Machine Learning : Introduction

What is Machine Learning?

1. "the field of study that gives computers the ability to learn without being explicitly programmed."
2. A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E

In general, any machine learning problem can be assigned to one of two broad classifications:

Supervised Learning

In supervised learning, we are given a data set and already know what our correct output should look like.

Supervised learning problems are categorized into "regression" and "classification" problems.

In a **regression problem**, we are trying to predict results within a continuous output.

Example : Given data about the size of houses on the real estate market, try to predict their price.

In a **classification problem**, we are instead trying to predict results in a discrete categories.

Example : Banks have to decide whether or not to give a loan to someone on the basis of his credit history.

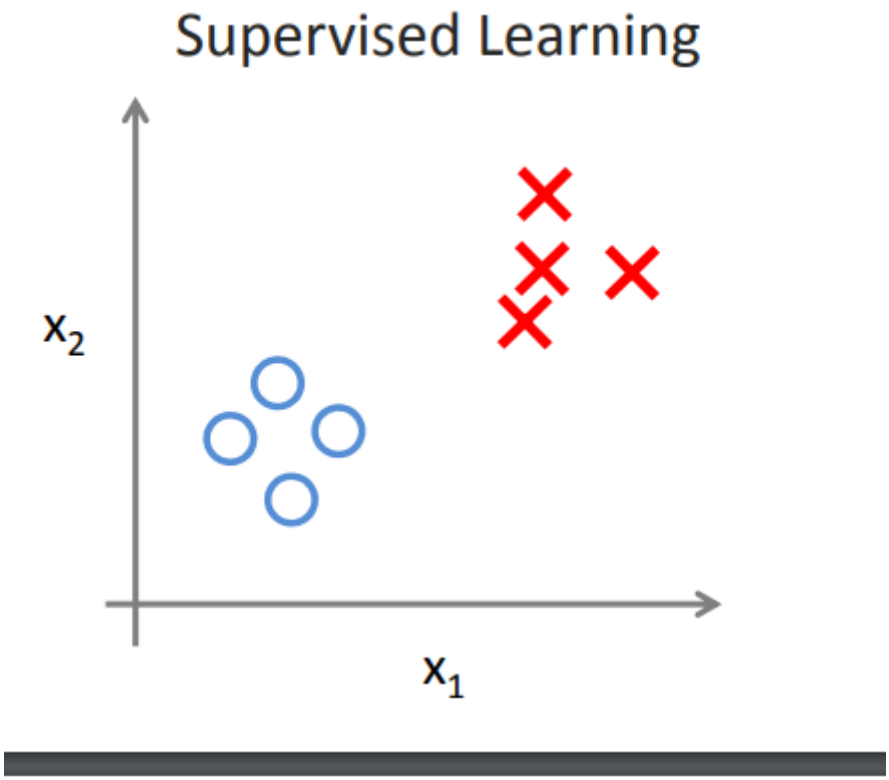
Unsupervised Learning

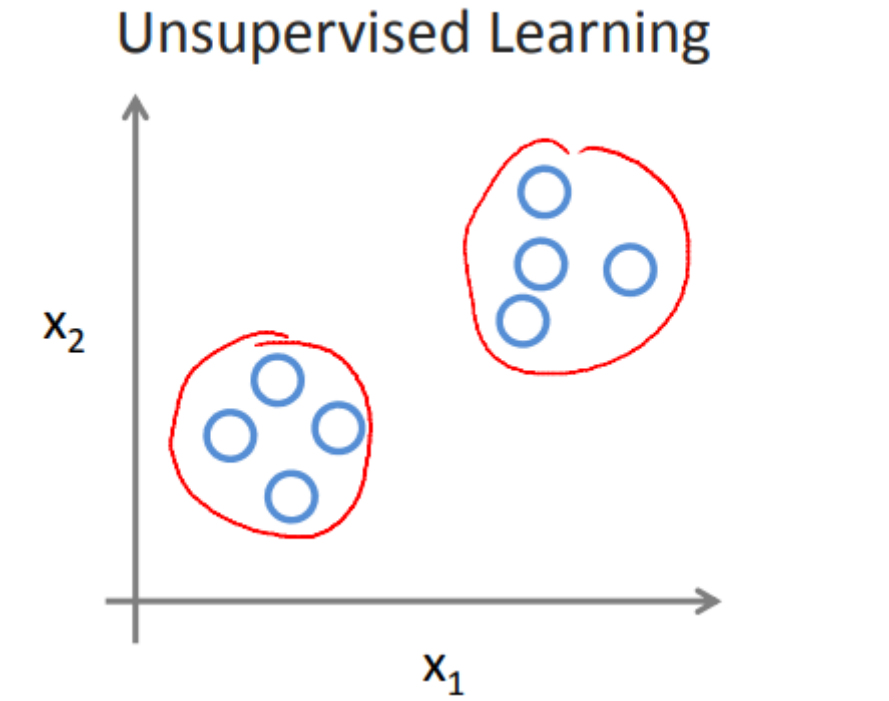
Unsupervised learning allows us to approach problems with little or no idea what our results should look like. the dataset is not labelled

We can derive structure from data where we don't necessarily know the effect of the variables.

We can derive this structure by clustering the data based on relationships among the variables in the data.

There is no feedback based on the prediction result


Supervised Learning
x_2
x_1


Unsupervised Learning
x_2
x_1

The Hypothesis Function

Our hypothesis function has the general form:

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x$$

Note that this is like the equation of a straight line. We give to $h_\theta(x)$ values for θ_0 and θ_1 to get our estimated output $\hat{y}$. In other words, we are trying to create a function called h_θ that is trying to map our input data (the x's) to our output data (the y's).

Cost Function

We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's compared to the actual output y's.

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

To break it apart, it is $\frac{1}{2}\ \bar{x}$ where $\bar{x}$ is the mean of the squares of $h_\theta(x_i) - y_i$, or the difference between the predicted value and the actual value.

This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved $\left(\frac{1}{2m}\right)$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.

Gradient Descent

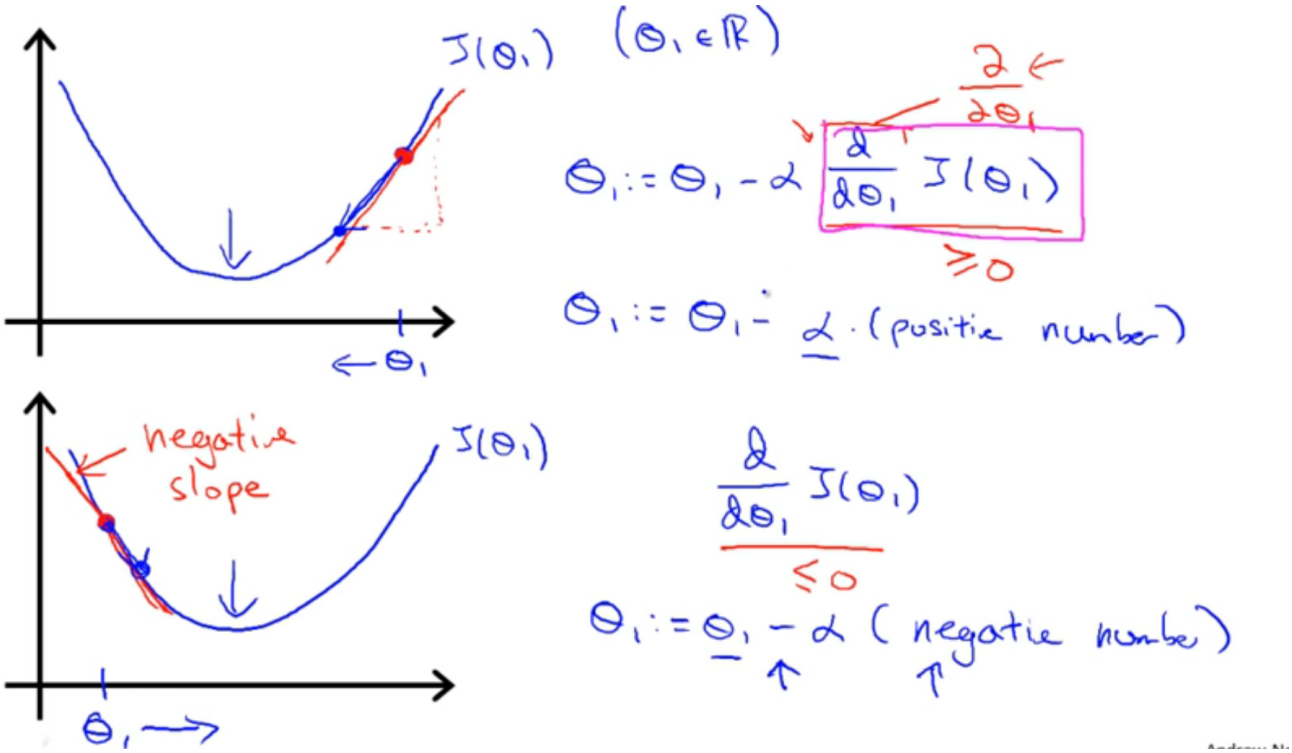
Gradient descent is an **optimization algorithm** used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. **In machine learning, we use gradient descent to update the parameters of our model** to minimize the **Cost function**. Parameters refer to coefficients in Linear Regression and weights in neural networks

Have some function $J(\theta_0, \theta_1)$

Want $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

Outline:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a minimum



REGRESSION

Gradient Descent for Linear Regression

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for $j = 1$ and $j = 0$)

}

Linear Regression Model

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

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

Andrew Ng

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

$$= \frac{2}{2\theta_j} \frac{1}{2m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2$$

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

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

Andrew Ng

Gradient descent algorithm

repeat until convergence {

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

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x^{(i)}$$

}

update θ_0 and θ_1 simultaneously

$\frac{2}{2\theta_0} J(\theta_0, \theta_1)$

$\frac{2}{2\theta_1} J(\theta_0, \theta_1)$

Linear Regression with Multiple Variables

Hypothesis: $h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$

$x_0 = 1$

Parameters: $\theta_0, \theta_1, \ldots, \theta_n$ θ $n+1$-dimensional vector

Cost function:

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

$J(\theta)$

Gradient descent:

Repeat {

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

$J(\theta)$

}

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

Andrew Ng

Gradient Descent

Previously (n=1):

Repeat {

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

$$\frac{\partial}{\partial \theta_0} J(\theta)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}$$

$x_1^{(i)}$

(simultaneously update θ_0, θ_1)

}

New algorithm $(n \geq 1)$:

Repeat {

$\frac{\partial}{\partial \theta_j} J(\theta)$

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

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

}

$x_0^{(i)} = 1$

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

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)}$$

...

Andrew Ng

Feature Scaling - Normalization

Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without **normalization**. For example, many classifiers calculate the distance between two points by the **Euclidean** distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

Another reason why feature scaling is applied is that few algorithms like Neural network gradient descent converge much faster with feature scaling than without it.

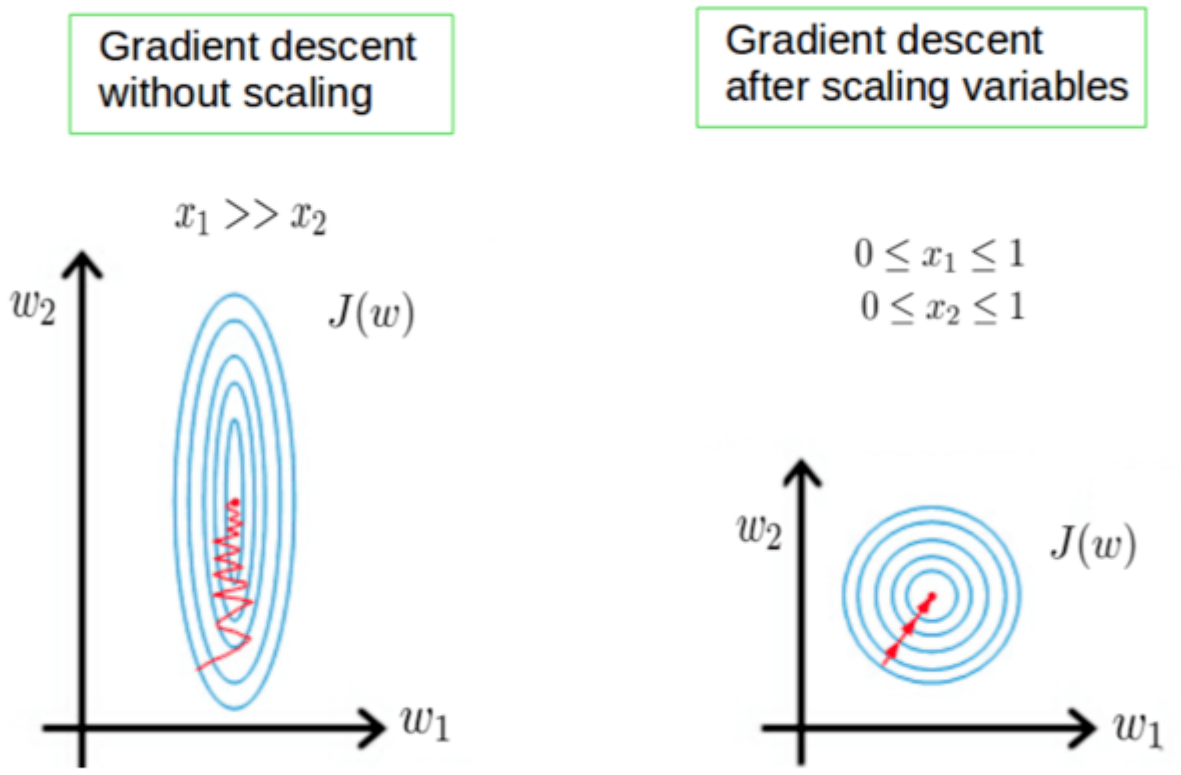
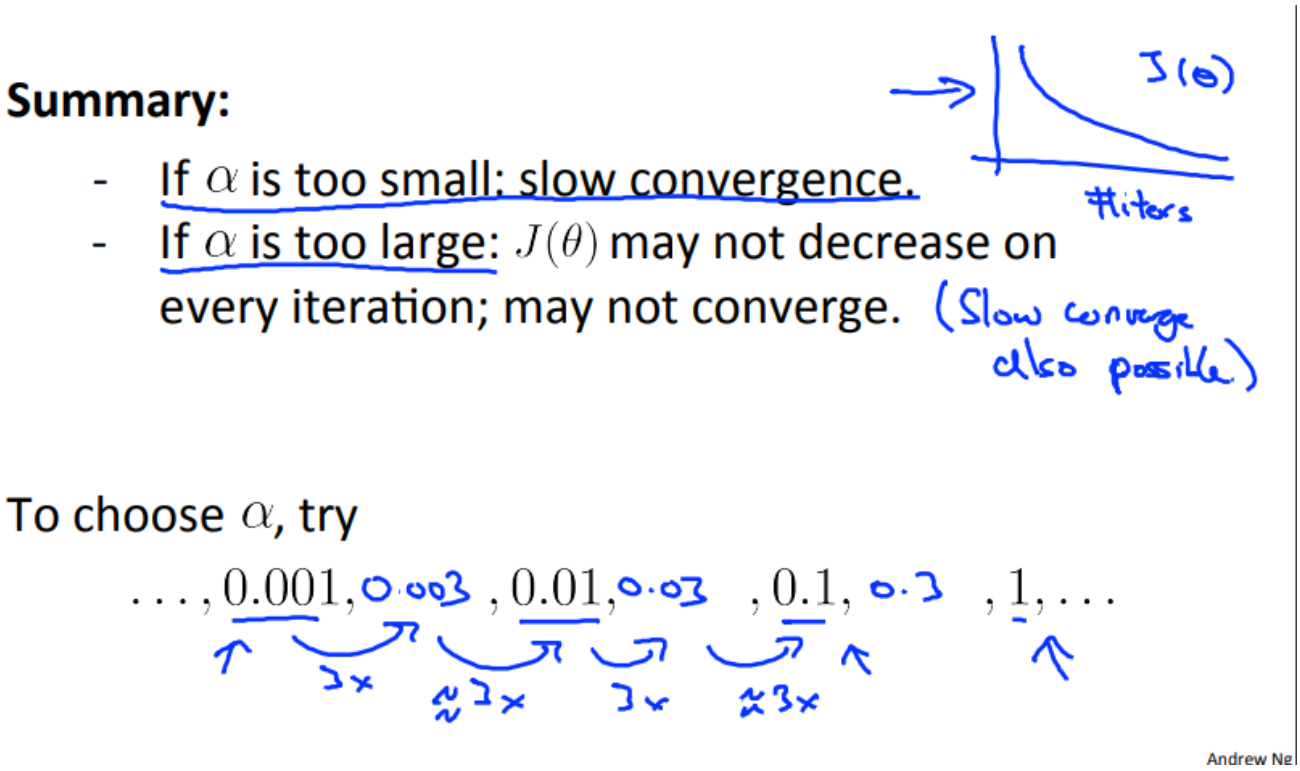


Photo Credit

How to choose learning rate



CLASSIFICATION

Logistic Regression

Don't be confused by the name "Logistic Regression"; it is named that way for historical reasons and is actually an approach to classification problems, not regression problems.

Binary Classification

Instead of our output vector y being a continuous range of values, it will only be 0 or 1.

Hypothesis Representation

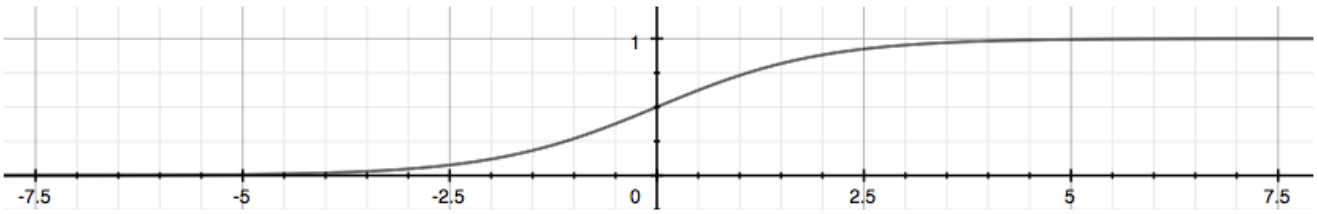
We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that y ∈ {0, 1}. To fix this, let's change the form for our hypotheses $h_\theta(x)$ to satisfy $0 \leq h_\theta(x) \leq 1$. This is accomplished by plugging $\theta^T x$ into the Logistic Function.

Our new form uses the "Sigmoid Function," also called the "Logistic Function":

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

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

The following image shows us what the sigmoid function looks like:



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

$h_\theta(x)$ will give us the **probability** that our output is 1. For example, $h_\theta(x) = 0.7$ gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

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

Decision Boundry

Decision Boundary

In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$$h_\theta(x) \geq 0.5 \rightarrow y = 1$$
$$h_\theta(x) < 0.5 \rightarrow y = 0$$

The way our logistic function g behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5:

$$g(z) \geq 0.5$$
$$when\ z \geq 0$$

Remember.

$$z = 0, e^0 = 1 \Rightarrow g(z) = 1/2$$
$$z \rightarrow \infty, e^{-\infty} \rightarrow 0 \Rightarrow g(z) = 1$$
$$z \rightarrow -\infty, e^{\infty} \rightarrow \infty \Rightarrow g(z) = 0$$

So if our input to g is $\theta^T X$, then that means:

$$h_\theta(x) = g(\theta^T x) \geq 0.5$$
$$when\ \theta^T x \geq 0$$

From these statements we can now say:

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

The **decision boundary** is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function.

Example:

$$\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix}$$
$$y = 1\ if\ 5 + (-1)x_1 + 0x_2 \geq 0$$
$$5 - x_1 \geq 0$$
$$-x_1 \geq -5$$
$$x_1 \leq 5$$

In this case, our decision boundary is a straight vertical line placed on the graph where $x_1 = 5$, and everything to the left of that denotes y = 1, while everything to the right denotes y = 0.

Again, the input to the sigmoid function g(z) (e.g. $\theta^T X$) doesn't need to be linear, and could be a function that describes a circle (e.g. $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$) or any shape to fit our data.

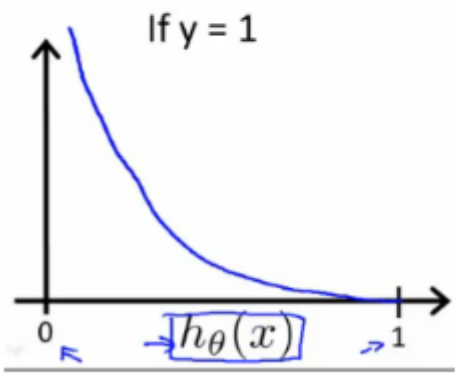
Cost Function

We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

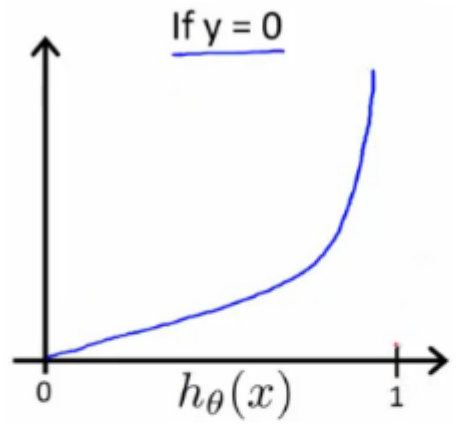
Instead, our cost function for logistic regression looks like:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$
$$\text{Cost}(h_\theta(x), y) = -\log(h_\theta(x)) \quad \text{if } y = 1$$
$$\text{Cost}(h_\theta(x), y) = -\log(1 - h_\theta(x)) \quad \text{if } y = 0$$

When y = 1, we get the following plot for $J(\theta)$ vs $h_\theta(x)$:



Similarly, when y = 0, we get the following plot for $J(\theta)$ vs $h_\theta(x)$:



$$\text{Cost}(h_\theta(x), y) = 0 \text{ if } h_\theta(x) = y$$
$$\text{Cost}(h_\theta(x), y) \to \infty \text{ if } y = 0 \text{ and } h_\theta(x) \to 1$$
$$\text{Cost}(h_\theta(x), y) \to \infty \text{ if } y = 1 \text{ and } h_\theta(x) \to 0$$

If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

Note that writing the cost function in this way guarantees that J(θ) is convex for logistic regression.

Gradient Descent

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

Notice that when y is equal to 1, then the second term $(1 - y) \log(1 - h_\theta(x))$ will be zero and will not affect the result. If y is equal to 0, then the first term $-y \log(h_\theta(x))$ will be zero and will not affect the result.

We can fully write out our entire cost function as follows:

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

A vectorized implementation is:

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

Gradient Descent

Remember that the general form of gradient descent is:

$$\begin{aligned} &Repeat \ \{ \\ &\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &\} \end{aligned}$$

We can work out the derivative part using calculus to get:

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

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in theta.

A vectorized implementation is:

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

Multiclass Classification: One-vs-all

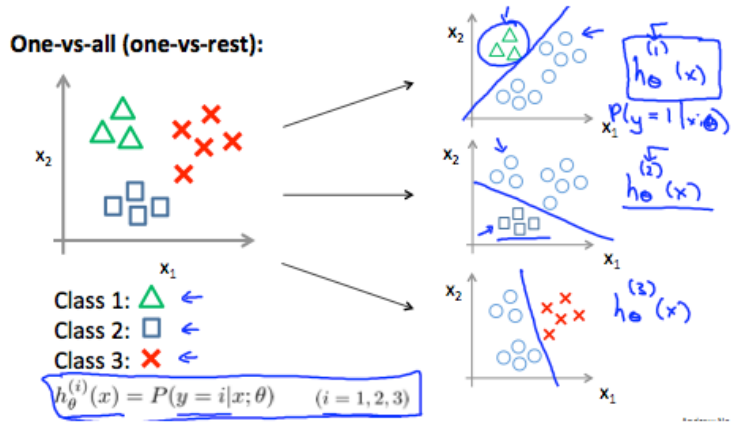
Now we will approach the classification of data when we have more than two categories. Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

Since y = {0,1...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.

$$
\begin{aligned}
&y \in \{0, 1 \dots n\} \\
&h_\theta^{(0)}(x) = P(y = 0 | x; \theta) \\
&h_\theta^{(1)}(x) = P(y = 1 | x; \theta) \\
&\cdots \\
&h_\theta^{(n)}(x) = P(y = n | x; \theta) \\
&\text{prediction} = \max_i (h_\theta^{(i)}(x))
\end{aligned}
$$

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

The following image shows how one could classify 3 classes:



To summarize:

Train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that y = i.

To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$

Regularization

- The Problem of Overfitting

Regularization is designed to address the problem of overfitting.

High bias or underfitting is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. eg. if we take $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$ then we are making an initial assumption that a linear model will fit the training data well and will be able to generalize but that may not be the case.

At the other extreme, overfitting or high variance is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1) Reduce the number of features:

a) Manually select which features to keep.

b) Use a model selection algorithm (studied later in the course).

2) Regularization

Keep all the features, but reduce the parameters θj

Regularization works well when we have a lot of slightly useful features.

Cost Function If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

Say we wanted to make the following function more quadratic:

$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

We'll want to eliminate the influence of $\theta_3 x^3$ and $\theta_4 x^4$

Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our cost function:

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

We've added two extra terms at the end to inflate the cost of θ_3 and θ_4

Now, in order for the cost function to get close to zero, we will have to reduce the values of $\theta_3 \theta$ and θ_4 to near zero. This will in turn greatly reduce the values of $\theta_3 x^3$ and $\theta_4 x^4$ in our hypothesis function.

We could also regularize all of our theta parameters in a single summation:

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

The λ, or lambda, is the **regularization parameter**. It determines how much the costs of our theta parameters are inflated. You can visualize the effect of regularization in this interactive plot : https://www.desmos.com/calculator/1hexc8ntqp

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting.

Regularized Linear Regression

We can apply regularization to both linear regression and logistic regression. We will approach linear regression first.

Gradient Descent

We will modify our gradient descent function to separate out θ_0 from the rest of the parameters because we do not want to penalize θ_0.

$$\begin{aligned}&\text{Repeat } \{\\ &\quad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}\\ &\quad \theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j\right] \qquad j \in \{1,2...n\}\\ &\}\end{aligned}$$

The term $\frac{\lambda}{m}\theta_j$ performs our regularization.

With some manipulation our update rule can also be represented as:

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

The first term in the above equation, $1 - \alpha\frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_j by some amount on every update.

Notice that the second term is now exactly the same as it was before.

Regularized Logistic Regression

Neural Networks

Neural networks are limited imitations of how our own brains work.

Model Representation

At a very simple level, neurons are basically computational units that take input (dendrites) as electrical input (called "spikes") that are channeled to outputs (axons).

In our model, our dendrites are like the input features $x_1 \cdots x_n$, and the output is the result of our hypothesis function:

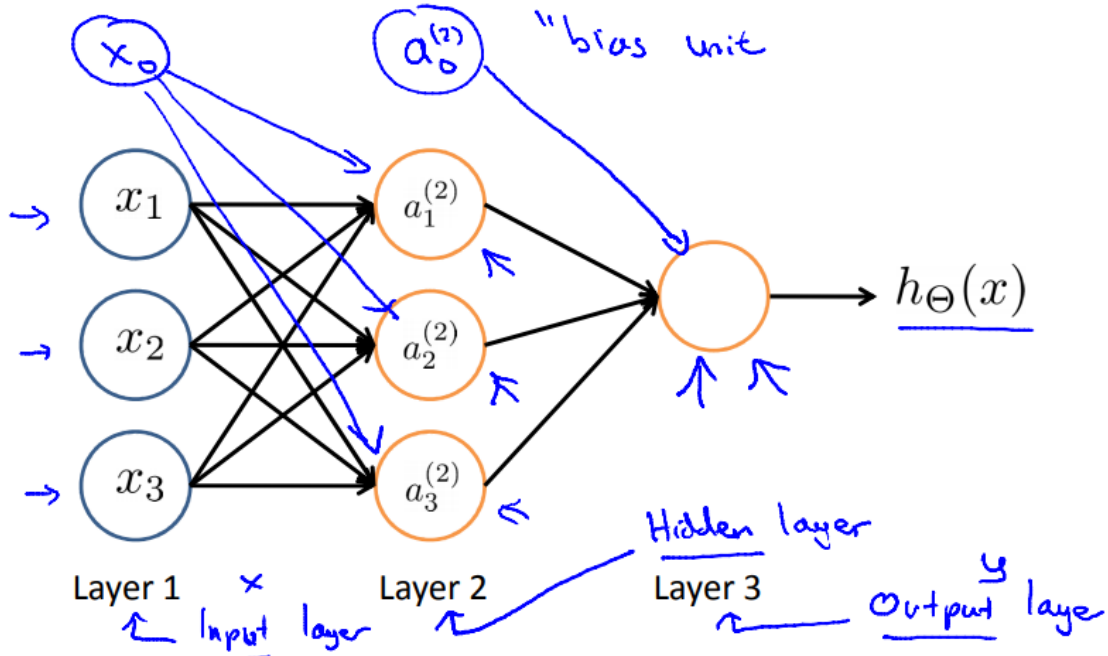
In this model our x0 input node is sometimes called the "bias unit." It is always equal to 1.

In neural networks, we use the same logistic function as in classification: $\frac{1}{1+e^{-\theta^T x}}$.

In neural networks however we sometimes call it a sigmoid (logistic) activation function.

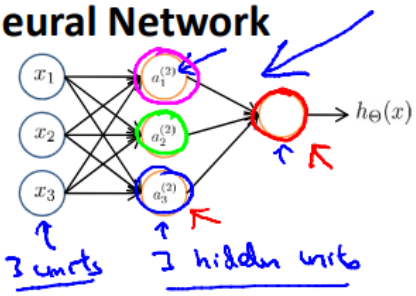
Our "theta" parameters are sometimes instead called "weights" in the neural networks model.

Neural Network



Andrew Ng

Neural Network



$a_i^{(j)}$ = "activation" of unit i in layer j

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

$\Theta^{(1)} \in \mathbb{R}^{3\times 4}$ $h_\Theta(x)$

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

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$\Theta^{(2)}$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$. $s_{j+1} \times (s_j + 1)$

Andrew Ng

In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replaced the variable z for all the parameters we would get:

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

In other words, for layer j=2 and node k, the variable z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)}x_0 + \Theta_{k,1}^{(1)}x_1 + \cdots + \Theta_{k,n}^{(1)}x_n$$

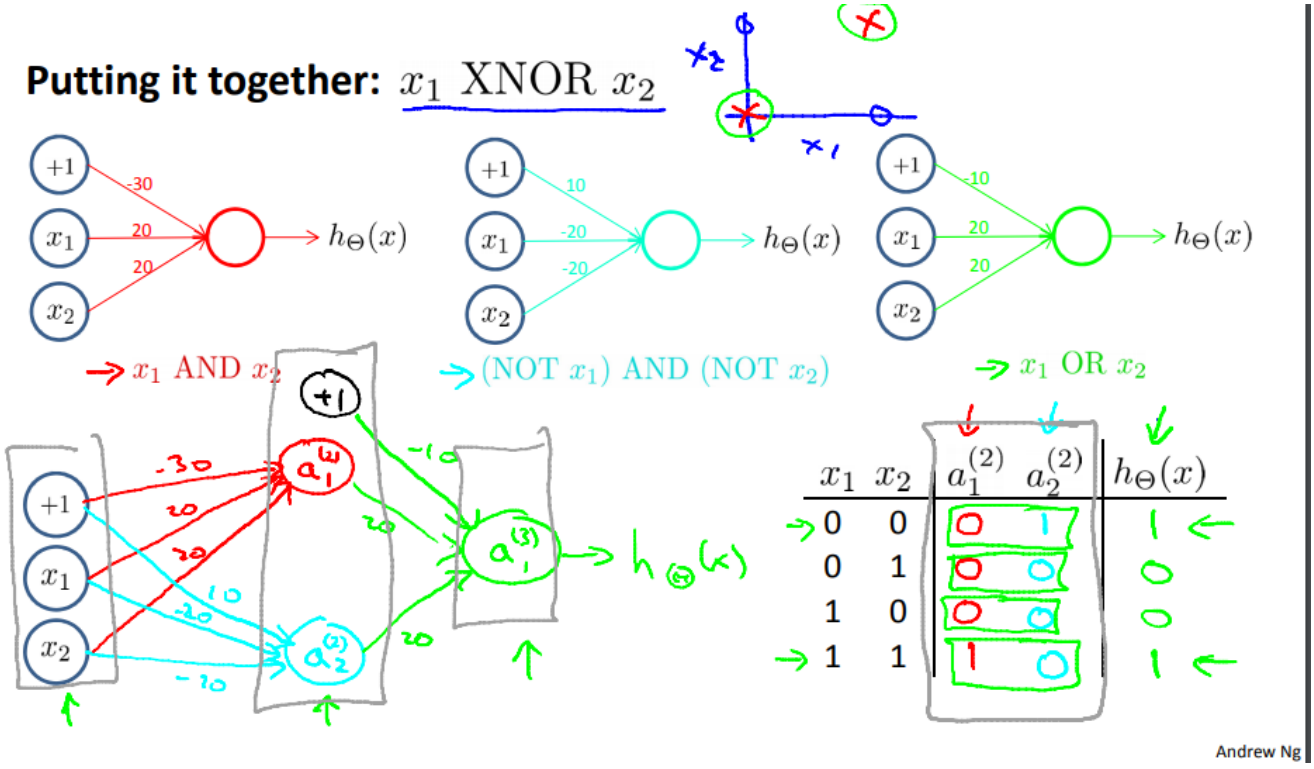
The vector representation of x and z^j is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \quad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \cdots \\ z_n^{(j)} \end{bmatrix}$$

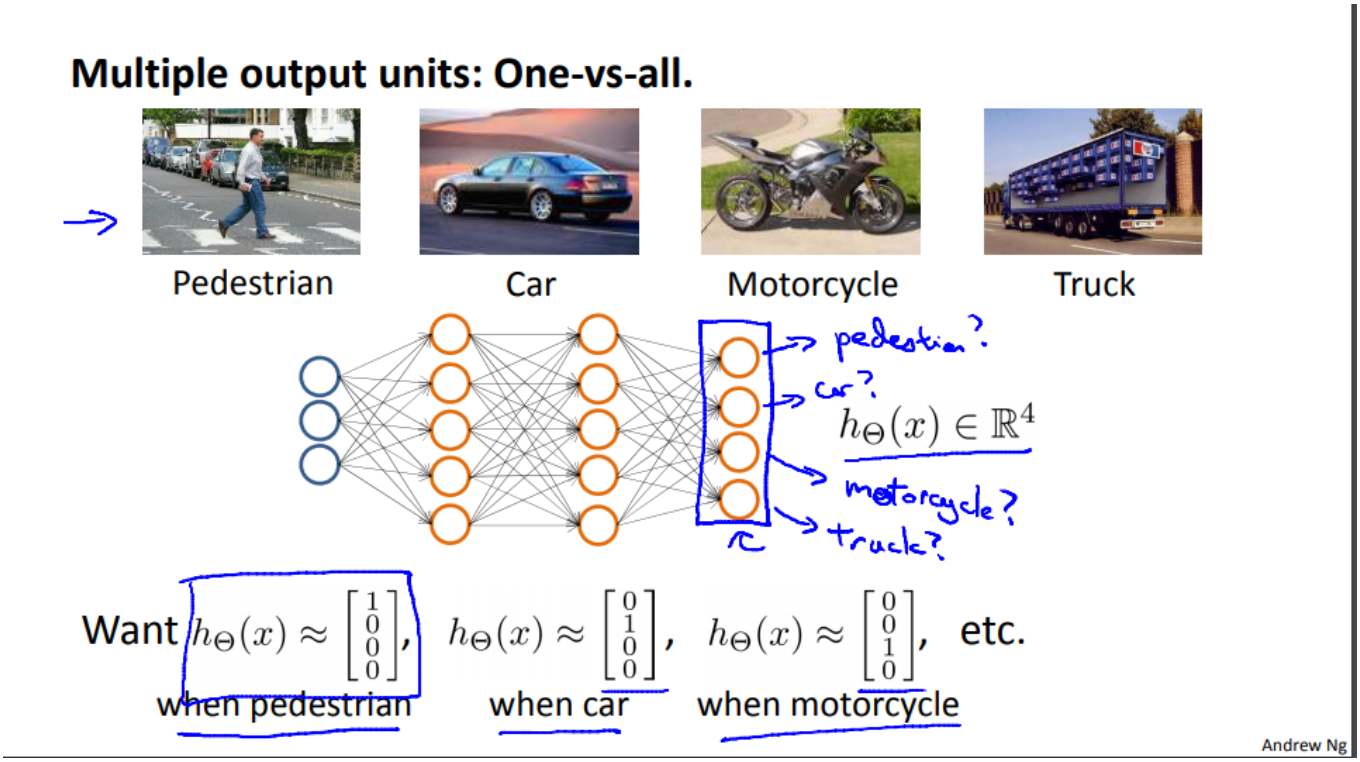
$$a^{(j)} = g(z^{(j)})$$

$$z^{(j+1)} = \Theta^{(j)}a^{(j)}$$

$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)})$$



Multi-class classification



Cost Function

Cost Function

Let's first define a few variables that we will need to use:

- L = total number of layers in the network
- s_l = number of units (not counting bias unit) in layer l
- K = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ as being a hypothesis that results in the k^{th} output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression. Recall that the cost function for regularized logistic regression was:

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

For neural networks, it is going to be slightly more complicated:

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

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
- the triple sum simply adds up the squares of all the individual Θs in the entire network.
- the i in the triple sum does **not** refer to training example i

Advice for Applying Machine Learning

Errors in your predictions can be troubleshooted by:

Getting more training examples: Fixes high variance

Trying smaller sets of features: Fixes high variance

Adding features: Fixes high bias

Adding polynomial features: Fixes high bias

Decreasing λ: Fixes high bias

Increasing λ: Fixes high variance.

Evaluating a Hypothesis

Model Selection and Train/Validation/Test Sets

The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than any other data set.

Without the Validation Set (note: this is a bad method - do not use it)

the training set consists of 70 % of your data and the test set is the remaining 30 %.

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the polynomial degree d with the least error using the test set.
3. Estimate the generalization error also using the test set with $J_{test}\left(\Theta^{(d)}\right)$, (d = theta from polynomial with lower error);

In this case, we have trained one variable, d, or the degree of the polynomial, using the test set. This will cause our error value to be greater for any other set of data.

Use of the CV set

To solve this, we can introduce a third set, the Cross Validation Set, to serve as an intermediate set that we can train d with. Then our test set will give us an accurate, non-optimistic error.

One example way to break down our dataset into the three sets is:

- Training set: 60%
- Cross validation set: 20%
- Test set: 20% We can now calculate three separate error values for the three different sets.

With the Validation Set

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the polynomial degree d with the least error using the cross validation set.
3. Estimate the generalization error using the test set with $J_{test}\left(\Theta^{(d)}\right)$, (d = theta from polynomial with lower error);

This way, the degree of the polynomial d has not been trained using the test set.

note: be aware that using the CV set to select 'd' means that we cannot also use it for the validation curve process of setting the lambda value.

Diagnosing Bias vs. Variance

- We need to distinguish whether **bias** or **variance** is the problem contributing to bad predictions.
- High bias is underfitting and high variance is overfitting. We need to find a golden mean between these two.

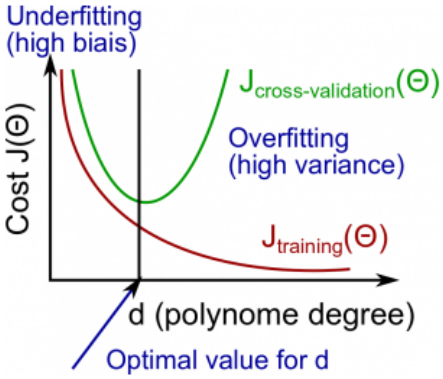
The training error will tend to **decrease** as we increase the degree d of the polynomial.

At the same time, the cross validation error will tend to **decrease** as we increase d up to a point, and then it will **increase** as d is increased, forming a convex curve.

High bias (underfitting): both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ will be high. Also, $J_{CV}(\Theta) \approx J_{train}(\Theta)$.

High variance (overfitting): $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be much greater than $J_{train}(\Theta)$.

The is represented in the figure below:



Regularization and Bias/Variance

Instead of looking at the degree d contributing to bias/variance, now we will look at the regularization parameter λ.

- Large λ: High bias (underfitting)
- Intermediate λ: just right
- Small λ: High variance (overfitting)

A large lambda heavily penalizes all the Θ parameters, which greatly simplifies the line of our resulting function, so causes underfitting.

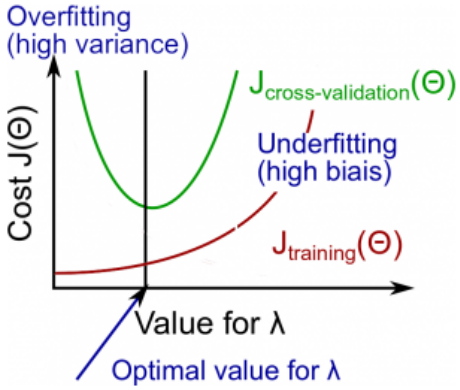
The relationship of λ to the training set and the variance set is as follows:

Low λ: $J_{train}(\Theta)$ is low and $J_{CV}(\Theta)$ is high (high variance/overfitting).

Intermediate λ: $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ are somewhat low and $J_{train}(\Theta) \approx J_{CV}(\Theta)$.

Large λ: both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ will be high (underfitting /high bias)

The figure below illustrates the relationship between lambda and the hypothesis:



In order to choose the model and the regularization λ, we need:

1. Create a list of lambdas (i.e. λ∈{0,0.01,0.02,0.04,0.08,0.16,0.32,0.64,1.28,2.56,5.12,10.24});
2. Create a set of models with different degrees or any other variants.
3. Iterate through the \lambdaλs and for each \lambdaλ go through all the models to learn some \ThetaΘ.
4. Compute the cross validation error using the learned Θ (computed with λ) on the $J_{CV}(\Theta)$ without regularization or λ = 0.
5. Select the best combo that produces the lowest error on the cross validation set.
6. Using the best combo Θ and λ, apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem.

Learning Curves

With high bias

Low training set size: causes $J_{train}(\Theta)$ to be low and $J_{CV}(\Theta)$ to be high.

Large training set size: causes both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ to be high with $J_{train}(\Theta) \approx J_{CV}(\Theta)$.

If a learning algorithm is suffering from **high bias**, getting more training data **will not (by itself) help much**.

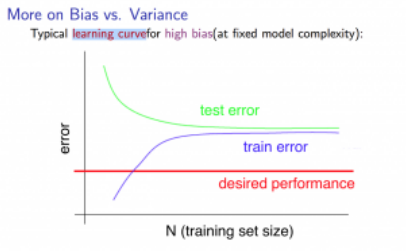
For high variance, we have the following relationships in terms of the training set size:

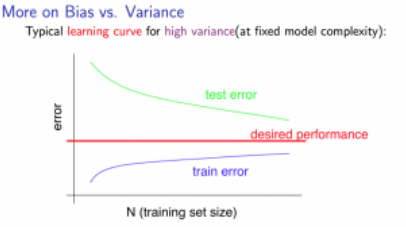
With high variance

Low training set size: $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be high.

Large training set size: $J_{train}(\Theta)$ increases with training set size and $J_{CV}(\Theta)$ continues to decrease without leveling off. Also, $J_{train}(\Theta) < J_{CV}(\Theta)$ but the difference between them remains significant.

If a learning algorithm is suffering from **high variance**, getting more training data is **likely to help.**





Model Selection:

Bias: approximation error (Difference between expected value and optimal value)

- High Bias = UnderFitting (BU)
- $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ both will be high and $J_{train}(\Theta) \approx J_{CV}(\Theta)$

Variance: estimation error due to finite data

- High Variance = OverFitting (VO)
- $J_{train}(\Theta)$ is low and $J_{CV}(\Theta) \gg J_{train}(\Theta)$

Intuition for the bias-variance trade-off:

- Complex model => sensitive to data => much affected by changes in X => high variance, low bias.
- Simple model => more rigid => does not change as much with changes in X => low variance, high bias.

One of the most important goals in learning: finding a model that is just right in the bias-variance trade-off.

Regularization Effects:

- Small values of λ allow model to become finely tuned to noise leading to large variance => overfitting.
- Large values of λ pull weight parameters to zero leading to large bias => underfitting.

Model Complexity Effects:

- Lower-order polynomials (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.
- Higher-order polynomials (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.
- In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

A typical rule of thumb when running diagnostics is:

- More training examples fixes high variance but not high bias.
- Fewer features fixes high variance but not high bias.
- Additional features fixes high bias but not high variance.
- The addition of polynomial and interaction features fixes high bias but not high variance.
- When using gradient descent, decreasing lambda can fix high bias and increasing lambda can fix high variance (lambda is the regularization parameter).
- When using neural networks, small neural networks are more prone to under-fitting and big neural networks are prone to over-fitting. Cross-validation of network size is a way to choose alternatives.

ML:Machine Learning System Design

Prioritizing What to Work On

Different ways we can approach a machine learning problem:

- Collect lots of data (for example "honeypot" project but doesn't always work)
- Develop sophisticated features (for example: using email header data in spam emails)
- Develop algorithms to process your input in different ways (recognizing misspellings in spam).

It is difficult to tell which of the options will be helpful.

Error Analysis

The recommended approach to solving machine learning problems is:

- Start with a simple algorithm, implement it quickly, and test it early.
- Plot learning curves to decide if more data, more features, etc. will help
- Error analysis: manually examine the errors on examples in the cross validation set and try to spot a trend.

It's important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance.

You may need to process your input before it is useful. For example, if your input is a set of words, you may want to treat the same word with different forms (fail/failing/failed) as one word, so must use "stemming software" to recognize them all as one.

For example, assume that we have 500 emails and our algorithm misclassifies a 100 of them. We could manually analyze the 100 emails and categorize them based on what type of emails they are. We could then try to come up with new cues and features that would help us classify these 100 emails correctly. Hence, if most of our misclassified emails are those which try to steal passwords, then we could find some features that are particular to those emails and add them to our model. We could also see how classifying each word according to its root changes our error rate:

It is very important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance. For example if we use stemming, which is the process of treating the same word with different forms (fail/failing/failed) as one word (fail), and get a 3% error rate instead of 5%, then we should definitely add it to our model. However, if we try to distinguish between upper case and lower case letters and end up getting a 3.2% error rate instead of 3%, then we should avoid using this new feature. Hence, we should try new things, get a numerical value for our error rate, and based on our result decide whether we want to keep the new feature or not.

Error Metrics for Skewed Classes

It is sometimes difficult to tell whether a reduction in error is actually an improvement of the algorithm.

- For example: In predicting a cancer diagnoses where 0.5% of the examples have cancer, we find our learning algorithm has a 1% error. However, if we were to simply classify every single example as a 0, then our error would reduce to 0.5% even though we did not improve the algorithm.

This usually happens with **skewed classes**; that is, when our class is very rare in the entire data set.

Or to say it another way, when we have lot more examples from one class than from the other class.

For this we can use **Precision/Recall**.

- Predicted: 1, Actual: 1 --- True positive
- Predicted: 0, Actual: 0 --- True negative
- Predicted: 0, Actual, 1 --- False negative
- Predicted: 1, Actual: 0 --- False positive

Precision: of all patients we predicted where y=1, what fraction actually has cancer?

$$\frac{\text{True Positives}}{\text{Total number of predicted positives}} = \frac{\text{True Positives}}{\text{True Positives} + \text{False positives}}$$

Recall: Of all the patients that actually have cancer, what fraction did we correctly detect as having cancer?

$$\frac{\text{True Positives}}{\text{Total number of actual positives}} = \frac{\text{True Positives}}{\text{True Positives} + \text{False negatives}}$$

These two metrics give us a better sense of how our classifier is doing. We want both precision and recall to be high.

In the example at the beginning of the section, if we classify all patients as 0, then our **recall** will be $\frac{0}{0+f} = 0$, so despite having a lower error percentage, we can quickly see it has worse recall.

Accuracy = $\frac{truepositive+truenegative}{totalpopulation}$

Note 1: if an algorithm predicts only negatives like it does in one of exercises, the precision is not defined, it is impossible to divide by 0. F1 score will not be defined too.

Trading Off Precision and Recall

We might want a **confident** prediction of two classes using logistic regression. One way is to increase our threshold:

- Predict 1 if: $h_\theta(x) \geq 0.7$
- Predict 0 if: $h_\theta(x) < 0.7$

This way, we only predict cancer if the patient has a 70% chance.

Doing this, we will have **higher precision** but **lower recall** (refer to the definitions in the previous section).

In the opposite example, we can lower our threshold:

- Predict 1 if: $h_\theta(x) \geq 0.3$
- Predict 0 if: $h_\theta(x) < 0.3$

That way, we get a very **safe** prediction. This will cause **higher recall** but **lower precision**.

The greater the threshold, the greater the precision and the lower the recall.

The lower the threshold, the greater the recall and the lower the precision.

In order to turn these two metrics into one single number, we can take the **F value**.

One way is to take the **average**:

$$\frac{P+R}{2}$$

This does not work well. If we predict all y=0 then that will bring the average up despite having 0 recall. If we predict all examples as y=1, then the very high recall will bring up the average despite having 0 precision.

A better way is to compute the **F Score** (or F1 score):

$$\text{F Score} = 2\frac{PR}{P+R}$$

In order for the F Score to be large, both precision and recall must be large.

We want to train precision and recall on the **cross validation set** so as not to bias our test set.

Trading off precision and recall

- Logistic regression: $0 \le h_\theta(x) \le 1$

  Predict 1 if $h_\theta(x) \ge 0.5$ 0.7 0.9 0.3

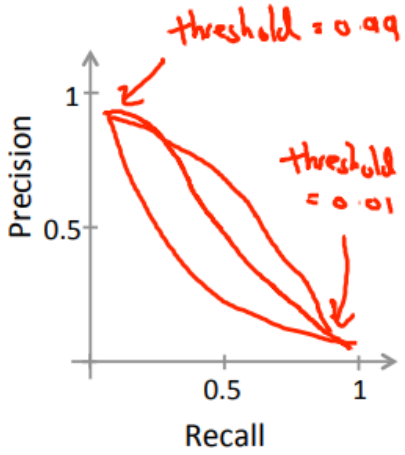
  Predict 0 if $h_\theta(x) < 0.5$ 0.7 0.9 0.3

- Suppose we want to predict $y = 1$ (cancer) only if very confident.
  - Higher precision, lower recall

- Suppose we want to avoid missing too many cases of cancer (avoid false negatives).
  - Higher recall, lower precision.

More generally: Predict 1 if $h_\theta(x) \ge$ threshold

- precision = $\frac{\text{true positives}}{\text{no. of predicted positive}}$
- recall = $\frac{\text{true positives}}{\text{no. of actual positive}}$



Andrew Ng

F_1 Score (F score)

How to compare precision/recall numbers?

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

Predict y=1 all the time

Average: $\frac{P+R}{2}$

F_1 Score: $2\frac{PR}{P+R}$

P=0 or R=0 ⇒ F-score = 0

P=1 and R=1 ⇒ F-score = 1

Data for Machine Learning

How much data should we train on?

In certain cases, an "inferior algorithm," if given enough data, can outperform a superior algorithm with less data.

We must choose our features to have enough information. A useful test is: Given input x, would a human expert be able to confidently predict y?

Rationale for large data: if we have a low bias algorithm (many features or hidden units making a very complex function), then the larger the training set we use, the less we will have overfitting (and the more accurate the algorithm will be on the test set).

Support Vector Machine (SVM)

Optimization Objective

The **Support Vector Machine** (SVM) is yet another type of *supervised* machine learning algorithm. It is sometimes cleaner and more powerful.

Recall that in logistic regression, we use the following rules:

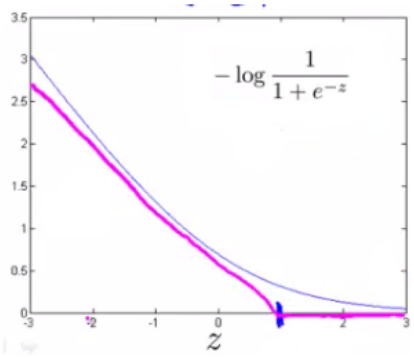
if y=1, then $h_\theta(x) \approx 1$ and $\Theta^T x \gg 0$

if y=0, then $h_\theta(x) \approx 0$ and $\Theta^T x \ll 0$

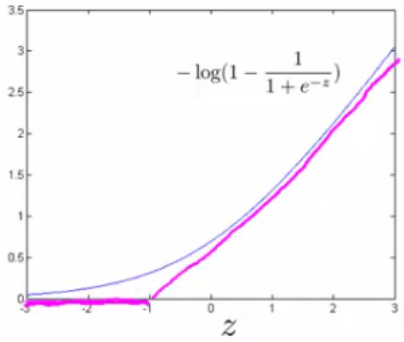
Recall the cost function for (unregularized) logistic regression:

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

To make a support vector machine, we will modify the first term of the cost function $-\log(h_\theta(x)) = -\log\left(\frac{1}{1+e^{-\theta^T x}}\right)$ so that when $\theta^T x$ (from now on, we shall refer to this as z) is **greater than** 1, it outputs 0. Furthermore, for values of z less than 1, we shall use a straight decreasing line instead of the sigmoid curve.(In the literature, this is called a hinge loss (https://en.wikipedia.org/wiki/Hinge_loss) function.)



Similarly, we modify the second term of the cost function $-\log(1-h_{\theta(x)}) = -\log\left(1-\frac{1}{1+e^{-\theta^T x}}\right)$ so that when z is **less than** -1, it outputs 0. We also modify it so that for values of z greater than -1, we use a straight increasing line instead of the sigmoid curve.



We shall denote these as $\text{cost}_1(z)$ and $\text{cost}_0(z)$ (respectively, note that $\text{cost}_1(z)$ is the cost for classifying when y=1, and $\text{cost}_0(z)$ is the cost for classifying when y=0), and we may define them as follows (where k is an arbitrary constant defining the magnitude of the slope of the line):

$$z = \theta^T x$$

$$\text{cost}_0(z) = \max(0, k(1+z))$$

$$\text{cost}_1(z) = \max(0, k(1-z))$$

Recall the full cost function from (regularized) logistic regression:

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

Note that the negative sign has been distributed into the sum in the above equation.

We may transform this into the cost function for support vector machines by substituting $\text{cost}_0(z)$ and $\text{cost}_1(z)$:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}\,\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\,\text{cost}_0(\theta^T x^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n}\Theta_j^2$$

We can optimize this a bit by multiplying this by m (thus removing the m factor in the denominators). Note that this does not affect our optimization, since we're simply multiplying our cost function by a positive constant (for example, minimizing $(u-5)^2+1$ gives us 5; multiplying it by 10 to make it $10(u-5)^2+10$ still gives us 5 when minimized).

$$J(\theta) = \sum_{i=1}^{m} y^{(i)}\,\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\,\text{cost}_0(\theta^T x^{(i)}) + \frac{\lambda}{2}\sum_{j=1}^{n}\Theta_j^2$$

Furthermore, convention dictates that we regularize using a factor C, instead of λ, like so:

$$J(\theta) = C\sum_{i=1}^{m} y^{(i)}\,\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\,\text{cost}_0(\theta^T x^{(i)}) + \frac{1}{2}\sum_{j=1}^{n}\Theta_j^2$$

This is equivalent to multiplying the equation by $C = \frac{1}{\lambda}$, and thus results in the same values when optimized. Now, when we wish to regularize more (that is, reduce overfitting), we *decrease* C, and when we wish to regularize less (that is, reduce underfitting), we *increase* C.

Finally, note that the hypothesis of the Support Vector Machine is *not* interpreted as the probability of y being 1 or 0 (as it is for the hypothesis of logistic regression). Instead, it outputs either 1 or 0. (In technical terms, it is a discriminant function.)

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

Large Margin Intuition

A useful way to think about Support Vector Machines is to think of them as *Large Margin Classifiers*.

If y=1, we want $\Theta^T x \geq 1$ (not just ≥0)

If y=0, we want $\Theta^T x \leq -1$ (not just <0)

Now when we set our constant C to a very **large** value (e.g. 100,000), our optimizing function will constrain Θ such that the equation A (the summation of the cost of each example) equals 0. We impose the following constraints on Θ:

$\Theta^T x \geq 1$ if y=1 and $\Theta^T x \leq -1$ if y=0.

If C is very large, we must choose Θ parameters such that:

$\sum_{i=1}^{m} y^{(i)} \text{cost}_1(\Theta^T x) + (1 - y^{(i)}) \text{cost}_0(\Theta^T x) = 0$

This reduces our cost function to:

$$J(\theta) = C \cdot 0 + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$$
$$= \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$$

Recall the decision boundary from logistic regression (the line separating the positive and negative examples). In SVMs, the decision boundary has the special property that it is **as far away as possible** from both the positive and the negative examples.

The distance of the decision boundary to the nearest example is called the **margin**. Since SVMs maximize this margin, it is often called a *Large Margin Classifier*.

The SVM will separate the negative and positive examples by a **large margin**.

This large margin is only achieved when **C is very large**.

Data is **linearly separable** when a **straight line** can separate the positive and negative examples.

If we have **outlier** examples that we don't want to affect the decision boundary, then we can **reduce** C.

Increasing and decreasing C is similar to respectively decreasing and increasing λ, and can simplify our decision boundary.

Kernels I

Kernels allow us to make complex, non-linear classifiers using Support Vector Machines.

Given x, compute new feature depending on proximity to landmarks $l^{(1)}$, $l^{(2)}$, $l^{(3)}$.

To do this, we find the "similarity" of x and some landmark $l^{(i)}$:

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

This "similarity" function is called a **Gaussian Kernel**. It is a specific example of a kernel.

The similarity function can also be written as follows:

$$f_i = similarity(x, l^{(i)}) = \exp(-\frac{\sum_{j=1}^{n}(x_j - l_j^{(i)})^2}{2\sigma^2})$$

There are a couple properties of the similarity function:

If $x \approx l^{(i)}$, then $f_i = \exp(-\frac{\approx 0^2}{2\sigma^2}) \approx 1$

If x is far from $l^{(i)}$, then $f_i = \exp(-\frac{(large\ number)^2}{2\sigma^2}) \approx 0$

In other words, if x and the landmark are close, then the similarity will be close to 1, and if x and the landmark are far away from each other, the similarity will be close to 0.

Each landmark gives us the features in our hypothesis:

$$\begin{aligned} l^{(1)} &\rightarrow f_1 \\ l^{(2)} &\rightarrow f_2 \\ l^{(3)} &\rightarrow f_3 \\ &\dots \\ h_\Theta(x) &= \Theta_1 f_1 + \Theta_2 f_2 + \Theta_3 f_3 + \dots \end{aligned}$$

σ^2 is a parameter of the Gaussian Kernel, and it can be modified to increase or decrease the **drop-off** of our feature f_i. Combined with looking at the values inside Θ, we can choose these landmarks to get the general shape of the decision boundary.

Kernels II

One way to get the landmarks is to put them in the **exact same** locations as all the training examples. This gives us m landmarks, with one landmark per training example.

Given example x:

$f_1 = similarity(x, l^{(1)})$, $f_2 = similarity(x, l^{(2)})$, $f_3 = similarity(x, l^{(3)})$, and so on.

This gives us a "feature vector," $f_{(i)}$ of all our features for example $x_{(i)}$. We may also set $f_0 = 1$ to correspond with Θ_0. Thus given training example $x_{(i)}$:

$$x^{(i)} \rightarrow \left[f_1^{(i)} = similarity(x^{(i)}, l^{(1)}) f_2^{(i)} = similarity(x^{(i)}, l^{(2)}) \vdots f_m^{(i)} = similarity(x^{(i)}, l^{(m)})\right]$$

Now to get the parameters Θ we can use the SVM minimization algorithm but with $f^{(i)}$ substituted in for $x^{(i)}$:

$$\min_\Theta C \sum_{i=1}^{m} y^{(i)} \text{cost}_1(\Theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$$

Using kernels to generate f(i) is not exclusive to SVMs and may also be applied to logistic regression. However, because of computational optimizations on SVMs, kernels combined with SVMs is much faster than with other algorithms, so kernels are almost always found combined only with SVMs.

Choosing SVM Parameters

Choosing C (recall that $C = \frac{1}{\lambda}$

- If C is large, then we get higher variance/lower bias
- If C is small, then we get lower variance/higher bias

The other parameter we must choose is σ^2 from the Gaussian Kernel function:

With a large σ^2, the features fi vary more smoothly, causing higher bias and lower variance.

With a small σ^2, the features fi vary less smoothly, causing lower bias and higher variance.

In practical application, the choices you do need to make are:

- Choice of parameter C
- Choice of kernel (similarity function)
- No kernel ("linear" kernel) -- gives standard linear classifier
- Choose when n is large and when m is small
- Gaussian Kernel (above) -- need to choose σ^2
- Choose when n is small and m is large

The library may ask you to provide the kernel function.

Note: do perform feature scaling before using the Gaussian Kernel.

Note: not all similarity functions are valid kernels. They must satisfy "Mercer's Theorem" which guarantees that the SVM package's optimizations run correctly and do not diverge.

You want to train C and the parameters for the kernel function using the training and cross-validation datasets.

Multi-class Classification

Many SVM libraries have multi-class classification built-in.

You can use the *one-vs-all* method just like we did for logistic regression, where $y \in 1, 2, 3, \ldots, K$ with $\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta(K)$. We pick class i with the largest $(\Theta^{(i)})^T x$.

Logistic Regression vs. SVMs

If n is large (relative to m), then use logistic regression, or SVM without a kernel (the "linear kernel")

Unsupervised Learning

Unsupervised learning is contrasted from supervised learning because it uses an unlabeled training set rather than a labeled one.

In other words, we don't have the vector y of expected results, we only have a dataset of features where we can find structure.

Clustering is good for:

- Market segmentation
- Social network analysis
- Organizing computer clusters
- Astronomical data analysis

Clustering - K-Means Algorithm

The K-Means Algorithm is the most popular and widely used algorithm for automatically grouping data into coherent subsets.

1. Randomly initialize two points in the dataset called the cluster centroids.
2. Cluster assignment: assign all examples into one of two groups based on which cluster centroid the example is closest to.
3. Move centroid: compute the averages for all the points inside each of the two cluster centroid groups, then move the cluster centroid points to those averages.
4. Re-run (2) and (3) until we have found our clusters.

Our main variables are:

K (number of clusters) Training set $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$

Where $x^{(i)} \in \mathbb{R}^n$

Note that we will not use the x0=1 convention.

The algorithm:

```
Randomly initialize K cluster centroids mu(1), mu(2), ..., mu(K)
Repeat:
   for i = 1 to m:
      c(i):= index (from 1 to K) of cluster centroid closest to x(i)
   for k = 1 to K:
      mu(k):= average (mean) of points assigned to cluster k
```

The **first for-loop** is the 'Cluster Assignment' step. We make a vector *c* where *c(i)* represents the centroid assigned to example *x(i)*.

We can write the operation of the Cluster Assignment step more mathematically as follows:

$c^{(i)} = argmin_k \ ||x^{(i)} - \mu_k||^2$

That is, each $c^{(i)}$ contains the index of the centroid that has minimal distance to $x^{(i)}$.

By convention, we square the right-hand-side, which makes the function we are trying to minimize more sharply increasing. It is mostly just a convention. But a convention that helps reduce the computation load because the Euclidean distance requires a square root but it is canceled.

Without the square:

$$||x^{(i)} - \mu_k|| = || \quad \sqrt{(x_1^i - \mu_{1(k)})^2 + (x_2^i - \mu_{2(k)})^2 + (x_3^i - \mu_{3(k)})^2 + \ldots} \quad ||$$

With the square:

$$||x^{(i)} - \mu_k||^2 = || \quad (x_1^i - \mu_{1(k)})^2 + (x_2^i - \mu_{2(k)})^2 + (x_3^i - \mu_{3(k)})^2 + \ldots \quad ||$$

...so the square convention serves two purposes, minimize more sharply and less computation.

The **second for-loop** is the 'Move Centroid' step where we move each centroid to the average of its group.

More formally, the equation for this loop is as follows:

$$\mu_k = \frac{1}{n}[x^{(k_1)} + x^{(k_2)} + \cdots + x^{(k_n)}] \in \mathbb{R}^n$$

Where each of $x^{(k_1)}, x^{(k_2)}, \ldots, x^{(k_n)}$ are the training examples assigned to group $m\mu_k$.

If you have a cluster centroid with **0 points** assigned to it, you can randomly **re-initialize** that centroid to a new point. You can also simply **eliminate** that cluster group.

After a number of iterations the algorithm will **converge**, where new iterations do not affect the clusters.

Optimization Objective

Recall some of the parameters we used in our algorithm:

- $c^{(i)}$ = index of cluster (1,2,...,K) to which example x(i) is currently assigned
- μ_k = cluster centroid k (μkϵℜn)
- $\mu_{c^{(i)}}$ = cluster centroid of cluster to which example x(i) has been assigned

Using these variables we can define our **cost function**:

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

Our **optimization objective** is to minimize all our parameters using the above cost function:

$$min_{c,\mu}\ J(c,\mu)$$

That is, we are finding all the values in sets c, representing all our clusters, and μ, representing all our centroids, that will minimize **the average of the distances** of every training example to its corresponding cluster centroid.

The above cost function is often called the **distortion** of the training examples.

In the **cluster assignment step**, our goal is to:

Minimize J(...) with $c^{(1)},\ldots,c^{(m)}$ (holding $\mu_1,\ldots,\mu_K$ fixed)

In the **move centroid** step, our goal is to:

Minimize J(...) with $\mu_1,\ldots,\mu_K$

With k-means, it is **not possible for the cost function to sometimes increase**. It should always descend.

Random Initialization There's one particular recommended method for randomly initializing your cluster centroids.

1. Have K<m. That is, make sure the number of your clusters is less than the number of your training examples.
2. Randomly pick K training examples. (Not mentioned in the lecture, but also be sure the selected examples are unique).
3. Set $\mu_1, \ldots, \mu_K$ equal to these K examples.

K-means can get stuck in local optima. To decrease the chance of this happening, you can run the algorithm on many different random initializations. In cases where K<10 it is strongly recommended to run a loop of random initializations.

Choosing the Number of Clusters

Choosing K can be quite arbitrary and ambiguous.

The elbow method: plot the cost J and the number of clusters K. The cost function should reduce as we increase the number of clusters, and then flatten out. Choose K at the point where the cost function starts to flatten out.

However, fairly often, the curve is very gradual, so there's no clear elbow.

Note: J will always decrease as K is increased. The one exception is if k-means gets stuck at a bad local optimum.

Another way to choose K is to observe how well k-means performs on a downstream purpose. In other words, you choose K that proves to be most useful for some goal you're trying to achieve from using these clusters.

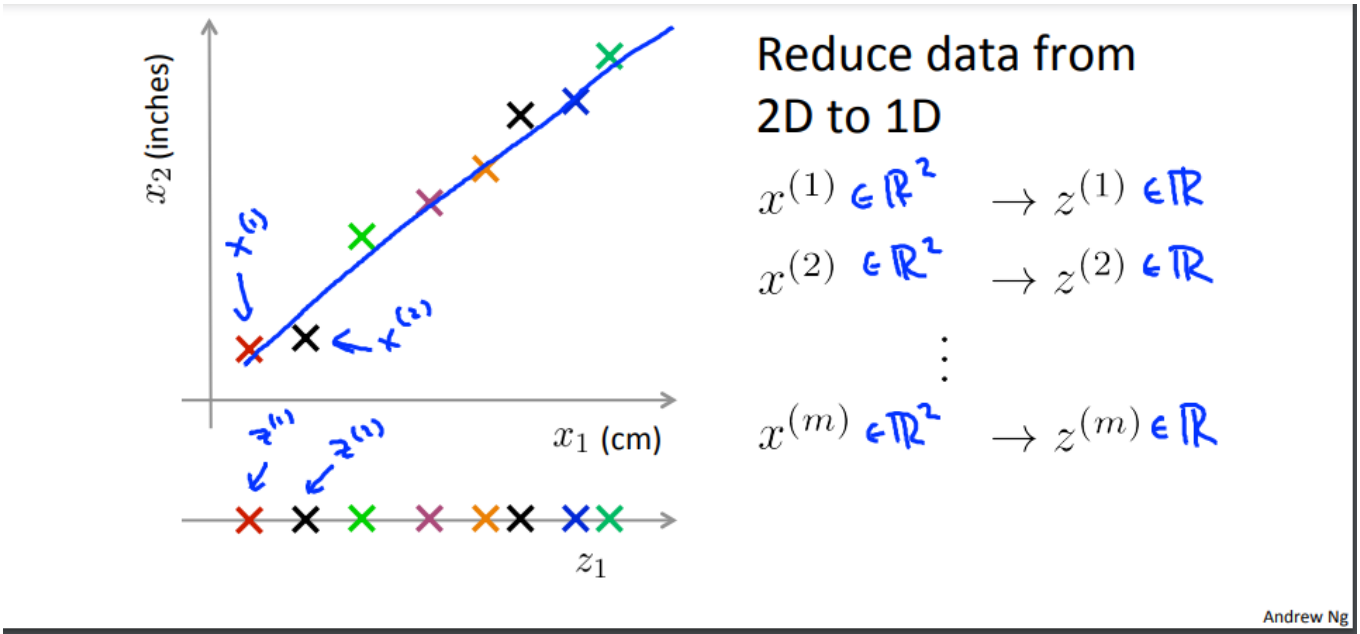
Dimensionality Reduction

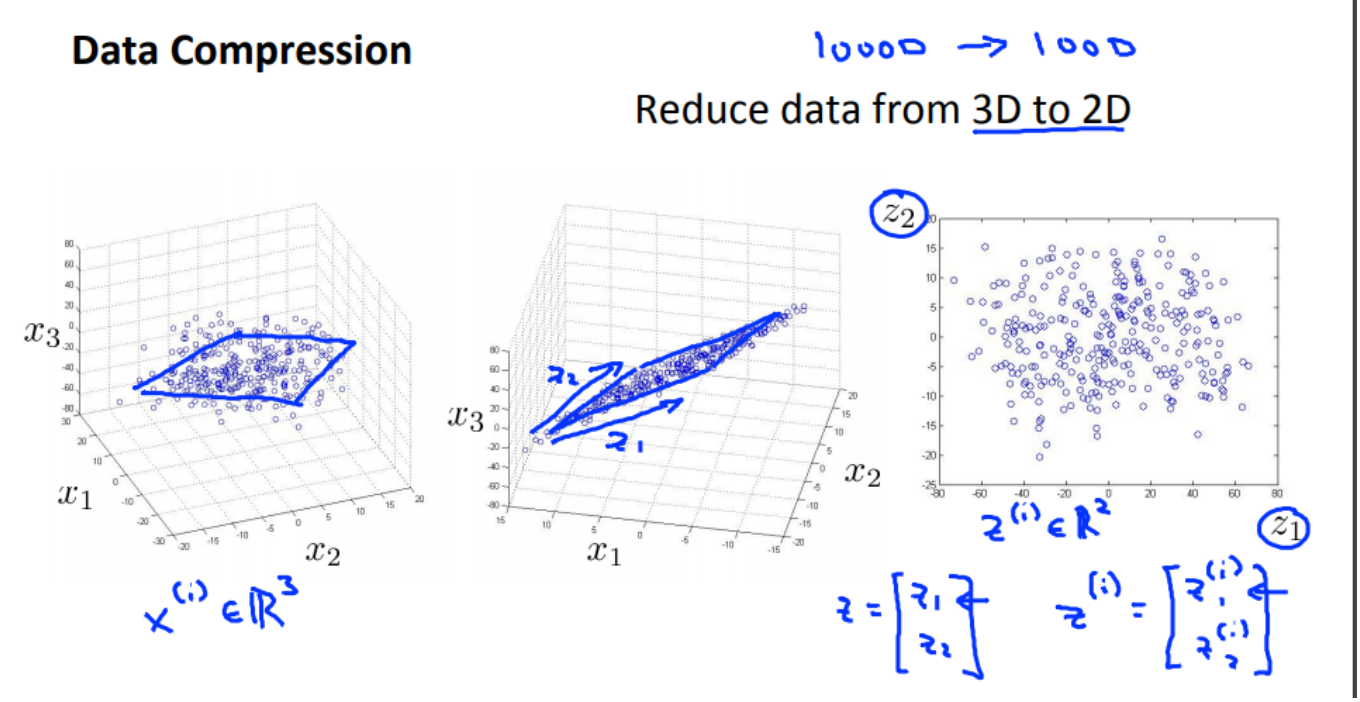
Motivation I: Data Compression

- We may want to reduce the dimension of our features if we have a lot of redundant data.
- To do this, we find two highly correlated features, plot them, and make a new line that seems to describe both features accurately. We place all the new features on this single line.

Doing dimensionality reduction will reduce the total data we have to store in computer memory and will speed up our learning algorithm.

Note: in dimensionality reduction, we are reducing our features rather than our number of examples. Our variable m will stay the same size; n, the number of features each example from $x^{(1)}$ to $x^{(m)}$ carries, will be reduced.





Motivation II: Visualization

It is not easy to visualize data that is more than three dimensions. We can reduce the dimensions of our data to 3 or less in order to plot it.

We need to find new features, z_1, z_2 (and perhaps z_3) that can effectively summarize all the other features.

Example: hundreds of features related to a country's economic system may all be combined into one feature that you call "Economic Activity."

Principal Component Analysis Problem Formulation(PCA)

The most popular dimensionality reduction algorithm

Problem formulation

Given two features, x_1x and x_2, we want to find a single line that effectively describes both features at once. We then map our old features onto this new line to get a new single feature.

The same can be done with three features, where we map them to a plane.

The goal of PCA is to reduce the average of all the distances of every feature to the projection line. This is the projection error.

Reduce from 2d to 1d: find a direction (a vector $u^{(1)} \in \mathbb{R}^n$) onto which to project the data so as to minimize the projection error.

The more general case is as follows:

Reduce from n-dimension to k-dimension: Find k vectors $u^{(1)}, u^{(2)}, \ldots, u^{(k)}$ onto which to project the data so as to minimize the projection error.

If we are converting from 3d to 2d, we will project our data onto two directions (a plane), so k will be 2.

PCA is not linear regression

In linear regression, we are minimizing the **squared error** from every point to our predictor line. These are vertical distances. In PCA, we are minimizing the **shortest distance**, or shortest orthogonal distances, to our data points. More generally, in linear regression we are taking all our examples in x and applying the parameters in Θ to predict y.

In PCA, we are taking a number of features $x_1, x_2, \ldots, x_n$, and finding a closest common dataset among them. We aren't trying to predict any result and we aren't applying any theta weights to the features.

Before we can apply PCA, there is a data pre-processing step we must perform:

Data preprocessing

- Given training set: x(1),x(2),…,x(m)
- Preprocess (feature scaling/mean normalization):

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

- Replace each $x_j^{(i)}$ with $x_j^{(i)} - \mu_j$
- If different features on different scales (e.g., x_1 = size of house, x_2x = number of bedrooms), scale features to have comparable range of values.

Above, we first subtract the mean of each feature from the original feature. Then we scale all the features

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

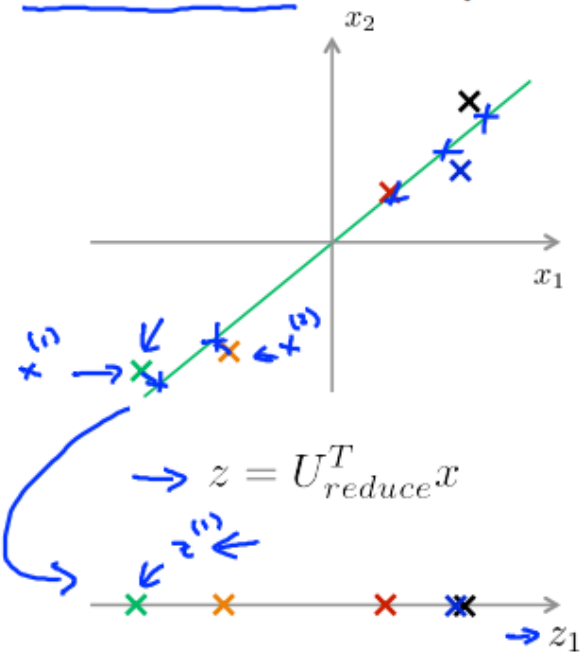
We can define specifically what it means to reduce from 2d to 1d data as follows:

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

The z values are all real numbers and are the projections of our features onto $u^{(1)}$

So, PCA has two tasks: figure out $u^{(1)}, \ldots, u^{(k)}$ and also to find $z_1, z_2, \ldots, z_m$

Reconstruction from compressed representation



$z \in \mathbb{R} \rightarrow x \in \mathbb{R}^2$

$x_{approx}^{(1)} = U_{reduce} \cdot z^{(1)}$

$x_{approx}^{(1)} \in \mathbb{R}^n$, U_{reduce}: $n \times k$, $z^{(1)}$: $k \times 1$, $n \times 1$

Andrew Ng

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

Typically, choose k to be smallest value so that

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

"99% of variance is retained"

Andrew Ng

Choosing k (number of principal components)

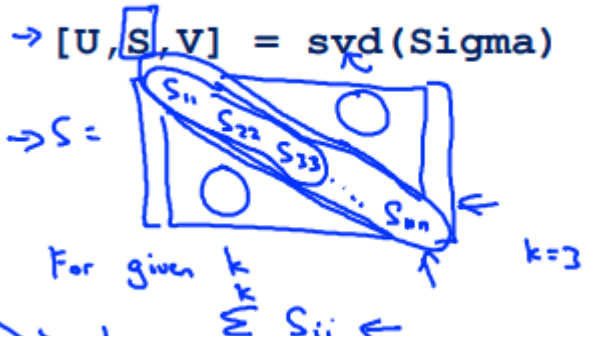
Algorithm:

Try PCA with $k = 1$

Compute $U_{reduce}, z^{(1)}, z^{(2)}, \ldots, z^{(m)}, x_{approx}^{(1)}, \ldots, x_{approx}^{(m)}$

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

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



Advice for Applying PCA

The most common use of PCA is to speed up supervised learning.

Given a training set with a large number of features (e.g. $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^{10000}$) we can use PCA to reduce the number of features in each example of the training set (e.g. $z^{(1)}, \ldots, z^{(m)} \in \mathbb{R}^{1000}$).

Note that we should define the PCA reduction from $x^{(i)}$ to $z^{(i)}$ only on the training set and not on the cross-validation or test sets. You can apply the mapping z(i) to your cross-validation and test sets after it is defined on the training set.

Applications

- Compressions Reduce space of data

Speed up algorithm

- Visualization of data Choose k = 2 or k = 3

Bad use of PCA: trying to prevent overfitting. We might think that reducing the features with PCA would be an effective way to address overfitting. It might work, but is not recommended because it does not consider the values of our results y. Using just regularization will be at least as effective.

Don't assume you need to do PCA. Try your full machine learning algorithm without PCA first. Then use PCA if you find that you need it.

Anomaly Detection

Just like in other learning problems, we are given a dataset $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$

We are then given a new example, x_{test}, and we want to know whether this new example is abnormal/anomalous.

We define a "model" p(x) that tells us the probability the example is not anomalous. We also use a threshold ϵ (epsilon) as a dividing line so we can say which examples are anomalous and which are not.

A very common application of anomaly detection is detecting fraud:

- $x^{(i)}$ = features of user i's activities
- Model p(x) from the data.
- Identify unusual users by checking which have p(x)<ϵ.

If our anomaly detector is flagging too many anomalous examples, then we need to decrease our threshold ϵ

Gaussian(Normal) Distribution

The Gaussian Distribution is a familiar bell-shaped curve that can be described by a function $\mathcal{N}(\mu, \sigma^2)$

Let x∈$\mathbb{R}$. If the probability distribution of x is Gaussian with mean μ, variance σ^2, then:

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

The little ∼ or 'tilde' can be read as "distributed as."

The Gaussian Distribution is parameterized by a mean and a variance.

Mu, or μ, describes the center of the curve, called the mean. The width of the curve is described by sigma, or σ, called the standard deviation.

The full function is as follows:

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

We can estimate the parameter μ from a given dataset by simply taking the average of all the examples:

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

We can estimate the other parameter, σ^2, with our familiar squared error formula:

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

Algorithm

Given a training set of examples, $\{x^{(1)}, \ldots, x^{(m)}$$whereeachexampleisavector,x \in \mathbb{R}^n$

$$p(x) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2) \cdots p(x_n; \mu_n, \sigma_n^2)$$

In statistics, this is called an "independence assumption" on the values of the features inside training example x.

More compactly, the above expression can be written as follows:

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

The algorithm

Choose features x_i that you think might be indicative of anomalous examples.

Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$

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

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

Given a new example x, compute p(x):

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

Anomaly if p(x)<ϵ

Developing and Evaluating an Anomaly Detection System

The importance of real-number evaluation

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: $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ (assume normal examples/not anomalous)
- Cross validation set: $(x_{cv}^{(1)}, y_{cv}^{(1)}), \ldots, (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$
- Test set: $(x_{test}^{(1)}, y_{test}^{(1)}), \ldots, (x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

$y=1$

Andrew Ng

Aircraft engines motivating example

- 10000 good (normal) engines
- 20 flawed engines (anomalous) 2-50 $y=1$

$\mu_1, \sigma_1^2, \ldots, \mu_n, \sigma_n^2$.

- Training set: 6000 good engines ($y=0$) $p(x) = p(x_1; \mu_1, \sigma_1^2) \cdots p(x_n; \mu_n, \sigma_n^2)$
- CV: 2000 good engines ($y = 0$), 10 anomalous ($y = 1$)
- Test: 2000 good engines ($y = 0$), 10 anomalous ($y = 1$)

Algorithm evaluation:

Fit model p(x) on training set $\lbrace x^{(1)},\dots,x^{(m)} $

On a cross validation/test example x, predict:

If p(x) < ϵ (anomaly), then y=1

If p(x) ≥ ϵ (normal), then y=0

Possible evaluation metrics (see "Machine Learning System Design" section):

- True positive, false positive, false negative, true negative.
- Precision/recall
- F_1 score Note that we use the cross-validation set to choose parameter ϵ

Anomaly Detection vs. Supervised Learning

Use anomaly detection when...

- We have a very small number of positive examples (y=1 ... 0-20 examples is common) and a large number of negative (y=0) examples.
- We have many different "types" of anomalies and it is 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.

Use supervised learning when...

- We have a large number of both positive and negative examples. In other words, the training set is more evenly divided into classes.
- We have enough positive examples for the algorithm to get a sense of what new positives examples look like. The future positive examples are likely similar to the ones in the training set.

Anomaly detection

• Fraud detection

• Manufacturing(e.g. aircraft engines)

• Monitoring machines in a data center

Supervised learning

• Email spam classification

• Weather prediction(sunny/ rainy/etc)

• Cancer classification

Choosing What Features to Use

The features will greatly affect how well your anomaly detection algorithm works.

We can check that our features are gaussian by plotting a histogram of our data and checking for the bell-shaped curve.

Some transforms we can try on an example feature x that does not have the bell-shaped curve are:

- log(x)
- log(x+1)
- log(x+c) for some constant
- $\sqrt{x}$
- $x^{1/3}$

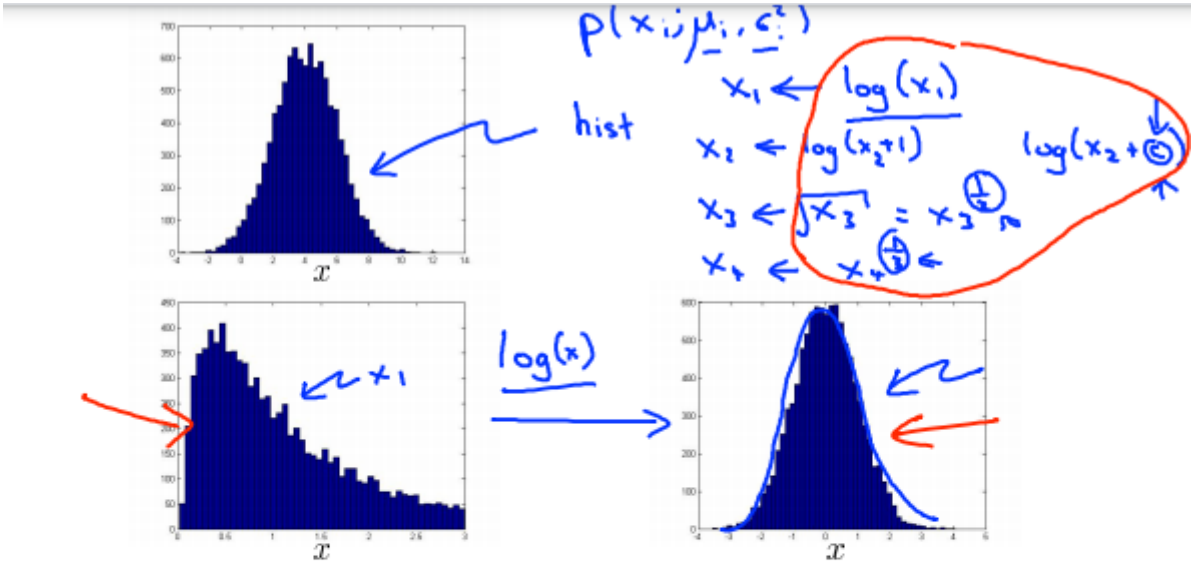
We can play with each of these to try and achieve the gaussian shape in our data.

There is an error analysis procedure for anomaly detection that is very similar to the one in supervised learning.

Our goal is for p(x) to be large for normal examples and small for anomalous examples.

One common problem is when p(x) is similar for both types of examples. In this case, you need to examine the anomalous examples that are giving high probability in detail and try to figure out new features that will better distinguish the data.

In general, choose features that might take on unusually large or small values in the event of an anomaly.

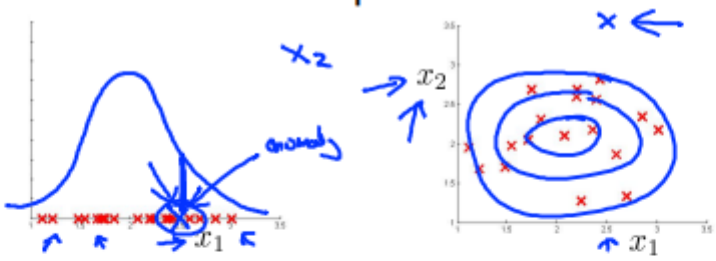


Error analysis for anomaly detection

Want $p(x)$ large for normal examples x.
$p(x)$ small for anomalous examples x.

Most common problem:

$p(x)$ is comparable (say, both large) for normal and anomalous examples



Monitoring computers in a data center

Choose features that might take on unusually large or small values in the event of an anomaly.

- x_1 = memory use of computer
- x_2 = number of disk accesses/sec
- x_3 = CPU load
- x_4 = network traffic

$x_5 = \frac{\text{CPU load}}{\text{network traffic}}$ $\quad x_6 = \frac{(\text{CPU load})^2}{\text{network traffic}}$

Multivariate Gaussian Distribution

The multivariate gaussian distribution is an extension of anomaly detection and may (or may not) catch more anomalies.

Instead of modeling $p(x_1), p(x_2), \ldots$ separately, we will model p(x) all in one go. Our parameters will be: $\mu \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$

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

The important effect is that we can model oblong gaussian contours, allowing us to better fit data that might not fit into the normal circular contours.

Varying Σ changes the shape, width, and orientation of the contours. Changing μ will move the center of the distribution

Anomaly Detection using the Multivariate Gaussian Distribution

When doing anomaly detection with multivariate gaussian distribution, we compute μ and Σ normally. We then compute p(x) using the new formula in the previous section and flag an anomaly if p(x) < ϵ.

The original model for p(x) corresponds to a multivariate Gaussian where the contours of $p(x; \mu, \Sigma)$ are axis-aligned.

The multivariate Gaussian model can automatically capture correlations between different features of x.

However, the original model maintains some advantages: it is computationally cheaper (no matrix to invert, which is costly for large number of features) and it performs well even with small training set size (in multivariate Gaussian model, it should be greater than the number of features for Σ to be invertible).

Recommender Systems

Problem Formulation

Recommendation is currently a very popular application of machine learning.

Say we are trying to recommend movies to customers. We can use the following definitions

- n_u = number of users
- n_m = number of 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)

Content Based Recommendations

We can introduce two features, x_1 and x_2 which represents how much romance or how much action a movie may have (on a scale of 0−1).

One approach is that we could do linear regression for every single user. For each user j, learn a parameter $\theta^{(j)} \in \mathbb{R}^3$. Predict user j as rating movie i with $(\theta^{(j)})^T x^{(i)}$ stars.

- $\theta^{(j)}$ = parameter vector for user j
- $x^{(i)}$ = feature vector for movie i

For user j, movie i, predicted rating: $(\theta^{(j)})^T(x^{(i)})$

- $m^{(j)}$ = number of movies rated by user j

To learn $\theta^{(j)}$, we do the following

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

This is our familiar linear regression. The base of the first summation is choosing all i such that $r(i,j) = 1$.

To get the parameters for all our users, we do the following:

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

We can apply our linear regression gradient descent update using the above cost function.

The only real difference is that we **eliminate the constant** $\frac{1}{m}$.

Collaborative Filtering

It can be very difficult to find features such as "amount of romance" or "amount of action" in a movie. To figure this out, we can use *feature finders*.

We can let the users tell us how much they like the different genres, providing their parameter vector immediately for us.

To infer the features from given parameters, we use the squared error function with regularization over all the users:

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

You can also **randomly guess** the values for theta to guess the features repeatedly. You will actually converge to a good set of features.

Collaborative Filtering Algorithm

To speed things up, we can simultaneously minimize our features and our parameters:

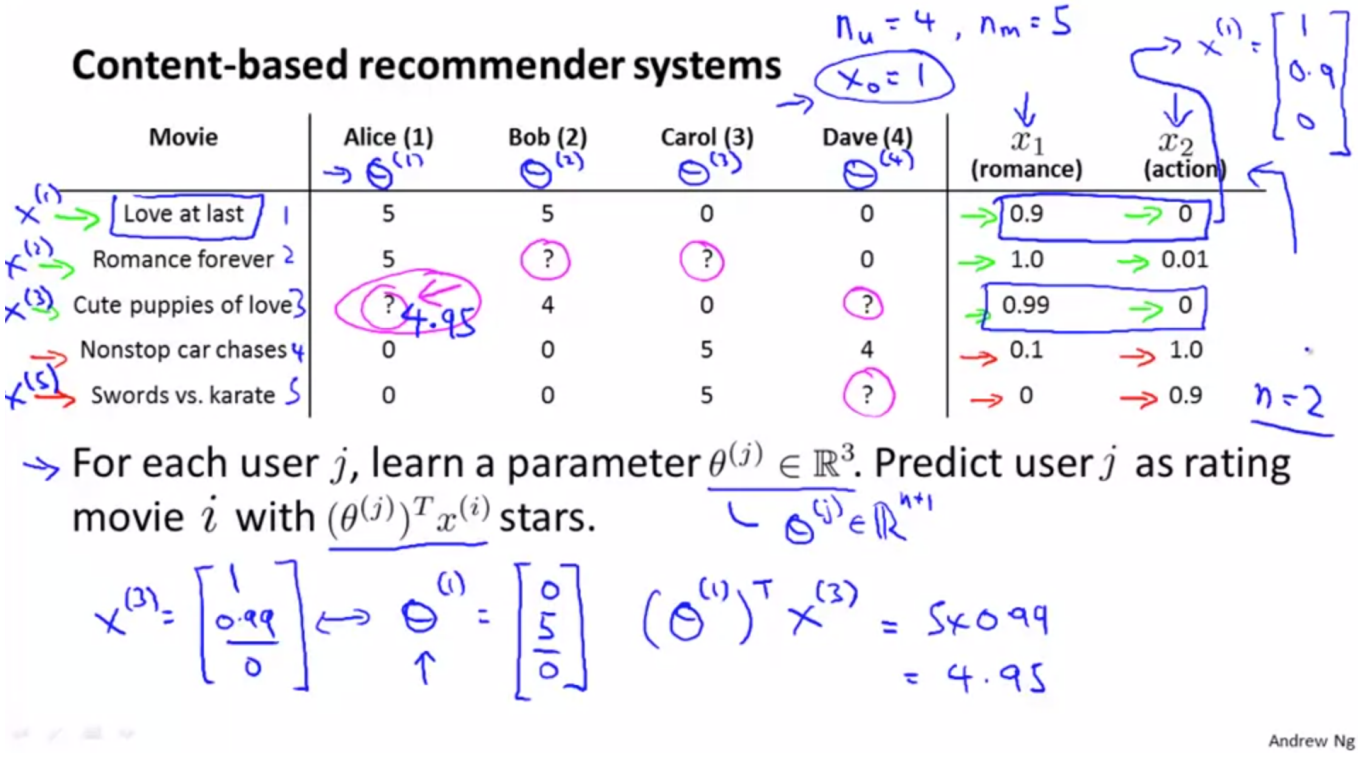
$$J(x,\theta) = \frac{1}{2}\sum_{(i,j):r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

It looks very complicated, but we've only combined the cost function for theta and the cost function for x.

Because the algorithm can learn them itself, the bias units where x0=1 have been removed, therefore $x \in \mathbb{R}^n$ and $\theta \in \mathbb{R}^n$.

These are the steps in the algorithm:

1. Initialize $x^{(i)},\dots,x^{(n_m)},\theta^{(1)},\dots,\theta^{(n_u)}$ to small random values. This serves to break symmetry and ensures that the algorithm learns features $x^{(i)},\dots,x^{(n_m)}$ that are different from each other.
2. Minimize $J(x^{(i)},\dots,x^{(n_m)},\theta^{(1)},\dots,\theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm).E.g. for every $j=1,\dots,n_u, i=1,\dots n_m$: $x_k^{(i)} := x_k^{(i)} - \alpha\left(\sum_{j:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})\theta_k^{(j)} + \lambda x_k^{(i)}\right)$ $\theta_k^{(j)} := \theta_k^{(j)} - \alpha\left(\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} + \lambda\theta_k^{(j)}\right)$
3. For a user with parameters θ and a movie with (learned) features x, predict a star rating of $\theta^T x$.



Andrew Ng

Implementation Detail: Mean Normalization

If the ranking system for movies is used from the previous lectures, then new users (who have watched no movies), will be assigned new movies incorrectly. Specifically, they will be assigned θ with all components equal to zero due to the minimization of the regularization term. That is, we assume that the new user will rank all movies 0, which does not seem intuitively correct.

We rectify this problem by normalizing the data relative to the mean. First, we use a matrix Y to store the data from previous ratings, where the ith row of Y is the ratings for the ith movie and the jth column corresponds to the ratings for the jth user.

We can now define a vector

$$\mu = [\mu_1, \mu_2, \dots, \mu_{n_m}]$$

such that

$$\mu_i = \frac{\sum_{j:r(i,j)=1} Y_{i,j}}{\sum_j r(i,j)}$$

Which is effectively the mean of the previous ratings for the ith movie (where only movies that have been watched by users are counted). We now can normalize the data by subtracting u, the mean rating, from the actual ratings for each user (column in matrix Y):

As an example, consider the following matrix Y and mean ratings μ:

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 4 & ? & ? & 0 \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}, \quad \mu = \begin{bmatrix} 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$$

The resulting Y' vector is:

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

Large Scale Machine Learning

Learning with Large Datasets

We mainly benefit from a very large dataset when our algorithm has high variance when m is small. Recall that if our algorithm has high bias, more data will not have any benefit.

Datasets can often approach such sizes as m = 100,000,000. In this case, our gradient descent step will have to make a summation over all one hundred million examples. We will want to try to avoid this -- the approaches for doing so are described below.

Stochastic Gradient Descent

Stochastic gradient descent is an alternative to classic (or batch) gradient descent and is more efficient and scalable to large data sets.

Stochastic gradient descent is written out in a different but similar way:

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

The only difference in the above cost function is the elimination of the m constant within $\frac{1}{2}$.

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

J_{train} is now just the average of the cost applied to all of our training examples.

The algorithm is as follows

1. Randomly 'shuffle' the dataset
2. For $i = 1 \dots m$

$$\Theta_j := \Theta_j - \alpha(h_\Theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

This algorithm will only try to fit one training example at a time. This way we can make progress in gradient descent without having to scan all m training examples first. Stochastic gradient descent will be unlikely to converge at the global minimum and will instead wander around it randomly, but usually yields a result that is close enough. Stochastic gradient descent will usually take 1-10 passes through your data set to get near the global minimum.

Mini-Batch Gradient Descent

Mini-batch gradient descent can sometimes be even faster than stochastic gradient descent. Instead of using all m examples as in batch gradient descent, and instead of using only 1 example as in stochastic gradient descent, we will use some in-between number of examples b.

Typical values for b range from 2-100 or so.

For example, with b=10 and m=1000:

Repeat:

For $i = 1, 11, 21, 31, \dots, 991$

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

We're simply summing over ten examples at a time. The advantage of computing more than one example at a time is that we can use vectorized implementations over the b examples.

Stochastic Gradient Descent Convergence

How do we choose the learning rate α for stochastic gradient descent? Also, how do we debug stochastic gradient descent to make sure it is getting as close as possible to the global optimum?

One strategy is to plot the average cost of the hypothesis applied to every 1000 or so training examples. We can compute and save these costs during the gradient descent iterations.

With a smaller learning rate, it is **possible** that you may get a slightly better solution with stochastic gradient descent. That is because stochastic gradient descent will oscillate and jump around the global minimum, and it will make smaller random jumps with a smaller learning rate.

If you increase the number of examples you average over to plot the performance of your algorithm, the plot's line will become smoother.

With a very small number of examples for the average, the line will be too noisy and it will be difficult to find the trend.

One strategy for trying to actually converge at the global minimum is to **slowly decrease α over time**. For example $\alpha = \frac{const1}{iterationNumber + const2}$

However, this is not often done because people don't want to have to fiddle with even more parameters.

Online Learning

With a continuous stream of users to a website, we can run an endless loop that gets (x,y), where we collect some user actions for the features in x to predict some behavior y.

You can update θ for each individual (x,y) pair as you collect them. This way, you can adapt to new pools of users, since you are continuously updating theta.

Map Reduce and Data Parallelism

We can divide up batch gradient descent and dispatch the cost function for a subset of the data to many different machines so that we can train our algorithm in parallel.

You can split your training set into z subsets corresponding to the number of machines you have. On each of those machines calculate $\sum_{i=p}^{q}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$, where we've split the data starting at p and ending at q.

MapReduce will take all these dispatched (or 'mapped') jobs and 'reduce' them by calculating:

$$\Theta_j := \Theta_j - \alpha\frac{1}{z}(temp_j^{(1)} + temp_j^{(2)} + \cdots + temp_j^{(z)})$$

For all $j = 0, \ldots, n$.

This is simply taking the computed cost from all the machines, calculating their average, multiplying by the learning rate, and updating theta.

Your learning algorithm is MapReduceable if it can be *expressed as computing sums of functions over the training set.* Linear regression and logistic regression are easily parallelizable.

For neural networks, you can compute forward propagation and back propagation on subsets of your data on many machines. Those machines can report their derivatives back to a 'master' server that will combine them.

Photo OCR

Photo OCR pipeline

1. Text detection

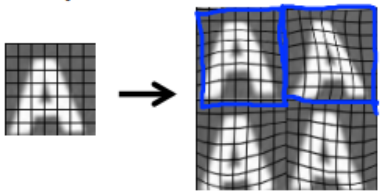
2. Character segmentation

3. Character classification

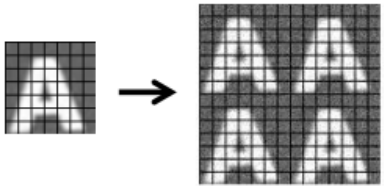
Andrew Ng

Synthesizing data by introducing distortions

- Distortion introduced should be representation of the type of noise/distortions in the test set.



- Audio: Background noise, bad cellphone connection
- Usually does not help to add purely random/meaningless noise to your data.



- x_i = intensity (brightness) of pixel i
- $x_i \leftarrow x_i +$ random noise

[Adam Coates and Tao Wang]

Andrew Ng

Discussion on getting more data

1. Make sure you have a low bias classifier before expending the effort. (Plot learning curves). E.g. keep increasing the number of features/number of hidden units in neural network until you have a low bias classifier.
2. "How much work would it be to get 10x as much data as we currently have?"
   - Artificial data synthesis
   - Collect/label it yourself
   - "Crowd source" (E.g. Amazon Mechanical Turk)

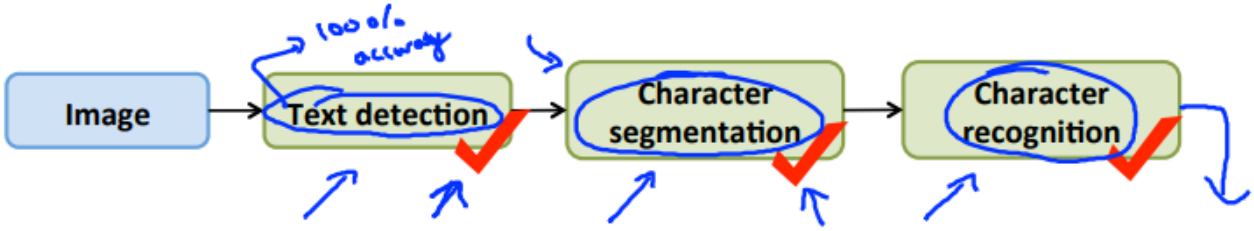
→ Hours?

m = 1,000
→ 10 secs / example
m = 10,000

Machine Learning

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

Estimating the errors due to each component (ceiling analysis)



What part of the pipeline should you spend the most time trying to improve?

Component	Accuracy	
Overall system	72%	↓ 17%
→ Text detection	89%	↓ 1%
Character segmentation	90%	↓ 10%
Character recognition	100%	

Andrew Ng