems

- Problem formulation
  - n_u = number of users
  - n_m = number of movies
  - $m^{(j)}$ = number of movies rated by user j
  - $r(i,j) = 1$ if user j has rated movie i (0 otherwise)
  - $y^{(i,j)}$ = rating by user j on movie i (defined only if $r(i,j) = 1$)
  - $\theta^{(j)}$ = parameter vector for user $j \in \mathbb{R}^{n+1}$
  - $x^{(i)}$ = feature vector for movie $i \in \mathbb{R}^{n+1}$

  - For user j, movie i, predicted rating: $\left(\theta^{(j)}\right)^T x^{(i)}$, so we want to learn $\theta^{(j)}$ for every person
  - Optimization objective (Given $x^{(1)}, \dots, x^{(n_m)}$, to learn $\theta^{(1)}, \dots, \theta^{(n_u)}$):

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

  - Gradient descent update:

$$\theta_0^{(j)} := \theta_0^{(j)} - \alpha \sum_{i:r(i,j)=1}\left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right) x_0^{(i)}$$
$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1}\left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right) x_k^{(i)} + \lambda\theta_k^{(j)}\right) \quad [\text{for } k \neq 0]$$

- **Collaborative filtering**
  - It is impossible for human to watch every movie and construct its feature vector $x^{(i)}$
  - Given $\theta^{(1)}, \dots, \theta^{(n_u)}$, we can estimate $x^{(1)}, \dots, x^{(n_m)}$:

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

  - Combining (1) and (2), minimize $x^{(1)}, \dots, x^{(n_m)}$ and $\theta^{(1)}, \dots, \theta^{(n_u)}$ simultaneously:

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

    * Bias term $x_0 = 1$ is not needed because the algorithm has the flexibility to learn by itself. If the algorithm wants a feature equals to 1, it can choose to learn one for itself (e.g. $x_1 = 1$)

  - Algorithm:
    (1) Initialize $x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}$ to small random values [Symmetry breaking]
    (2) Minimize $J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)})$ using gradient descent algorithm (or advanced optimization algorithm)

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

    (3) For a user j with parameters $\theta^{(j)}$ and a movie i with (learned) features $x^{(i)}$, predict a rating of $\left(\theta^{(j)}\right)^T x^{(i)}$

- Low rank matrix factorization
  - Let Y be the matrix storing rating by user j on movie i
  - $Y = \left[y^{(i,j)}\right] = \left[\left(\theta^{(j)}\right)^T x^{(i)}\right]$
  - Let $X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \dots \\ \left(x^{(n_m)}\right)^T \end{bmatrix}, \Theta = \begin{bmatrix} \left(\theta^{(1)}\right)^T \\ \dots \\ \left(\theta^{(n_u)}\right)^T \end{bmatrix}, x^{(i)} \in \mathbb{R}^{n\times 1},\ \theta^{(j)} \in \mathbb{R}^{n\times 1}$
  - We get: $Y = X\Theta^T$

- Mean normalization
  - Use $Y' = \left[y^{(i,j)} - \mu_i\right]$ instead to learn Θ and X
  - For a user j with parameters $\theta^{(j)}$ and a movie i with (learned) features $x^{(i)}$, predict a rating of $\left(\theta^{(j)}\right)^T x^{(i)} + \mu_i$

  * Feature scaling is not needed because product ratings should be on similar scale

- Construct recommendation
  - For each product i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$
  - Suggest few movies j: $\underset{1\le j\le n_m, j\neq i}{\operatorname{argmin}} \left\|x^{(i)} - x^{(j)}\right\|$

Photo Optical Character Recognition (OCR)

- Sliding windows
  - Applied in text detection (2D sliding) and character segmentation (1D sliding)
  - Slide the window (with specific size) further to the right and run that patch through the classifier again, until this window slides over different locations in the image. The amount we shift the rectangle each time is **slide parameter**
  - Increase the size of window and repeat previous step again
  - Before classifying in each iteration, we need to resize the image to the size of images in classifier

- Text detection
  - For detection of some objects such as pedestrians, pedestrians can be different distances away from the camera and so the height of these rectangles can be different depending on how far away they are, but the **aspect ratio** is the same
  - For text detection the height and width ratio is different for different lines of text

  - Algorithm:
    (1) Perform sliding windows algorithms (2D sliding) to classify (e.g. use neural network or logistic classifier) whether a window is text region
    (2) Use white to show where the classifier thinks it has found text, different shades of grey correspond to the probability output by classifier
    (3) Apply **expansion operator** on the output of classifier, which takes each of the white regions in image and expands that white region. Mathematically, for every pixel in image, if it is within some distance (e.g. 5∼10 pixels) of some other white pixel, then color that pixel white. By this, some possible connected components can be formed
    (4) Look at the connected components and white regions, draw bounding boxes around them. Use a simple heuristic algorithm to rule out rectangles whose aspect ratios is weird (because width of bounding boxes around text is much larger than their height)
    (5) Draw rectangles around the ones whose aspect ratio looks like text regions

- Character segmentation
  - Perform sliding windows algorithms (1D sliding) to classify (e.g. use neural network or logistic classifier) whether a window can be split into 2 characters to get the locations for splitting the rectangles into single character

- Character classification
  - Apply a supervised learning (e.g. neural network or logistic classifier) to take an image as input and classify which alphabet or numerical digits it is