ML-Class, Fall 2011

Dhruvkaran Mehta
http://www.about.me/dhruvkaran
@dhruvkaran

January 6, 2012

Abstract

These notes capture various aspects taught by Stanford's Prof. Andrew Ng in the inaugural batch of `http://www.ml-class.org` held in the fall of 2011. The sole purpose of these notes, is to patch my personal intuition-holes. They capture what seemed non-trivial to me. Given, my generic background, the same might appeal to a larger population.

Contents

1 Linear Regression with one variable

1.1 Problem definition

Given an input variable `x` (also known as predictor) and output variable `y` (or, prediction), build a model that predicts `y` for a value of `x` that hasn't been seen before.

More concretely,

Input:

$$\{(x^{(i)}, y^{(i)})\})\forall i \in [1, m], x^{(i)} \in \mathbb{R}, y^{(i)} \in \mathbb{R} \tag{1}$$

Output:

$$\{\Theta_0, \Theta_1\} \tag{2}$$

Such that:

$$y \sim \Theta_0 + \Theta_1 \times x \tag{3}$$

Hypothesis:

$$h_\Theta(x) = \Theta_0 + \Theta_1 \times x \tag{4}$$

Goal:

$$\min_{\Theta_0\Theta_1} J(\Theta_0, \Theta_1) \tag{5}$$

Where:

$$J(\Theta_0, \Theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2 \tag{6}$$

1.2 More about the cost function

The cost function used in equation 6 (also known as OLS, or ordinary least squares) is convex and so this minimization problem has a global minimum.

1.3 Gradient Descent

Gradient descent is an optimization algorithm used for convex optimization problems. [In real life, we use advanced optimization techniques like `fminunc` and `fmincg` instead.]

repeat

$$\Theta_j \leftarrow \Theta_j - \alpha \times \frac{\partial J(\Theta_0, \Theta_1)}{\partial \Theta_j} \forall j \in [0, 1] \tag{7}$$

where:

$$\frac{\partial J(\Theta_0, \Theta_1)}{\partial \Theta_j} = \frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)}) \times x_j^{(i)} \tag{8}$$

(For convenience, $x_0 \leftarrow 1$)

until $\{\Theta_0, \Theta_1\}$ converge

Notes:

- Updates to all Θ's happen simultaneously.
- α is the learning rate. Too small $\rightarrow$ slow convergence. Too high $\rightarrow$ oscillations and no convergence
- As the algorithm approaches a minimum, the derivative will be smaller and so the steps will be smaller. One needn't change α.

2 Linear Regression with multiple variables

2.1 Problem Definition

Given n predictors $x_1, x_2, ...x_n$ and outcome y build a model that predicts y for a value of $\mathbf{x}$ $(x_1, x_2, ...x_n)$ that hasn't been seen before.

More concretely,

Input:

$$\{(\mathbf{x}^{(i)}, y^{(i)})\})\forall i \in [1, m], \mathbf{x}^{(i)} \in \mathbb{R}^n, y^{(i)} \in \mathbb{R} \tag{9}$$

Output:

$$\mathbf{\Theta} = \{\Theta_0, \Theta_1, \Theta_2, ..., \Theta_n\} \in \mathbb{R}^{n+1} \tag{10}$$

Such that:

$$y \sim \Theta_0 + \Theta_1 \times x_1 + \Theta_2 \times x_2 + ... + \Theta_n \times x_n \tag{11}$$

Hypothesis:

$$h_\Theta(x) = \Theta_0 + \Theta_1 \times x_1 + \Theta_2 \times x_2 + ... + \Theta_n \times x_n \tag{12}$$

Goal:

$$\min_{\Theta_0, \Theta_1, ..., \Theta_n} J(\Theta_0, \Theta_1, ..., \Theta_n) \tag{13}$$

Where:

$$J(\mathbf{\Theta}) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})^2 \tag{14}$$

2.2 Notes

- For convenience, we define $x_0 \leftarrow 1$. So now, each sample $\mathbf{x} \in \mathbb{R}^{n+1}$.
- So now, $\mathbf{X} \in \mathbb{R}^{m \times (n+1)}, \mathbf{y} \in \mathbb{R}^{m \times 1}, \mathbf{\Theta} \in \mathbb{R}^{(n+1) \times 1}$
- $h_\Theta(\mathbf{x}^{(i)}) = \mathbf{\Theta}^T \mathbf{x}^{(i)}$ $(\mathbf{x}^{(i)} \in \mathbb{R}^{(n+1) \times 1})$

2.3 Cost Function

The cost function and the derivative remain the same as they were with linear regression for one variable.

$$J(\mathbf{\Theta}) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{15}$$

$$\frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(\mathbf{x}^{(i)}) - y^{(i)}) \times x_j^{(i)} \tag{16}$$

2.4 Feature Scaling and Normalization

All features used in the predictor vectors must be **scaled and normalized** for the following reasons:

- There is only one learning rate, α for all features. If the features are not scaled, it is possible that there will be oscillations for the model parameter, Θ corresponding to one feature while the Θ corresponding to another feature might converge too slowly. This could result in no convergence.
- If one feature is way larger than another feature in its typical values, not scaling will mean that the larger feature is dominant in predicting the value of the outcome since the hypothesis is a linear combination of the features. The underlying assumption is that all features are equally important and all Θ's are of comparable magnitudes.

Feature scaling and normalization can be done in the following ways:

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j} \tag{17}$$

or,

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{max_j - min_j} \tag{18}$$

In both those equations, the denominator is a measure of *feature spread* which can be replaced by any other measure that serves the same purpose.

2.5 Debugging

An easy way to debug any linear optimization problem when working with a convex optimization function is to plot *J v/s iterations* curve as seen in figure 1. If the cost function decreases with every iteration, the algorithm is working. Oscillations mean the algorithm is skipping the minimum or hovering around it. One way to fix that is to decrease learning rate, α.

2.6 Normal Equation - the *analytical* solution

This solution for $\mathbf{\Theta}$ can also be found by trying to solve for $\mathbf{\Theta}$ using $\mathbf{X\Theta} = \mathbf{y}$:

$$\mathbf{\Theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{19}$$

Using this equation, there is no need to choose α since it is not iterative. But, it is much slower to use this method if the number of features n is very large. That slows down the computation of the $n \times n$ matrix inverse tremendously. For values of `n` larger than 1000-10000, it is definitely much more advisable to use the gradient descent or some other iterative optimization algorithm.

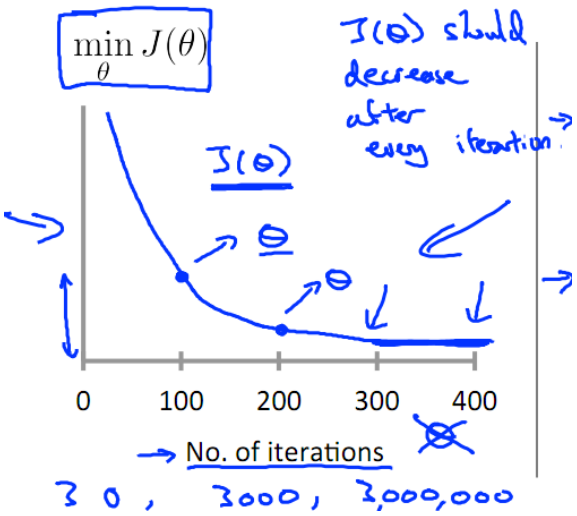

Figure 1: Debugging linear regression

3 Logistic Regression *(Classification)*

3.1 Problem Definition

Given m training examples $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), ..., (\mathbf{x}^{(m)}, y^{(m)})\}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and $y \in \{0, 1\}$, build the model:

$$y = g(\Theta_0 + \Theta_1 \times x_1 + \Theta_2 \times x_2 + ... + \Theta_n \times x_n) > \Delta \tag{20}$$

Hypothesis:

$$h_\Theta(\mathbf{x}) = g(\mathbf{\Theta}^T\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{\Theta}^T\mathbf{x}}} \in (-1, 1) \tag{21}$$

where, $g(\mathbf{z}) = \frac{1}{1+e^{-z}}$, also known as the **sigmoid**, or **logistic** function.
We will try to get $h_\Theta(\mathbf{x})$ to model the probability $p(y = 1; x, \Theta)$ and then use Δ to threshold the probability.

3.2 Decision boundary

The decision boundary is the curve which represents the wall between membership and non-membership. It can be moved by changing the threshold of the decision, Δ. Typically, due to the nature of the sigmoid function, we choose $\Delta \leftarrow 0.5$, since $h_\Theta(\mathbf{x}) > 0.5$ when $\mathbf{\Theta}^T\mathbf{x} > 0$. Although it is important to bear in mind that when trying to play with false positives and negatives, etc. this threshold can help. There are situations where we want absolutely no false positives(cancer detection) and then there are some where we want no false negatives(quality assurance). Δ can help achieve this balance.

In it's current form, the hypothesis 21 can only support linear decision boundaries. To have non-linear boundaries, we can add extra (interactive) features like $x_1^2, x_2^2, x_1x_2, ...$

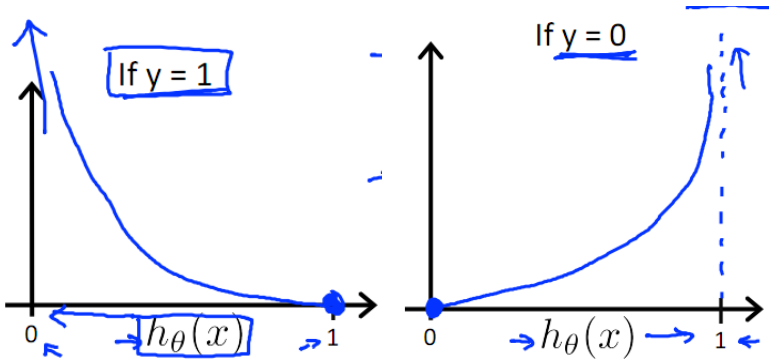

Figure 2: Logistic regression - Cost functions

3.3 Cost function

The cost function for a **single classification** for the logistic regression is:

$$J = -y \times \log(h_\Theta(\mathbf{x})) - (1-y) \times \log(1 - h_\Theta(\mathbf{x})) \tag{22}$$

Looking at this equation carefully, it penalizes y towards the wrong ends heavily. The first term, $-y \times \log(h_\Theta(\mathbf{x}))$ is activated only when $y = 1$ and penalizes being away from the value of 1 heavily (as illustrated in figure 2) and vice versa.

Over all the examples, the cost function is:

$$J(\mathbf{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)} \times \log(h_\Theta(\mathbf{x}^{(i)})) - (1-y^{(i)}) \times \log(1 - h_\Theta(\mathbf{x}^{(i)}))] \tag{23}$$

And the cost derivative remains the same as with linear regression:

We know that for one example, $J = -y \times \log \mathbf{h} - (1-y) \times \log(1-\mathbf{h})$

$$\therefore \frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = \frac{-y}{\mathbf{h}}(-\mathbf{h}^2)\exp^{-\mathbf{\Theta}^T\mathbf{x}}(-x_j) - \frac{(1-y)}{(1-\mathbf{h})}\mathbf{h}^2\exp^{-\mathbf{\Theta}^T\mathbf{x}}(-x_j) \tag{24}$$

$$\therefore \frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = -y\mathbf{h}\exp^{-\mathbf{\Theta}^T\mathbf{x}}x_j - (1-y)\mathbf{h}(x_j) \tag{25}$$

$$\therefore \frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = -yx_j + y\mathbf{h}x_j + \mathbf{h}x_j - y\mathbf{h}x_j = (\mathbf{h}-y)x_j \tag{26}$$

$$\frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(\mathbf{x}^{(i)}) - y^{(i)}) \times x_j^{(i)} \tag{27}$$

3.4 Prediction

Once we know $\mathbf{\Theta} = (\Theta_0, \Theta_1, \Theta_2, ..., \Theta_n)$, we predict using:

$$p(y = 1|\mathbf{x}^{(test)}, \mathbf{\Theta}) = g(h_\Theta(\mathbf{x}^{(test)})) \tag{28}$$

and we predict y = 1 if $p(y = 1|\mathbf{x}^{(test)}, \mathbf{\Theta}) > \Delta$

3.5 Multi-class classification

To use the same technique with `N` classes instead of `2` classes, we would train `N` *one v/s all* classfiers and then **pick the class which maximizes** $h_\Theta(\mathbf{x}^{(test)})$

4 Vectorization

When writing code for machine learning, it is beneficial to make use of the years of research and hard work that has gone into building highly optimized linear algebra libraries on whatever platform is in question. **Vectorized** code that leverages super-fast matrix multiplication, inversion, transpose, etc, usually runs orders of magnitude faster than non-vectorized, "loopy" code. In this section, we will study vectorized implementations of linear and logistic regression

4.1 Framework

`m` training examples, with `n` features in each example.
Input matrix: $\mathbf{X} \in \mathbb{R}^{m \times n}$
For convenience, we add an extra (bias term) feature, $x_0 \leftarrow 1$
So, now, $\mathbf{X} \in \mathbb{R}^{m \times n+1}$
Target Prediction for training, or, output matrix: $\mathbf{y} \in \mathbb{R}^{m \times 1}$
Model Parameters: $\mathbf{\Theta} = (\Theta_0, \Theta_1, ..., \Theta_n) \in \mathbb{R}^{(n+1 \times 1)}$

4.2 Linear Regression

[J, **grad**] = function($\mathbf{\Theta}, \mathbf{X}, \mathbf{y}$)
$\mathbf{err} \leftarrow \mathbf{X\Theta} - \mathbf{y}$
{Matrix multiplication and subtraction to compute errors on all training examples}
$J = \frac{1}{2m} \times \mathbf{err}^T \times \mathbf{err}$
{Vector-vector multiplication to perform a **sum of products** for $\sum_{i=1}^{m} {err^{(i)}}^2$}
$\mathbf{grad} = \frac{1}{m} \times (\mathbf{err}^T \times \mathbf{X})^T$ {But, $(AB)^T = B^T A^T$**, so,** $\mathbf{grad} = \frac{1}{m} \times \mathbf{X}^T \times \mathbf{err}$**}**

4.3 Logistic Regression

[J, **grad**] = function($\mathbf{\Theta}, \mathbf{X}, \mathbf{y}$)
$\mathbf{h} \leftarrow sigmoid(\mathbf{X\Theta})$
{Leveraging a vectorized implementation of sigmoid}
$\mathbf{err} \leftarrow \mathbf{h} - \mathbf{y}$
$J = \frac{-1}{m}[\mathbf{y}^T \times \log(\mathbf{h}) + (1 - \mathbf{y}^T) \times \log(1 - \mathbf{h})]$
{Using a vectorized log function}
$\mathbf{grad} = \frac{1}{m} \times (\mathbf{err}^T \times \mathbf{X})^T$ {But, $(AB)^T = B^T A^T$**, so,** $\mathbf{grad} = \frac{1}{m} \times \mathbf{X}^T \times \mathbf{err}$**}**

4.4 Tricks

There are many more subtle reasons one should strive for completely vectorized code. For instance, if we use vectorized implementations, and later in time, move to a multi-threaded architecture or even a compute cluster, underlying libraries can often utilize modern machines better with parallelization built in with linear algebra libraries. Here we discuss some vectorization tricks:

- To sum over products, always consider vector-vector multiplication
- Most platforms like octave and numpy will let you index a matrix with a vector which can repeat. For instance `eye(k)(<insert_indexing_vector>,:)`

5 Regularization

5.1 Symptom: Overfitting

Overfitting is a common problem in machine learning problems. It's best described as a model which is so tuned to a training set that while it performs really well on that data, it suffers from severe performance penalties on new data, or what is more commonly referred to as a test set. In other words, the model fails to *generalize* it's learnings for data it has not seen before. Overfitting usually occurs due to:

- Number of features, `n` >> the number of training examples, `m`
- Complex interactive features exist in the training predictors. For example, $x_1, x_2, x_1x_2, x_1^2x_2, ...$ These not only increasing the number of features being trained on, but also reduce the SNR(Signal-to-noise ratio) and so the amount of information in the data is really lesser than it seems.

5.2 Cure: Feature Curation and Regularization

To improve the generalization capability of your model:

- Reduce the *number of features* and the *interactivity* of the features. Look out for features which *proxy* the information from each other and dampen the SNR. *Irrelevant* features also make this worse.
- *Regularization* is the technique used to keep the magnitudes of the model parameters in $\mathbf{\Theta}$ i.e. $(\Theta_0, \Theta_1, ..., \Theta_n)$ small. This effectively means, we want to minimize $\|\mathbf{\Theta}\|$ while actually minimizing the error function. Just minimizing $\|\mathbf{\Theta}\|$ by itself will not help.

5.3 Regularization for linear regression

With regularization, the linear regression equation looks like:

$$J(\mathbf{\Theta}) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\Theta_j^2 \tag{29}$$

and if $j > 0$,

$$\frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)}) \times x_j^{(i)} + \frac{\lambda}{m} \times \Theta_j \tag{30}$$

else if j = 0,

$$\frac{\partial J(\mathbf{\Theta})}{\partial \Theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)}) \times x_j^{(i)} \tag{31}$$

Take notice, that the regularization term does not try to suppress the parametric bias, Θ_0 since it is left out of the regularization term.

5.3.1 Regularized linear regression - an interpretation

As per the gradient descent algorithm, we could re-write the above equations to mean:

$$\Theta_j \leftarrow \Theta_j(1 - \frac{\alpha\lambda}{m}) - \frac{\alpha}{m}\sum_{i=1}^{m}[h_\Theta(\mathbf{x}^{(i)} - y^{(i)}] \times x_j^{(i)} \tag{32}$$

As λ increases, it puts pressure on the cost function to reduce $\|\boldsymbol{\Theta}\|$ and that begins to take priority over reducing the actual OLS error. So there is a tradeoff involved.

- As $\lambda \to 0$, the model is more prone to over fitting and less prone to generalization.
- As $\lambda \to \infty$, or more generally, takes on very large values, the cost function pressurizes, $\|\boldsymbol{\Theta}\| \to 0$ and the models is rendered a meaningless underfit which always predicts the value Θ_0. One way to look at extreme generalization is an under fit model.

5.4 Regularization - Normal Equations

This is what the normal equation looks like (remember, its an analytical solution for $\boldsymbol{\Theta}$:

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

To use regularisation, you add a matrix in the $(X^T X)$ term that is an identity matrix with [1,1] set to 0 multiplied by λ:

$$\Theta = (\mathbf{X^T X} + \lambda \begin{pmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix})^{-1}\mathbf{X^T y}$$

5.5 Regularized logistic regression

$$J(\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)} \times \log(h_\Theta(\mathbf{x}^{(i)})) - (1 - y^{(i)}) \times \log(1 - h_\Theta(\mathbf{x}^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\Theta_j^2 \tag{33}$$

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \Theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(\mathbf{x}^{(i)}) - y^{(i)}) \times x_j^{(i)} + \frac{\lambda}{m} \times \Theta_j \tag{34}$$

Remember that like earlier, with advanced optimization functions, we still only need a concave optimization function and its gradient.

6 Neural Networks - Theory and Forward Propagation

6.1 Introduction

The need for neural networks arises from the need for non-linear hypothesis. There are two ways to accomplish a non-linear decision boundary:

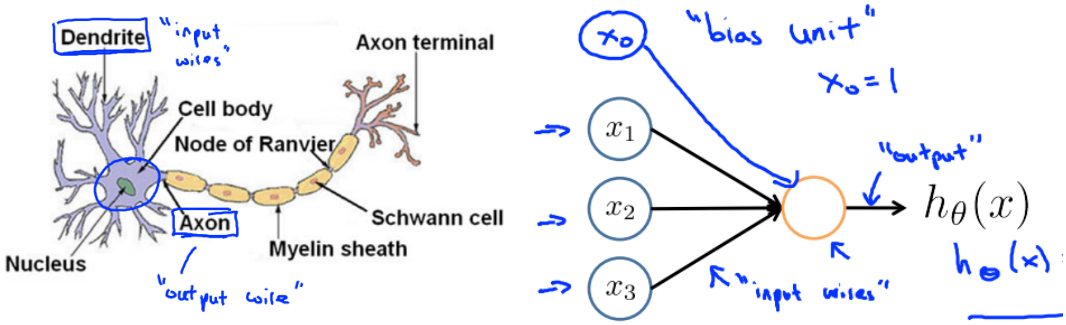

Figure 3: Neuron - in nature and math

- To add non-linear interactive features like x_1x_2, x_1^2 ($x_1^2 + x_2^2$ will give us a circular hypothesis), etc.
- Use systems like neural networks which can model a non-linear decision boundary by *internalizing* non-linear and interactive features(explained later).

The problem with adding interactive terms is that the total number of input features explodes combinatorially.

6.2 Theory

Neural networks in nature, are *networks* made up of *nodes* which have many *incoming connections*, or *inputs* and one *outgoing connection*, or *output*. These nodes, called *neurons* are *activated* based on a *weighted sum of inputs* if the sum is above a *threshold*.

6.3 Terminology

- A neural network has an input layers where the number of neurons is `num_input_features` + 1 and hidden layers which have a similar extra *bias* or intercept unit.
- The output layer has as many neurons as `num_output_classes`
- Inputs: $\{(\mathbf{x}^{(i)}, y^{(i)})\}\forall i \in [1, m], \mathbf{x}^{(i)} \in (R)^n$
- We add the bias input unit and so $\mathbf{X} \in \mathbb{R}^{m\times(n+1)}$
- $a_i^{(j)} \rightarrow$ *activation* of the i^{th} neuron *layer j*. This is the sigmoid of the weighted sum before thresholding. i.e. $h_\Theta(\mathbf{x}) = \frac{\mathbf{1}}{\mathbf{1}+\exp^{-\Theta\mathbf{x}}}$. In other words,

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + ... + \Theta_{1n}^{(1)}x_n) \tag{35}$$

  where $g(z) = sigmoid(z)$.
- $L \rightarrow$ number of layers in the neural network.
- $s_j \rightarrow$ the number of neurons in layer j (not counting the bias unit)
  - $s_0 \leftarrow$ `num_input_features`
  - $\mathbf{\Theta}^{(j)} \in \mathbb{R}^{s_{j+1}\times(s_j+1)}$ (The extra column is adding the bias unit) helps propagate the activations form layer j to j + 1

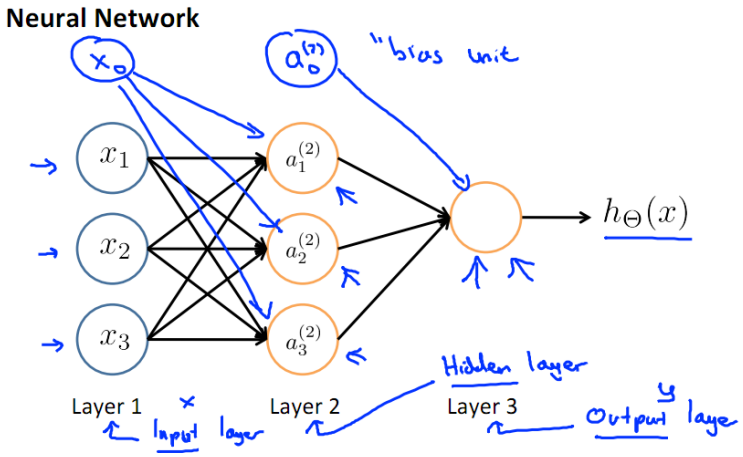

Figure 4: Forward Propagation

6.4 Forward Propagation

In order to understand forward propagation better, let's consider the following scenario: $s_1 = 3, s_2 = 3, s_3 = 1$ (See figure 4), i.e. 3 input features two output classes (one output node can classify between two classes).

{Input layer activations are the inputs.}
$a_0^{(1)} \leftarrow 1, a_1^{(1)} \leftarrow x_1^{(i)}, a_2^{(1)} \leftarrow x_2^{(i)}, a_3^{(1)} \leftarrow x_3^{(i)}$
{For the hidden layers:}
$a_1^{(2)} \leftarrow g(z_1^{(2)}) = g(\Theta_{10}^{(1)} a_0^{(1)} + \Theta_{11}^{(1)} a_1^{(1)} + \Theta_{12}^{(1)} a_2^{(1)} + \Theta_{13}^{(1)} a_3^{(1)})$
$a_2^{(2)} \leftarrow g(z_2^{(2)})$
$a_3^{(2)} \leftarrow g(z_3^{(2)})$
Now, we set $a_0^{(2)} \leftarrow 1$
output $= h_{\mathbf{\Theta}}(\mathbf{x}) \leftarrow g(z_1^{(3)}) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(1)} a_1^{(2)} + \Theta_{12}^{(1)} a_2^{(2)} + \Theta_{13}^{(1)} a_3^{(2)})$

6.4.1 Vectorization

$\mathbf{a}^{(1)} \leftarrow \mathbf{x} \in \mathbb{R}^{4\times 1}$
$\mathbf{\Theta}^{(1)} \in \mathbb{R}^{3\times 4}$
$\mathbf{z}^{(2)} \leftarrow \mathbf{\Theta}^{(1)} \mathbf{a}^{(1)}$
$\mathbf{a}^{(2)} \leftarrow g(\mathbf{z}^{(2)})$
Now, we add the bias $a_0^{(2)} \leftarrow 1$ so $\mathbf{a}^{(2)} \in \mathbb{R}^{4\times 1}$
$\mathbf{\Theta}^{(2)} \in \mathbb{R}^{1\times 4}$
$\mathbf{z}^{(3)} \leftarrow \mathbf{\Theta}^{(2)} \mathbf{a}^{(2)}$
$\mathbf{a}^{(3)} \leftarrow g(\mathbf{z}^{(3)}) \in \mathbb{R}$

6.5 Interpretation and notes

The hidden layer activations are basically non-linear, derived, interactive features which the network learns.

- More than 2 hidden layers are usually unnecessary and the changes which backpropagate into the third hidden are minimal - so not much to be learnt by adding a third hidden layer in most cases.

- Usually, it makes sense to have the same number of units in all the hidden layers. The more the fan out from the input layer to the hidden layers, the more is the number of available *inferred* features.

Another intuitive way to look at neural nets is to see how simple logical functions like AND, OR, XNOR can be implemented using just an input and an output layer.

- For AND: $\mathbf{\Theta} \leftarrow \begin{pmatrix} -15 & 10 & 10 \end{pmatrix}$
- For OR: $\mathbf{\Theta} \leftarrow \begin{pmatrix} 20 & -15 & -15 \end{pmatrix}$

6.6 Multi-class classification using neural networks

We will now try to use neural networks to perform a k-class one-v/s-all style classification. So, instead of $y^{(i)} \in \{0,1\}, \mathbf{y}^{(i)} \in \{0,1\}^K$. $\mathbf{y}^{(i)}$ looks like $\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ Also, $s_L \leftarrow k$

6.7 Cost function

The cost function for neural networks used in classification problems looks very similar to the cost function for the logistic regression.

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)} \log h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k)] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2 \tag{36}$$

Take note of how the regularization term has skipped the bias terms by not allowing for i = 0;

7 Backpropagation algorithm

There is an overarching theme in the algorithms we have discussed so far. The key point in the formulation of the algorithm is the when we find a (convex) cost function function which varies as we change the parameters. We then proceed to understand how this cost would change if we changed each of these parameters up or down (this is the gradient step). Using the gradient, we then alter the parameters to obtain a lower cost.

The cost function for a neural network discussed towards the end of the previous section is no different. The various $\mathbf{\Theta}^{(l)}$ matrices which propagate activations from layer l to the next one are the parameters. (For implementation, we will roll them all out into a long vector and then unroll them again when we want to play with them. This way we can use them with optimization routines like `fminunc` and `fmincg`)

The focus of this section will be to understand how to compute $\frac{\partial J}{\partial \Theta_{ij}^{(l)}}$ and then use that to perform optimization to train the neural network.

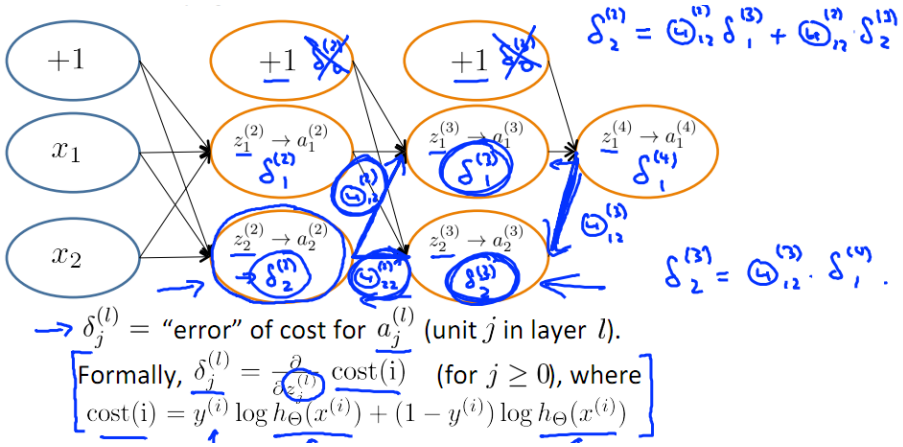

Figure 5: Backpropagation of errors

7.1 Activation forward propagation and cost function

Just to recapitulate, forward propagation takes the input $\mathbf{x}^{(i)}$ and computes the final decisions $\mathbf{h}_{\mathbf{\Theta}}(\mathbf{x}^{(i)})$ using:

$$\begin{aligned}
\mathbf{a}^{(1)} &= \mathbf{x}^{(i)} \\
\mathbf{z}^{(2)} &= \mathbf{\Theta}^{(1)}\mathbf{a}^{(1)} \\
\mathbf{a}^{(2)} &= g(\mathbf{z}^{(2)}) \\
&\vdots \quad \vdots \\
\mathbf{z}^{(L)} &= \mathbf{\Theta}^{(L-1)}\mathbf{a}^{(L-1)} \\
\mathbf{h}_{\mathbf{\Theta}}(\mathbf{x}^{(i)}) = \mathbf{a}^{(L)} &= g(\mathbf{z}^{(2)})
\end{aligned}$$

and, the cost function is:

$$J(\mathbf{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)} \log h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k)] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2 \tag{37}$$

7.2 Error backpropagation - derivate

We need to add some more terminology in order to pursue our goal of $\frac{\partial J}{\partial \Theta_{ij}^{(l)}}$. We denote:

$\delta_j^{(L)} \to$ "error" for *unit* j in *layer* l

To propagate the errors backwards(see fig: 5) from the output layer, to the first hidden layer(input layer has no errors - it is the ground truth), we use the following:

$\delta_j^{(L)} = a_j^{(L)} - y_j$
The vectorized form of saying the same thing is:

$$\delta^{(L)} = \mathbf{a}^{(L)} - \mathbf{y} \tag{38}$$

Backpropagate the error using,

$$\delta^{(L-1)} = (\mathbf{\Theta}^{(L-1)})^T \delta^{(L)} \times g'(\mathbf{z}^{(L-1)}) \tag{39}$$

Now, since,

$$g(\mathbf{z}^{(L-1)}) = \frac{1}{1+\exp^{-\mathbf{\Theta}^{(L-2)}\mathbf{a}^{(L-2)}}} = \frac{1}{1+\exp^{-\mathbf{z}^{(L-1)}}} \tag{40}$$

$$\therefore g'(\mathbf{z}^{(L-1)}) = \frac{1}{1+\exp^{-\mathbf{z}^{(L-1)}2}} \times \exp^{-\mathbf{z}^{(L-1)}} = g(\mathbf{z}^{(L-1)}) \times (1 - g(\mathbf{z}^{(L-1)})) = \mathbf{a}^{(L-1)} \times (1 - \mathbf{a}^{(L-1)}) \tag{41}$$

From 39 and 41, we have:

$$\delta^{(L-1)} = (\mathbf{\Theta}^{(L-1)})^T \delta^{(L)} \times \mathbf{a}^{(L-1)} \times (1 - \mathbf{a}^{(L-1)}) \tag{42}$$

Similarly, we go all the way to:

$$\delta^{(2)} = (\mathbf{\Theta}^{(2)})^T \delta^{(3)} \times \mathbf{a}^{(2)} \times (1 - \mathbf{a}^{(2)}) \tag{43}$$

Ignoring regularization, it turns out that (a more detailed derivation was out of the scope of the course):

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \tag{44}$$

7.3 Backpropagation - Formalization and Vectorization

Given a supervised learning training set, $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), ..., (\mathbf{x}^{(m)}, \mathbf{y}^{(m)})\}$, we want to compute, J according to equation 37 and another set of matrices, $\mathbf{D}^{(l)}$ where $D_{ij}^{(l)}$ holds the value for $\frac{\partial J}{\partial \Theta_{ij}^{(l)}}$ which we saw earlier can be computed as $a_j^{(l)} \delta_i^{(l+1)}$. Let's look into a fairly vectorized implementation of back propagation.

for $l = 1 \rightarrow L-1$ **do**
  $\mathbf{\Delta}^{(l)} \leftarrow zerosLike(\mathbf{\Theta}^{(l)})$
end for
for $i = 1 \rightarrow m$ **do**
  {The outer loop can be vectorized too. Not done here for simplicity.}
  $\mathbf{a}^{(1)} \leftarrow \mathbf{x}^{(i)}$ {First layer activations are the inputs}
  {Forward propagate the inputs through the parametrized implementation}
  for $l = 2 \rightarrow L$ **do**
    $\mathbf{z}^{(l)} \leftarrow \mathbf{\Theta}^{(l-1)} \mathbf{a}^{(l-1)}$
    $\mathbf{a}^{(l)} \leftarrow g(\mathbf{z}^{(l)})$
  end for
  {Now, we compute the error and *backpropagate* it}
  $\delta^{(L)} \leftarrow \mathbf{a}^{(L)} - \mathbf{y}$
  for $l = L-1 \rightarrow 2$ **do**
    $\delta^{(l)} \leftarrow (\mathbf{\Theta}^{(l)})^T \delta^{(l+1)} \times \mathbf{a}^{(l)} \times (1 - \mathbf{a}^{(l)})$
  end for
  for $l = 1 \rightarrow L-1$ **do**
    $\mathbf{\Delta}^{(l)} \leftarrow \mathbf{\Delta}^{(l)} + \delta^{(l+1)} (\mathbf{a}^{(l)})^T$ {Vectorization for: $\Delta_{ij}^{(l)} \leftarrow \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$}
  end for
end for
for $l = 1 \rightarrow L-1$ **do**

$$\mathbf{reg\Theta}^{(l)} \leftarrow \begin{pmatrix} 0 & 1 & \dots & 1 \\ 0 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \dots & 1 \end{pmatrix} \times \mathbf{\Theta}^{(l)}$$

{We don't regularize the bias terms}

$\mathbf{D}_{ij}^{(l)} \leftarrow \frac{1}{m}\mathbf{\Delta}^{(l)} + \lambda \times \mathbf{reg}\mathbf{\Theta}^{(l)}$ {And, then, we know, $D_{ij}^{(l)}$ holds the value for $\frac{\partial J}{\partial \Theta_{ij}^{(l)}}$}

end for

7.4 Numerical gradient checking

When computing the gradient for back propagation, it can be hard to debug errors due to the complex nature of the backpropagation. We compute the gradient vector numerically using the first principle of derivative calculus with a small ϵ:

$$\begin{pmatrix} \frac{\partial J(\Theta_{10},\Theta_{11},...,\Theta_{s_L,s_{L-1}+1})}{\partial \Theta_{10}} = \frac{J(\Theta_{10}+\epsilon,\Theta_{11},...,\Theta_{s_L,s_{L-1}+1}) - J(\Theta_{10}-\epsilon,\Theta_{11},...,\Theta_{s_L,s_{L-1}+1})}{2\epsilon} \\ \frac{\partial J(\Theta_{10},\Theta_{11},...,\Theta_{s_L,s_{L-1}+1})}{\partial \Theta_{10}} = \frac{J(\Theta_{10},\Theta_{11}+\epsilon,...,\Theta_{s_L,s_{L-1}+1}) - J(\Theta_{10},\Theta_{11}-\epsilon,...,\Theta_{s_L,s_{L-1}+1})}{2\epsilon} \\ \vdots \\ \frac{\partial J(\Theta_{10},\Theta_{11},...,\Theta_{s_L,s_{L-1}+1})}{\partial \Theta_{10}} = \frac{J(\Theta_{10},\Theta_{11},...,\Theta_{s_L,s_{L-1}+1}+\epsilon) - J(\Theta_{10},\Theta_{11},...,\Theta_{s_L,s_{L-1}+1}-\epsilon)}{2\epsilon} \end{pmatrix}$$

Then we use the numerically computed gradients with the rolled up vaues for $D_{ij}^{(l)}$. However, once you are done using numerical gradient checking to debug the backpropagation, it is important to switch the debugging off. This numerical gradient check has terrible performance characteristics.

7.5 Notes

Imagine if all the initial values for $\Theta_{ij}^{(l)}$ were symmetrical/identical. In the first iteration, all the activation values will be the same for all units in a given hidden layer or even the output layer. We might as well have only one unit in each hidden unit and the output unit! To break the symmetry, we use **random initialization** for each $\Theta_{ij}^{(l)}$ between $[-\epsilon, \epsilon]$.

There is an important decision to be made for:

- Number of hidden layers: Usually one, or, two. More than two is not quite useful since the backpropagation errors are reduced to insignificant values with three hidden layers.
- Number of units in the hidden layer: Rule of thumb is to use the same number of units in each hidden layer. The more the better. But, that also means more features and a hypothesis that is more prone to over-fitting. So we need to be careful and use regularization.

8 Applied Machine Learning

When working on a machine learning problem it is often hard to decide what to work on next, what's working well, and what needs work and tuning. Sometimes, a seemingly "well-trained" model which has good training accuracy will fail to perform on data it has not seen before because it has *overfit* and fails to *generalize*. We will try to devise *diagnostics* to help with this. Diagnostics help us decide which of the following steps should be taken next:

- Get more data
- Use lesser features
- Use more features
- Add more polynomial/derived/interactive features - basically like using more features.
- Decrease regularization by reducing λ

- Increase regularization by increasing λ

8.1 Hypothesis evaluation

In order to evaluate our hypothesis, we will perform the following train-evaluate cycle:

- Data split: Split the training data into 70% for training set and 30% for test set. So now, we have two sets of data, the training set $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), ..., (\mathbf{x}^{(0.7m)}, y^{(0.7m)})\}$ and the test set, $\{(\mathbf{x}^{(1)}_{test}, y^{(1)}_{test}), (\mathbf{x}^{(2)}_{test}, y^{(2)}_{test}), ..., (\mathbf{x}^{(0.3m)}_{test}, y^{(0.3m)}_{test})\}$.
- Learn the model parameters $\mathbf{\Theta}$ on the training set and compute $J_{test}(\mathbf{\Theta})$. Use the correct cost function depending on the optimization problem at hand - for regression, use, OLS, for classification, use, misclassification error, etc.

Since the model parameters $\mathbf{\Theta}$ are fit to the training set, the *training error*, $J_{train}(\mathbf{\Theta})$ is likely to be lower than the *test error*, $J_{test}(\mathbf{\Theta})$.

8.2 Cross-Validation set

Consider the scenario where we are trying to choose between many model hypothesis like: $h_{\mathbf{\Theta}}(x) = \Theta_0 + \Theta_1 x_1$
$h_{\mathbf{\Theta}}(x) = \Theta_0 + \Theta_1 x_1 + \Theta_2 x_1^2$
...
$h_{\mathbf{\Theta}}(x) = \Theta_0 + \Theta_1 x_1 + \Theta_2 x_1^2 + \Theta_5 x_1^5$
Now, if we use $J_{test}(\mathbf{\Theta})$ to decide which hypothesis to use, we have actually used the test set to fit another *model parameter* viz. the *degree* of the model hypothesis. Reporting this error as the model health will likely be an optimistic estimate as explained in the previous section.

Let's revisit the test split:

- Let's split it as 60% for training data, 20% for test set and 20% for what we will now call the *cross-validation* set. So now, we have three sets of data, the training set $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), ..., (\mathbf{x}^{(0.6m)}, y^{(0.6m)})\}$, the test set, $\{(\mathbf{x}^{(1)}_{test}, y^{(1)}_{test}), (\mathbf{x}^{(2)}_{test}, y^{(2)}_{test}), ..., (\mathbf{x}^{(0.2m)}_{test}, y^{(0.2m)}_{test})\}$ and the cross-validation set: $\{(\mathbf{x}^{(1)}_{cv}, y^{(1)}_{cv}), (\mathbf{x}^{(2)}_{cv}, y^{(2)}_{cv}), ..., (\mathbf{x}^{(0}_{cv}$

This time, we will use the cross-validation set to pick the model parameter i.e. the hypothesis *degree* and then use the test set to *report* the model health. This will avoid any over-optimistic estimates and embarrassment that comes with them.

8.3 Bias and Variance

A model with a high *bias* is one that is under-fitting the dataset, or in other words is too simple, has too few features, etc and is not able to capture all the information in the data into the model. On the other hand, a model with a high *variance* is one which over-fits and fails to generalize well to data it has not seen before.

Let's analyze the scenario of fitting the *degree* of the model hypothesis for a linear regression, such as the one described in the previous section. In this case, we will compute the following errors, for *each* value of the degree of the hypothesis:

$$J_{train}(\mathbf{\Theta}) = \frac{1}{2m} \sum_{i=1}^{m} [h_{\mathbf{\Theta}}(\mathbf{x}^{(i)}) - y^{(i)}]^2 \tag{45}$$

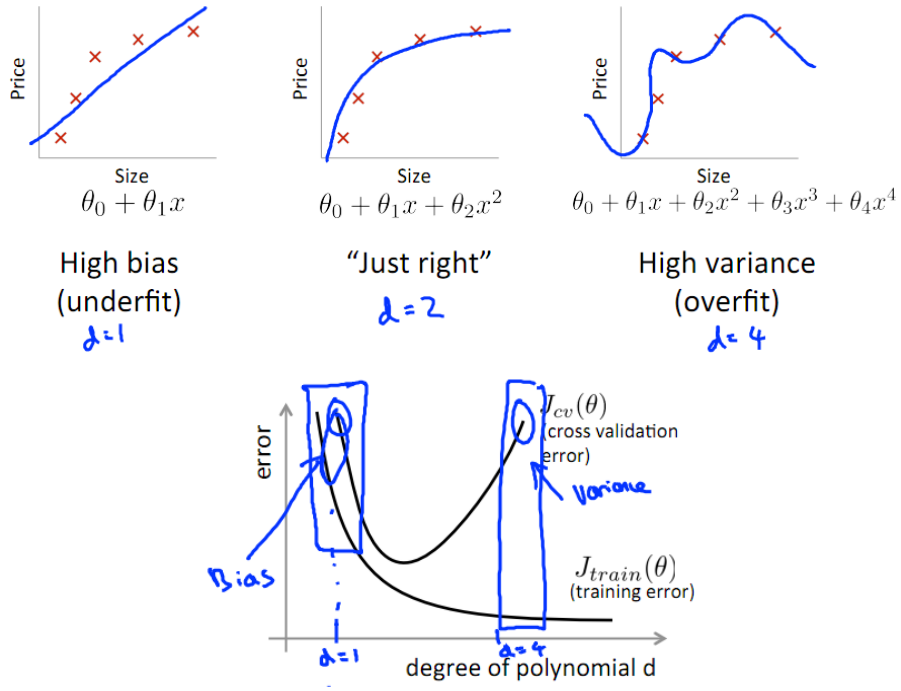

Figure 6: Learning curve - Model hypothesis degree

$$J_{cv}(\mathbf{\Theta}) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} [h_{\mathbf{\Theta}}(\mathbf{x}_{cv}^{(i)}) - y_{cv}^{(i)}]^2 \tag{46}$$

From figure 6, we can tell that the region of high bias or underfit(when the degree of the polynomial is low, and the straight looking lines are not able to capture the complex data), is characterized by: **high,** J_{train}**, high,** J_{cv} **and** $J_{train} \sim J_{cv}$ whereas, the region of high variance or overfit(when the degree of the polynomial is so high, that it captures the noise in the data along with the signals and fails to generalize), is characterized by: **low,** J_{train}**, high,** J_{cv} **and** $J_{train} << J_{cv}$

While our job is to just pick the model parameter(s) that minimizes the cross-validation error and then report the test error, it is important to understand bias and variance in order to diagnose problems in the model.

If we were doing the same study for the regularization parameter λ instead of the model degree, the **learning curves** would look like figure 7

8.4 More about learning curves

It's intuitive that when the size of the training set is small, it is easier to learn and highly likely that the model will over-fit the data. As the number of examples in the training set increases, the tendency to over-fit drops(as there are many more constraints for the model to satisfy), but still, the cross-validation error is likely to be higher than the training error. The learning curves for the training-set size `m` look like figure 8

As before, in under fit(**high bias**) scenario, as m increases, we observe, **high,** J_{train}**, high,** J_{cv} **and** $J_{train} \sim J_{cv}$(See figure: 9)

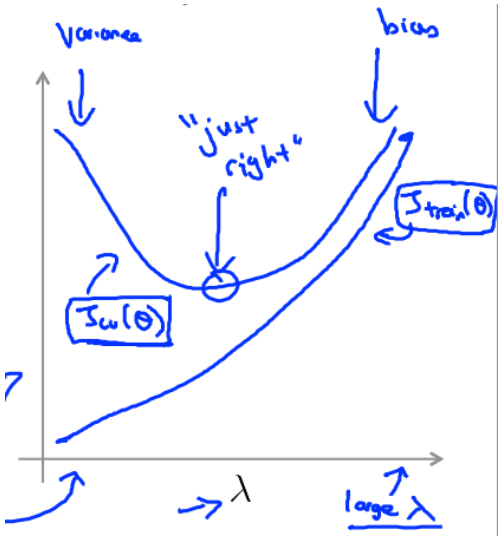

Figure 7: Learning curve - Regularization Parameter

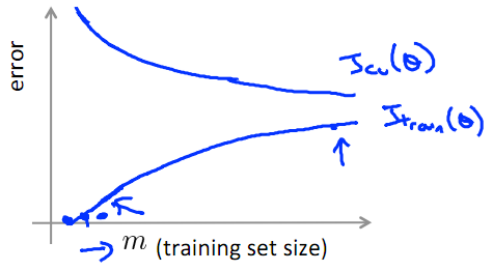

Figure 8: Learning curve - Training set size

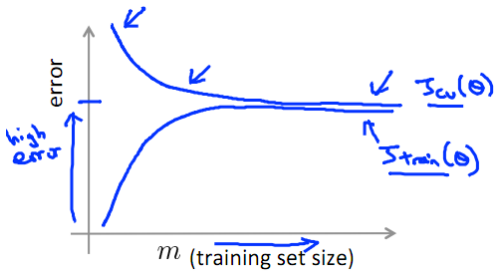

Figure 9: Learning curve - Training set size - High Bias

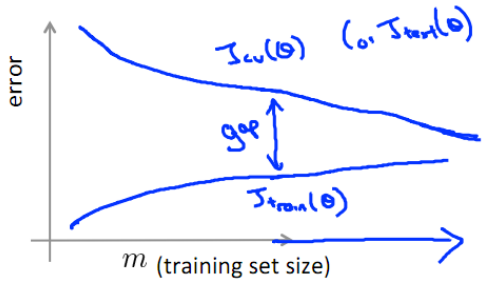

Figure 10: Learning curve - Training set size - High Variance

And in the over-fit(**high variance**) scenario, **low,** J_{train}**, high,** J_{cv} **and** $J_{train} << J_{cv}$ (See figure: 10)

8.5 Remedies from learning curve diagnostics

Getting more data for under-fitted models is useless since they haven't even modeled all the information in the existing training information. Whereas, over-fitted models can benefit from more data.

Next step	Will solve...
Get more data	High variance
Reduce features	High variance
Increase regularization(λ)	High variance
Additional features	High bias
Add polynomial, interactive, derived features	High bias
Decrease regularization(λ)	High bias

In general, it makes sense to start with a very simple model and algorithm, plot the *learning curves*, and then iterate in a typical agile software engineering fashion. This definitely beats starting out with a large assumption and then burning man-hours in the wrong direction.

8.6 Skewed datasets

Skewed datasets present a special challenge. Consider a cancer dataset with 1% positive examples and 99% negative examples. Now, if misclassification error was used as the benchmarking metric in a classification environment, then a classifier which always predicted *negative* for cancer, would have 99% accuracy, or, 1% misclassification! In this situation we need to change the metric which we use to fit the model parameters using the cross-validation error. Only the metric changes, nothing else. We still follow the algorithm-model-learning curve workflow.

8.6.1 $F_1 score$

Let's dissect a cancer classifier's performance using the chart below:

		Actual	
		1	0
Predicted	1	True Positive(tp)	False Positive(fp)
	0	False Negative(fn)	True Negative(tn)

We notice that accuracy or misclassification error can be obtained as:

$misclassification = \frac{fp+fn}{tp+fp+tn+fn}$. The problem here is that the total number of true samples is too low for this to be a meaningful metric. Instead we define:

$$precision = \frac{tp}{tp+fp} \tag{47}$$

$$recall = \frac{tp}{tp+fn} \tag{48}$$

and,

$$F_1 score = \frac{2 \times precision \times recall}{precision + recall} \tag{49}$$

This is a very well defined *tradeoff*. The reader might try to understand how, for instance, varying the threshold, $\mathbf{\Delta}$ for $h_{\mathbf{\Theta}}(\mathbf{x})$ in a logistic regression setup, might change the precision and recall. A cancer classifier needs to be high precision but a low recall might be slightly more manageable since people should not be put on chemotherapy unless absolutely sure. Also, multiple **high precision, low recall** classifiers can be combined for a good output. On the other hand, for quality assurance in a manufacturing environment, we would like a **low precision, high recall** classifier to make sure everything being delivered is perfect.

8.6.2 Synthesizing positive examples

Another way of dealing with skewed datasets is to *artificially synthesize positive samples*. This is possible with problems like PhotoOCR, where we can obtain various fonts, overlay them onto backgrounds of a reasonable contrast and then apply wavelet transforms, etc. Note that adding noise which is not seen in the real world, like gaussian pixelations, will not really help the cause of building a dataset that is not skewed. The added noise has to be realistic.

8.7 Ceiling Analysis

Another way of deciding which component to work next is ceiling analysis. While diagnostic learning curves help us what should be worked on in a particular machine learning algorithm, ceiling analysis can help us decide which component of a machine learning *pipeline* (composed of several machine learning algorithms), should be worked on next. Consider a pipeline like
Picture →Smoothing → Face Detection → Face Recognition. To perform ceiling analysis, we will replace the output of each layer with the perfect(often hand-crafted) dataset as input to the next layer. This will result in a table like:

Perfect	Pipeline accuracy
Nothing	63%
Smoothing	63%
Smoothing, Face Detection	93%
Smoothing, Face Detection, Face Recognition	100%

So now, it becomes clear that the biggest gains are to be obtained with better face detection.

9 Support Vector Machines

Support vector machines are a more modern implementation of a classification technique. In this section we get an opportunity to exhibit several concepts like large margin intuition and kernel techniques.

9.1 An alternative view of logistic regression

The model hypothesis for a logistic regression was:

$$h_{\mathbf{\Theta}}(\mathbf{x}^{(i)}) = \frac{1}{1 + \exp^{-\mathbf{\Theta}^T \mathbf{x}^{(i)}}} \tag{50}$$

The sigmoid function looks like figure 11 So, for logistic regression, we need, $\mathbf{\Theta}^T \mathbf{x} >> 0$ if $y = 1$, and,

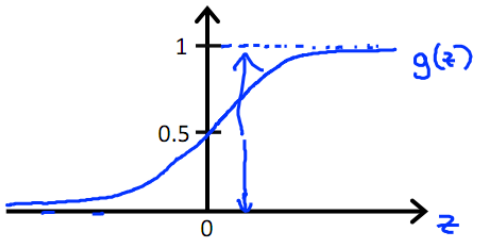

Figure 11: Sigmoid function: $\frac{1}{1+\exp^{-x}}$

$\mathbf{\Theta}^T \mathbf{x} << 0$ if $y = 0$.
The cost function for logistic regression if $J = -y \times \log(h_\Theta(\mathbf{x})) - (1 - y) \times \log(1 - h_\Theta(\mathbf{x}))$ for a given example. Let's re-write this as: $J = y \times cost_1(\mathbf{x}) + (1 - y) \times cost_0(\mathbf{x})$ where $cost_1(\mathbf{x})$ is the cost imposed on the hypothesis prediction when the target y is 1 and vice versa.

Now, we get to the cost function of the Support Vector Machine:

$$J = y \times cost_1(\mathbf{\Theta}^T \mathbf{x}) + (1 - y) \times cost_0(\mathbf{\Theta}^T \mathbf{x}) \tag{51}$$

We replace the previous cost functions with curves which zero out when $\mathbf{\Theta}^T \mathbf{x} > 1$ for y = 1 as shown in figure 12 and when $\mathbf{\Theta}^T \mathbf{x} < -1$ for y = 0 as shown in figure 13.

Now, to classify absolutely correctly, we just need to make sure that we learn model parameters $\mathbf{\Theta}$ such that $\mathbf{\Theta}^T \mathbf{x} > 1$ for y = 1 and $\mathbf{\Theta}^T \mathbf{x} < -1$ for y = 0. This will make more sense as we begin to talk about large-margin intuition in the next section. The overall objective for a support vector machine looks like:

$$\min_{\mathbf{\Theta}} C \sum_{i=1}^{m} [y^{(i)} cost_1(\mathbf{\Theta}^T \mathbf{x}^{(i)}) + (1 - y^{(i)}) cost_0(\mathbf{\Theta}^T \mathbf{x}^{(i)})] + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 \tag{52}$$

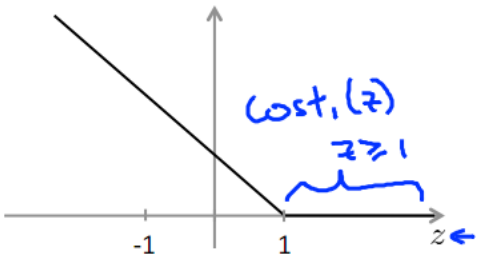

Figure 12: $cost_1(\mathbf{\Theta}^T \mathbf{x})$

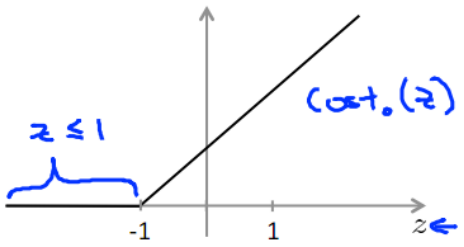

Figure 13: $cost_0(\mathbf{\Theta}^T\mathbf{x})$

Notice how λ has been replaced by another constant C for mathematical convenience. It still plays the role similar to λ, only it's correlation with regularization in inverted.

9.2 Large Margin Intuition

Now, we will examine equation 52 to understand how a support vector machine chooses a decision boundary which maximizes the **margins** from all the classes. For the purposes of this discussion, let's assume that the SVM constant C is set to a very large value. Assume $C \to \infty$. A large value of C acts like a small value of λ - so this is going to be like studying SVM with no regularization; it will be prone to over fitting the data.

As C goes up, the penalty associated with the term with $cost_1$ and $cost_0$ goes up, so the SVM is highly incentivized to make those costs 0, i.e., make $||\mathbf{\Theta}^T\mathbf{x}|| > 1$ - even if doing so has other side effects, like over-fitting.

Assuming a large value of C, we can reformulate equation 52 as:

$$\min_{\mathbf{\Theta}} \frac{1}{2}\sum_{j=1}^{n}\Theta_j^2 \text{ s.t. } \mathbf{\Theta}^T\mathbf{x}^{(i)} > 1 \text{ if } y^{(i)} = 1 \text{ and } \mathbf{\Theta}^T\mathbf{x}^{(i)} < -1 \text{ if } y^{(i)} = 0 \tag{53}$$

in other words:

$$\min_{\mathbf{\Theta}} \frac{1}{2}\sum_{j=1}^{n}\Theta_j^2 \text{ s.t. } ||\mathbf{\Theta}^T\mathbf{x}^{(i)}|| > 1 \tag{54}$$

9.2.1 Understanding dot-product

Vector *dot product* is basically the process of *projecting* one vector onto another.

$$\vec{u}\cdot\vec{v} = ||\vec{u}||||\vec{v}||cos(\phi) = ||\vec{p}||||\vec{v}|| \text{ where } \vec{p} \text{ is the projection of } \vec{u} \text{ onto } \vec{v} \tag{55}$$

The next thing to remember is that $\vec{\mathbf{\Theta}}$ is the **normal** vector to the decision boundary at each point. Hence, $\mathbf{\Theta}^T\mathbf{x}$ which is the dot-product can be re-written as, $\vec{\mathbf{\Theta}}\cdot\vec{\mathbf{x}}$. What the un-regularized SVM is really trying to do if make sure that $\vec{\mathbf{\Theta}}\cdot\vec{\mathbf{x}}$, or the projection of the data vector ($\vec{\mathbf{x}}$) onto the normal vector ($\vec{\mathbf{\Theta}}$) is *maximized*. (Since, Trying to make $\mathbf{\Theta}^T\mathbf{x}^{(i)} > 1$ means, making $\vec{p^{(i)}}||\vec{\mathbf{\Theta}}|| > 1$) With this incentive, along, trying to $\min_{\mathbf{\Theta}} \frac{1}{2}\sum_{j=1}^{n}\Theta_j^2$ i.e. $\min_{\mathbf{\Theta}} \frac{1}{2}||\vec{\mathbf{\Theta}}||^2$. This translates to making the projection of $\vec{\mathbf{x}}^{(i)}$ onto $\vec{\mathbf{\Theta}}$ large and gives the SVM, a large margin property (See fig: 14).

A "perfect"(or, in other words, unregularized) SVM, however, will try to get a perfect decision boundary because the penalty on misclassification is too high. **Lowering that penalty(C), will decrease the**

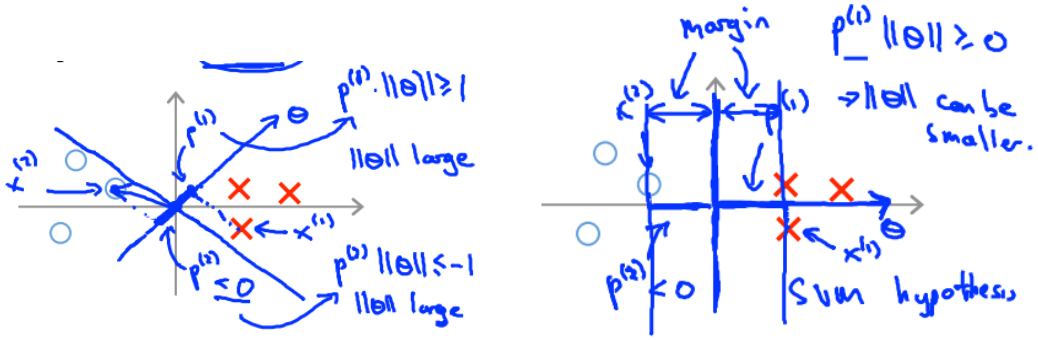

Figure 14: SVM - large margin intuition

large margin property but increase regularization. Hence, C gives us a knob into modeling v/s generalization and must be studied with learning curves just as λ was.

9.3 Kernels

The kernel trick is what makes SVMs special. It's possible to use this trick to make several mathematical optimizations possible. The kernels trick projects the data onto a higher-dimensional space where it is possible to find a linear decision boundary.

Say we have `n` features $(x_1, x_2, ..., x_n)$. The kernel trick is to replace: $(x_1, x_2, ..., x_n)$ with new derived features $(f_1, f_2, ..., f_k)$ where $f_j^{(i)} = similarityFunction(\mathbf{x}^{(i)}, \mathbf{l}_j)$. To do this, we first need to choose the *landmark* points, $(l_1, l_2, ..., l_k)$ where $l_k \in \mathbb{R}^n$, the same space as the input examples. Usually, the *similarityFunction* is gaussian:

$$similarityFunction(\mathbf{x}^{(i)}, \mathbf{l}_j) = \exp(\frac{-||\mathbf{x}^{(i)} - \mathbf{l}_j||^2}{2\sigma^2}) \tag{56}$$

This function has the value of 1 at the landmark location and varies in a gaussian fashion all the way to 0 as we move far away from the landmark. This particular similarity function(also known as a *kernel*) introduces another model parameter we need to figure out a value for: σ, the spread of the similarity.

The model now predicts $h_{\mathbf{\Theta}}(\mathbf{x}) = 1$ when $\Theta_0 + \Theta_1 f_1 + \Theta_2 f_2 + ... + \Theta_n f_n > 1$ and $h_{\mathbf{\Theta}}(\mathbf{x}) = 0$ when $\Theta_0 + \Theta_1 f_1 + \Theta_2 f_2 + ... + \Theta_n f_n < -1$ (in accordance with $cost_1$ and $cost_0$).

The outstanding question then is how do we go about choosing the *landmark* points, $\{l_1, l_2, ..., l_k\}$. We have to think about the landmark points as magnets(landmarks) which emit a magnetic field(decision boundary field) and the resulting magnetic field(decision boundary) is a combination of those emitted by the individual magnets. We can also think of the spread, σ as the magnetic strength of the pole(fig: 15).

As a side note, there are many kernels which people use. If someone says they are using a *linear* kernel, it means they are not using landmark-expansion. i.e. $\mathbf{f} = \mathbf{x}$. Not any similarity function can be a valid kernel. A valid kernel has to satisfy a theorem called *Mercer's Theorem.* Some valid kernels are:

- Polynomial kernel: $similarityFunction(\mathbf{x}, \mathbf{l}) = (\mathbf{x}^T\mathbf{l})^d, (\mathbf{x}^T\mathbf{l} + const)^d, ...$
- String kernel
- Chi-square kernel

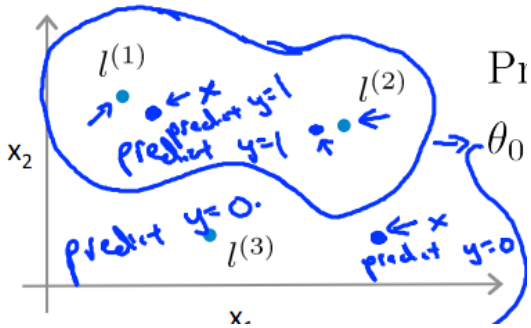

Figure 15: SVM - Gaussian kernel - Do you see the magnetic field?

9.3.1 The trick!

The SVM kernel trick is to set $(\mathbf{l}_1, \mathbf{l}_2, ..., \mathbf{l}_m) = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)})$here, k=m. So now:

$$\mathbf{f}^{(i)} = (similarityFunction(\mathbf{x}^{(i)}, \mathbf{l}_1), similarityFunction(\mathbf{x}^{(i)}, \mathbf{l}_2), ..., similarityFunction(\mathbf{x}^{(i)}, \mathbf{l}_m)) \quad (57)$$

which makes for,

$$\mathbf{f}^{(i)} = (similarityFunction(\mathbf{x}^{(i)}, \mathbf{x}^{(1)}), similarityFunction(\mathbf{x}^{(i)}, \mathbf{x}^{(2)}), ..., similarityFunction(\mathbf{x}^{(i)}, \mathbf{x}^{(m)})) \quad (58)$$

We have transformed $\mathbf{x}^{(i)} \in \mathbb{R}^{n+1} \rightarrow \mathbf{f}^{(i)} \in \mathbb{R}^{m+1}$ (still have the bias term). Basically, the new features denote how similar the input example is to each of the other input examples. *It's like dropping a north magnetic pole at each positive example and a south magnetic pole at each negative example.*

And now we will predict using $\mathbf{\Theta}^T\mathbf{f}$, where $\mathbf{\Theta}$, will be trained using the cost function:

$$\min_{\mathbf{\Theta}} C \sum_{i=1}^{m} [y^{(i)} cost_1(\mathbf{\Theta}^T\mathbf{f}^{(i)}) + (1 - y^{(i)}) cost_0(\mathbf{\Theta}^T\mathbf{f}^{(i)})] + \frac{1}{2}\mathbf{\Theta}^T\mathbf{\Theta} \quad (59)$$

(Vectorization note: $\mathbf{\Theta}^T\mathbf{\Theta} = \sum_{j=1}^{n} \Theta_j^2$ and $n = m$)

In most numerical implementations, $\mathbf{\Theta}^T\mathbf{\Theta}$ is actually implemented as $\mathbf{\Theta}^T\mathbf{M}\mathbf{\Theta}$ and this is what helps tremendously speed up this matrix multiplication. Since $\mathbf{\Theta} \in \mathbb{R}^{m+1}$(m is the number of training examples, which can easily be very large), computing $\mathbf{\Theta}^T\mathbf{\Theta}$ is very slow without this trick. This numerical trick does not work well with the cost function used for logistic regression and renders the kernel trick useless in that scenario.

9.4 Discussion about SVM parameters

- $C(\sim \frac{1}{\lambda})$: High C $\rightarrow$**(Low bias, High variance)**; Low C $\rightarrow$ **(High bias, Low variance)**. Typical value: ~ 1
- σ: **High** σ means that the features vary more smoothly. Prone to under fitting. Leads to **high bias, low variance**. Low σ means features vary less smoothly, **low bias, high variance**, more prone to over-fitting.

9.5 Don't re-invent the wheel

Due to the complex nature of the mathematical optimizations involved in support vector machines, it usually makes sense to use existing packages and libraries like: `liblinear/libsvm` to get the job done. They only need a valid kernel and values for model parameters, C, λ to be plugged in.

9.6 Notes

When using a gaussian kernel, it is very important to run **feature scaling and normalization** before running the SVM algorithm. It should be obvious to the reader(from equation 56) how non-normalized features could give more importance to some features than others.

To perform multi-class classification across k-classes using SVMs, we will have to implement k-one v/s all svm classifiers.

Neural networks can be used in almost all situations where an SVM will do well - but they are likely to be much slower to train.

9.7 Logistic Regression v/s Support Vector Machines

Different situations call for different techniques. When considering the choice between logistic regression and support vector machines, keep the following in mind:

$n >> m$	$n = 10,000, m = 10..1000$	Too many features. Prone to over-fitting(high variance). Use Logistic regression or an SVM(**without a kernel**)
n is small. m is intermediate	$n = 1..1000, m = 10..10,000$	Use SVM(**with a gaussian/RBF kernel**)
n is small. m is large	$n = 1..1000, m = 50,000+$	Try to add more features. Prone to under-fitting(high bias). Use Logistic regression or an SVM(**without a kernel**)

10 Clustering

Clustering is a unsupervised technique to form groups of input examples. The training set, $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, ..., \mathbf{x}^{(m)}\}$ has no labels. The goal is define clusters into one of which, each of the `m` input examples belong. Clustering is used for market segmentation and analysis, social network analysis and datacenter health studies amongst many other applications.

10.1 k-Means

Input: $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, ..., \mathbf{x}^{(m)}\}$, and the number of clusters, K Output: $\{\boldsymbol{\mu}^{(1)}, \boldsymbol{\mu}^{(2)}, \boldsymbol{\mu}^{(3)}, ..., \boldsymbol{\mu}^{(K)}\}$, the K centroids for the c-clusters.

Note: This time, $\mathbf{x} \in \mathbb{R}^n$. There is no bias term, x_0.

10.2 Algorithm

Randomly initialize all $\{\boldsymbol{\mu}^{(1)}, \boldsymbol{\mu}^{(2)}, \boldsymbol{\mu}^{(3)}, ..., \boldsymbol{\mu}^{(K)}\}$
while all $\boldsymbol{\mu}$'s have not converged **do**

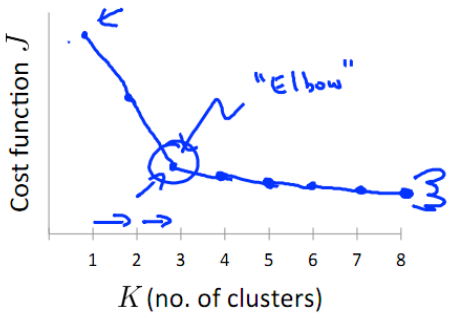

Figure 16: The elbow method

{Cluster assignment step}
for $i = 1 \rightarrow m$ **do**
  for all $k = 1 \rightarrow K$ **do**
    $dist^{(k)} \leftarrow ||\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)}||$
  end for
  $c^{(i)} \leftarrow index$ such that $dist^{(index)}$ is minimized.
end for{Move the centroids}
for $k = 1 \rightarrow K$ **do**
  $\boldsymbol{\mu} \leftarrow mean(\mathbf{x}^{(i)}) \forall i \in \{c^{(i)} == k\}$
end for
end while

The only awkward requirement for the k-means algorithm is knowing the number of clusters, K up front. For data that is not visually well separated, this number should come from downstream requirements like "how many t-shirt sizes do I want to make?"

10.3 Optimization objective

The cost function for each step of the k-means algorithm can be specified as:

$$J(\mathbf{c}^{(1)}, \mathbf{c}^{(2)}, ..., \mathbf{c}^{(m)}, \boldsymbol{\mu}^{(1)}, \boldsymbol{\mu}^{(2)}, ..., \boldsymbol{\mu}^{(K)}) = \frac{1}{m} \sum_{i=1}^{m} ||\mathbf{x}^{(i)} - \boldsymbol{\mu}_{\mathbf{c}^{(i)}}||^2 \tag{60}$$

In the *cluster-assignment* step (when the points are all assigned to one of the clusters), this cost function is minimized w.r.t $\{\mathbf{c}^{(1)}, \mathbf{c}^{(2)}, ..., \mathbf{c}^{(m)}\}$ whilst the $\boldsymbol{\mu}$'s are held constant and in the *centroid-move* step (when we recompute the cluster centroids form the newly assigned points), we minimize it w.r.t. $\{\boldsymbol{\mu}^{(1)}, \boldsymbol{\mu}^{(2)}, ..., \boldsymbol{\mu}^{(K)}\}$ when the $\mathbf{c}$'s remain unchanged.

Another way to decide the number of clusters is the elbow method (See fig: 16). We pick the number of clusters at which the cost function drops suddenly. This usually finds a natural number of clusters. But, often, as the reader might expect, an elbow is not to be found in real world data.

10.4 Notes

- The random initialization has a large impact on the way the k-means algorithm performs and it is often advised to set the initial centroids to K randomly chosen examples.

- The k-means cost function is **not** convex. The algorithm can get stuck in local minimums when the random initialization is a terrible one. This problem usually manifests itself with bad clusters which do not make sense even visually. The way to get around this problem is to run the algorithm several hundred times with a different random initialization each time, and record the cost function at convergence. Finally, we pick the clusters from the run with the minimum final cost.

11 Principal component analysis

PCA is a method to reduce the *dimensionality* of a dataset while maintaining the *information* in it. Sometimes, we need to take a dataset with 100s of features and reduce the number of features. Usually the underlying motivation is:

- To make a already well performing machine learning algorithm run faster by reducing the number of features and hence the size of the involved matrices.
- To bring the data down to 2 or 3 dimensions so it can be visualized. Take this with a grain of salt. Visualizing features you don't know the meaning of, will only take you so far. Still useful for deciding the number of clusters, etc.
- Data compression

11.1 Problem formulation

Given inputs $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}\}$ where $\mathbf{x}^{(i)} \in \mathbb{R}^n$, find a transformation matrix, such that the projection of any input using the matrix is $\mathbf{z}^{(i)} \in \mathbb{R}^k$ where $k < n$.

i.e.

Find $\mathbf{U_{red}}$ s.t. $\mathbf{U_{red}} \in \mathbb{R}^{k \times n}$, $\mathbf{z}^{(i)} = \mathbf{U_{red}} \times \mathbf{x}^{(i)}$ such that the projection error is minimized. The projection error is the error in the data recovered in the `n` dimensional space using the transpose of the original transformation. So, if $\mathbf{x}^{(i)}_{approx} = \mathbf{U_{red}}^T \times \mathbf{z}^{(i)}$, the objective for PCA is:

$$\min_{\mathbf{U_{red}}} \frac{1}{m} \sum_{i=1}^{m} ||\mathbf{x}^{(i)} - \mathbf{x}^{(i)}_{approx}||^2 \tag{61}$$

11.2 PCA is not linear regression

Often , there's confusion between linear regression and PCA. The difference is illustrated in figure 17. While the linear regression cost function is trying to minimize the *y-distance* between the actuals and the prediction, PCA is trying to minimize the *projected* distance.

11.3 Pre-processing

Before starting to run principal component analysis, it is important to perform **feature scaling and normalization** using $x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$ where s is some measure of feature spread, like standard deviation, variance, or range($max_j - min_j$).

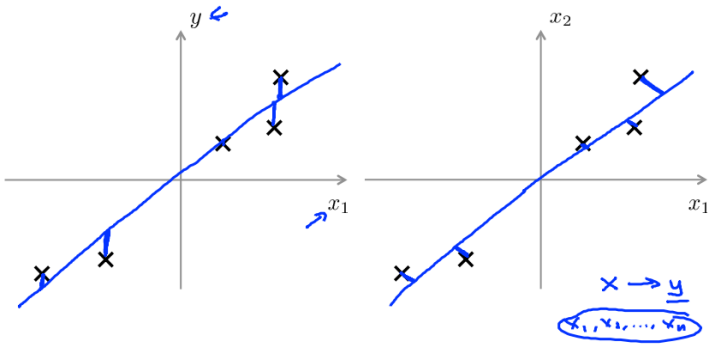

Figure 17: PCA is not linear regression

11.4 Algorithm

As a first step, we compute the covariance matrix for the input data.

$$\mathbf{\Sigma} = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}^{(i)})(\mathbf{x}^{(i)})^T \tag{62}$$

This is another reason the feature normalization is important. This step actually assumes that the features are zero-meaned. Remember covariance computation? Also, this can be vectorized as $\mathbf{\Sigma} = \frac{1}{m}\mathbf{X}^T\mathbf{X}$ where $\mathbf{X} \in \mathbb{R}^{m\times n}$. The covariance matrix $\mathbf{\Sigma} \in \mathbb{R}^{n\times n}$ has the following properties:

- All the elements are positive. The matrix is symmetric across its diagonal.
- The diagonal elements, Σ_{ii} represent the variance of the i^{th} feature.
- The non-diagonal elements, $\Sigma_{ij} = \Sigma_{ji}$, represent the co-variance of i^{th} and the j^{th} features. The magnitude of the co-variance represents how often the two features vary together. The sign of the same denotes if the features vary in the same direction(Positive v/s negative correlation).

The idea central to PCA is to come up with `n` **new** features which are sorted by how much information there is in each of the new features. **Intuitively, the spread of a feature is its importance. The more the variance or spread of the feature, the more it tells us about the input.**. Think about it this way; if a feature does not vary at all(i.e. has a constant value for all inputs), its variance is 0. It's useless to include this feature in the analysis. On the other hand a feature that has a wide-spread across inputs, adds a lot on information. A real, varying value of the feature, tells us a story about the input example.

We use the *singular value decomposition* technique to do this. While an in-depth discussion of SVD is out of scope, here's an intuition about it. SVD takes the covariance matrix $\mathbf{\Sigma}$ as it's input. Then, it begins searching across various *new* axes, for a new direction such that when all the data is projected onto this direction, the spread is maximized. Having picked one, axes, SVD, now proceeds to do the same, only this time, the possible axes are reduced to the set that is orthogonal to the first one. Finally, we obtain a set of `n` *eigenvectors*, or what is also a *basis* for the new vector space.

More concretely, we run:

$$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = svd(\mathbf{\Sigma}) \tag{63}$$

The matrix $\mathbf{U}$ represents eigenvectors of the new space:

$$\begin{pmatrix} | & | & | & \dots & | \\ | & | & | & \dots & | \\ | & | & | & \dots & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 & \dots & \mathbf{u}_n \\ | & | & | & \dots & | \\ | & | & | & \dots & | \\ | & | & | & \dots & | \end{pmatrix} \tag{64}$$

The dimensions corresponding to $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, ..., \mathbf{u}_n\}$ have decreasing amounts of spread/information.

The matrix $\mathbf{S}$ is the covariance matrix for the *new* features. It has some interesting properties:

- Non-diagonals elements, $\mathbf{S}_{ij}$ are all almost 0. Almost none of the new features vary with each other.
- The values of the diagonals elements, $\mathbf{S}_{ii}$ are larger than in $\mathbf{\Sigma}_{ii}$. The variance of the new features is larger.
- As we go down the diagonal, the magnitude of $\mathbf{S}_{ii}$ decreases as i increases.

Our next job is to pick a number $k < n$ and then use $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, ..., \mathbf{u}_k\}$ as the new basis for projection into the reduced space.

$$\mathbf{U_{red}} = \begin{pmatrix} | & | & | & \dots & | \\ | & | & | & \dots & | \\ | & | & | & \dots & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 & \dots & \mathbf{u}_k \\ | & | & | & \dots & | \\ | & | & | & \dots & | \\ | & | & | & \dots & | \end{pmatrix}^T \tag{65}$$

With, $\mathbf{U_{red}} \in \mathbb{R}^{k \times n}, \mathbf{X} \in \mathbb{R}^{m \times n}$, we compute the new training data, $\mathbf{Z} \in \mathbb{R}^{m \times k}$ as:

$$\mathbf{Z} = \mathbf{X} \times \mathbf{U_{red}}^T \tag{66}$$

Approximations of the original data can be re-composed using the same transformation as:

$$\mathbf{X_{approx}} = \mathbf{Z} \times \mathbf{U_{red}} \tag{67}$$

11.5 Choosing the number of principal components

The *average squared projection error* for PCA is: $\frac{1}{m}\sum_{i=1}^{m} ||\mathbf{x}^{(i)} - \mathbf{x}_{approx}^{(i)}||^2$ and the total variance in the data set is given by: $\frac{1}{m}\sum_{i=1}^{m} ||\mathbf{x}^{(i)}||^2$. We need to choose 'k' such that most variance in the data is preserved. So, we need to choose k, such that:

$$\frac{\frac{1}{m}\sum_{i=1}^{m} ||\mathbf{x}^{(i)} - \mathbf{x}_{approx}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m} ||\mathbf{x}^{(i)}||^2} < \epsilon \tag{68}$$

To maintain 99% of the variance, set, $\epsilon = 0.01$. Since the diagonal elements of $\mathbf{S}$ are the variance numbers in the new space, we need to make sure that

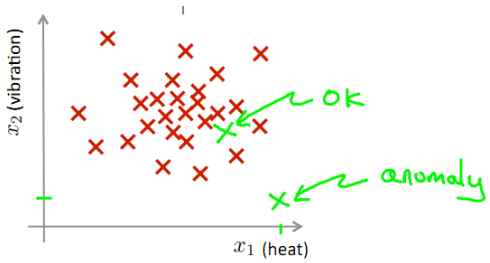

Figure 18: Anomaly in the data

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m} S_{ii}} > (1 - \epsilon) \tag{69}$$

We could also compute various values of $\mathbf{U_{red}}$ for various values of k and then compute the optimal k using equation 68, but since SVD provides the new co-variance matrix, this is easier.

11.6 Applications

- Speed-up an already well performing supervised learning algorithm. **Use only the training set** to compute $\mathbf{U_{red}}$. Then use it for everything else including new predictions. It is now a part of the learnt pipeline.
- Data compression and storage
- Visualization
- **Do not use PCA to reduce the number of features in order to prevent over-fitting**. PCA will retain most of the information in the features. This will not actually help. PCA does not use the information in the labels $y^{(i)}$. Regularization does a better job for this motive. Try everything else, get the pipeline working and then use PCA to add a speed-up.

12 Anomaly Detection

Anomaly detection is a form of unsupervised classification used to find outlier input examples in a given set. It is used in various applications including fraud detection, manufacturing defect detection, quality assurance and monitoring datacenter machines. (See fig: 18)

12.1 Problem formulation

Given $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}\}$, decide if $\mathbf{x}_{test}$ is anomalous ($y = 1$ if $p(\mathbf{x}_{test}) < \epsilon$).

12.2 Modeling

The underlying assumption is that we know the distribution from which the various features are drawn. For the discussions in the text, we will use the gaussian distribution as the assumption. So we hypothesize that:

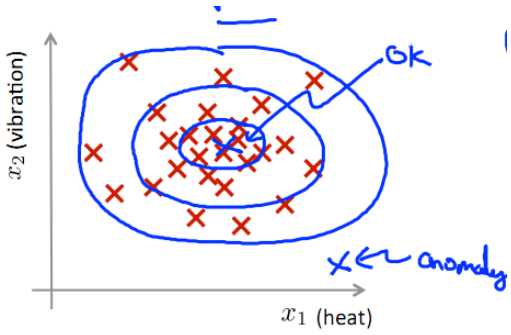

Figure 19: Gaussian Anomaly - top view

$$x_j \sim \mathcal{N}(\mu_j, \sigma_j^2) \; \forall j \in [1, n] \tag{70}$$

The probability density function for the gaussian distribution is:

$$p(\mathbf{x}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp^{\frac{-(\mathbf{x}-\mu)^2}{2\sigma^2}} \tag{71}$$

12.3 Algorithm

Choose features x_j which might be indicative of anomalous examples.
for $j \leftarrow 1 \rightarrow n$ **do**
$\mu_j \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$
$\sigma_j^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$
end for

Now, predict the $p(\mathbf{x}_{test})using$,

$$p(\mathbf{x}_{test}) = p(x_{test,1}; \mu_1, \sigma_1^2) \times p(x_{test,2}; \mu_2, \sigma_2^2) \times ... \times p(x_{test,n}; \mu_n, \sigma_n^2) = \prod_{j=1}^{n} p(x_{test,j}; \mu_j, \sigma_j^2) \tag{72}$$

And then predict an anomaly, i.e. y = 1 if $p(\mathbf{x}_{test}) < \epsilon$ (See fig: 19)

12.4 Algorithm Evaluation

It's very important that we find a way to evaluate the algorithm we just formulated. We will try to do that in this section using a *labeled dataset*, $\{(\mathbf{x}^{(i)}, y^{(i)})\}$.

- Split the dataset into 60% training, 20% cross-validation and 20% test data. Assume that all the training data in normal, non-anomalous(y = 0).
- Now, fit the models for $x_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$
- Use the cross-validaton set to decide a value for ϵ which leads to a good $F_1 score$, precision, recall, misclassification error, or whatever metric suits the situation.
- Report the evaluation metric on the test set with the chosen *model parameter* ϵ.

12.5 Anomaly detection and supervised classification

The next logical question is why would one not just use logistic regression or a support vector machine. Why are we learning an altogether new technique which does the same thing?

Anomaly Detection(Unsupervised)	**Classification(Supervised)**
Use when $\|\|\{y == 1\}\|\| << \|\|\{y == 0\}\|\|$	Better used when the dataset is more or less balanced in volume for each of the classes. $\|\|\{y == 1\}\|\| \sim \|\|\{y == 0\}\|\|$
Models what's *normal* and then declares as anomalous anything that does not fit the model.	Tries to model *each* class separately.
Future anomalies can look different from past ones so long as they don't look *normal*	Past anomalies must reflect any kind that may occur in the future.(This is hard, for instance, in a low volume manufacturing setup, but works when we have tons of data, like in email spam classification.)

12.6 Choosing features to use

We need to tackle the problem of picking features which take on outlier values in anomalous cases. Ones which have an 'elbow' in $p(\mathbf{x})$ at a certain boundary. Consider the case when the features are for computers in a datacenter with features like cpu usage, disk, memory, and network traffic. Now, if most of these run web servers, cpu and network traffic are going to be highly correlated. So, while cpu and network traffic don't really indicate machine performance, the feature $\frac{Cpu}{NetworkTraffic}$ does. It will take on **very low values** if the software on the server is not working as intended and **very high values** if the hardware is malfunctioning. **The trick, is to pick out features which take on very small or very large values in anomalous cases.**

Another thing to keep in mind is to perform **feature transformation** if they are not distributed in a gaussian fashion. While this is not a large problem even if they aren't, a number of simple transforms like $x_1 \leftarrow \log(x_1)$, $x_2 \leftarrow \log(x_2 + const)$, $x_3 \leftarrow \sqrt{x_3}$ can lead us to *derived* gaussian features.

12.7 Multivariate gaussian distribution

The gaussian modeling discussed thus far has been a particular manifestation which assumes that the chosen features are all independent to each other. It has a *feature independence* assumption. As we saw in the previous section, this is not always true. cpu and network traffic are often correlated, and such an assumption makes it harder to choose features.

Another way to look at this is that the feature independence assumption leads to a combined model which can only model ellipses which have axes parallel to the feature axes.(See fig: 20 for a contour projection plot)

Sometimes, we need to model data which has correlations and looks more like fig: 21.

The original model fails to capture this kind of *feature co-variance*.

12.8 Modeling

Nothing changes. We just model $\boldsymbol{\mu} \in \mathbb{R}^n$ and $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$ ($\boldsymbol{\Sigma}$ is the **co-variance matrix**) for all features at the same time. And the pdf is:

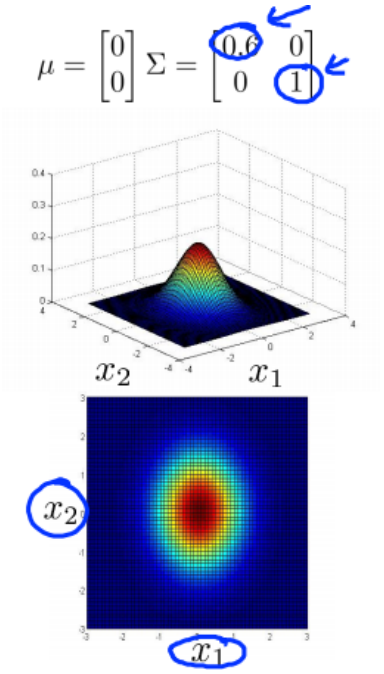

Figure 20: Non-correlated features

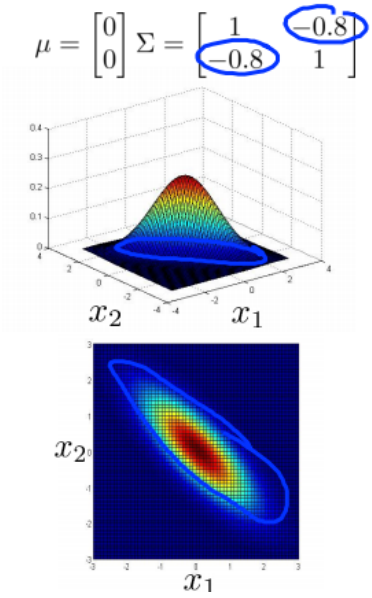

Figure 21: Correlated features

$$p(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{2\pi^{\frac{n}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}}} \exp^{\frac{-1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})} \tag{73}$$

where,

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}^{(i)} \tag{74}$$

and,

$$\boldsymbol{\Sigma} = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}^{(i)}-\boldsymbol{\mu})(\mathbf{x}^{(i)}-\boldsymbol{\mu})^T \tag{75}$$

A closer look reveals that the real difference in what gets modeled is that this time we capture the non-diagonal terms, Σ_{ij} which is the *covariance* of the i^{th} and j^{th} features. If all the non-diagonal terms were 0 as in:

$$\begin{pmatrix} \sigma_1^2 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2^2 & 0 & \dots & 0 \\ \vdots & 0 & \ddots & \dots & 0 \\ \vdots & \vdots & \dots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \sigma_n^2 \end{pmatrix}$$

this model would be identical to the original one.

12.9 Comparison

- The feature independence assumption makes the original model computationally cheaper and it usually works out just fine.
- In the case where $m < n$, the co-variance matrix Σ is *non-invertible* and only the original model is feasible.
- On the flip side, the multi-variate model captures co-variance and makes for a easier time in picking the features.

13 Recommender systems

Consider the problem of recommending great products or movies to a user on the website. Let's lay the terminology groundwork. The data looks like:

Movie	**User 1**	**User 2**	...	**User n_u**
Movie 1	5	4	...	1
Movie 2	?	2	...	5
⋮	⋮	⋮	...	⋮
Movie n_m	2	2	...	?

Number of users: n_u, number of movies: n_m. $r(i,j) = 1$ iff movie i is rated by user j else it is 0. And lastly, $y^{(i,j)}$ is the rating given to movie i by user j.

13.1 Applications

- Predict the rating for a movie a user hasn't rated yet.
- Offer the user another product, given which product they already bought/viewed.

13.2 Content based recommendations

For a moment, let's assume that someone gave us a set of *features* for each movie. Something along the lines of *romance, comedy, action, thriller, foreign, etc.* Couldn't we learn a set of model parameters the *each* user, $\mathbf{\Theta}^{(j)}$, using linear regression with each of the *rated* movies (where $r(i,j) = 1$) as the input examples? Consider: $\mathbf{\Theta}^{(j)} \rightarrow$ prediction parameters for user j and $\mathbf{x}^{(i)} \rightarrow$features for movie i. Once we learn the model for user j, we can predict their rating for an unrated movie using:

$$prediction = (\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} \tag{76}$$

This is linear regression, and so we have the bias term i.e. $\mathbf{\Theta}^{(j)} \in \mathbb{R}^{n+1}$ where n is the number of available features. Let's say that the number of movies rated by user j are $m^{(j)}$. The per-user linear regression has the objective:

$$\min_{\mathbf{\Theta}^{(j)}} \frac{1}{2m^{(j)}} \sum_{i:r(i,j)=1} [(\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)}]^2 + \frac{\lambda}{2m^{(j)}} \sum_{k=1}^{n} (\Theta_k^{(j)})^2 \tag{77}$$

Combining all the per-user regressions, we have,

$$\min_{\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}} J(\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}) \tag{78}$$

where,

$$J(\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}) = \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} [(\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)}]^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\Theta_k^{(j)})^2 \tag{79}$$

(Notice how the $m^{(j)}$ term is missing. Turns out that constant does not really affect the algorithm.)

13.3 Collaborative filtering

Now, we approach the case where the movies don't have input features, but all users have told us how much they like comedy and hate romance. It is possible to perform a inverted linear regression and back out what we *think* the feature values for a movie should be.

So, given $\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}, learn, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}$

Similar to equation 77, this time, the objective function, for a single movie, is:

$$\min_{\mathbf{x}^{(i)}} \frac{1}{2} \sum_{j:r(i,j)=1} [(\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)}]^2 + \frac{\lambda}{2m} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{80}$$

and similar to 82 that for many movies, is:

$$\min_{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}} J(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}) \tag{81}$$

where,

$$J(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}) = \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} [(\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)}]^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{82}$$

The process of collaborative filtering is just to repeat these two processes, with some ratings missing and no given features, over and over until convergence, as in: $\mathbf{\Theta} \rightarrow \mathbf{x} \rightarrow \mathbf{\Theta} \rightarrow \mathbf{x} \rightarrow \mathbf{\Theta} \rightarrow \dots$

13.4 Algorithm

The alternating process described in the previous section can be combined into one optimization objective by treating all of $\{\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}\}$ as *model parameters*. The minimization objective for collaborative filtering is:

$$\begin{aligned} J(\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}) = \quad & \tfrac{1}{2} \textstyle\sum_{(i,j):r(i,j)=1} [(\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)}]^2 + \\ & \tfrac{\lambda}{2} \textstyle\sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \\ & \tfrac{\lambda}{2} \textstyle\sum_{j=1}^{n_u} \sum_{k=1}^{n} (\Theta_k^{(j)})^2 \end{aligned} \tag{83}$$

The derivatives required for gradient descent or advanced optimization techniques are:

$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{(i,j):r(i,j)=1} \{[(\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)}] \Theta_k^{(j)} + \lambda x_k^{(i)}\} \tag{84}$$

and,

$$\frac{\partial J}{\partial \Theta_k^{(j)}} = \sum_{(i,j):r(i,j)=1} \{[(\mathbf{\Theta}^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)}] x_k^{(i)} + \lambda \Theta_k^{(j)}\} \tag{85}$$

13.4.1 Formal Algorithm

Initialize $\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}$ to small random values.
Use gradient descent or an advanced optimization technique along with equations: 83, and 84, 85, to train model parameters $\mathbf{\Theta}^{(1)}, \mathbf{\Theta}^{(2)}, ..., \mathbf{\Theta}^{(n_u)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n_m)}$

- Now, we can predict the rating for a user, movie using $pred = (\mathbf{\Theta}^{(j)})^T x^{(i)}$
- We can also tell the similarity between two movies, by using, $||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||$ as the distance between them.

13.5 Vectorization: Low rank matrix factorization

We are given the matrices:

$$\mathbf{Y} = \begin{pmatrix} y^{(1,1)} & y^{(1,2)} & \dots & y^{(1,n_u)} \\ y^{(2,1} & y^{(2,2)} & \dots & y^{(2,n_u)} \\ \vdots & \vdots & \dots & \vdots \\ y^{(n_m,1} & y^{(n_m,2)} & \dots & y^{(n_m,n_u)} \end{pmatrix}$$

$$\mathbf{PRED} = \begin{pmatrix} (\mathbf{\Theta}^{(1)})^T x^{(1)} & (\mathbf{\Theta}^{(2)})^T x^{(1)} & \dots & (\mathbf{\Theta}^{(n_u)})^T x^{(1)} \\ (\mathbf{\Theta}^{(1)})^T x^{(2)} & (\mathbf{\Theta}^{(2)})^T x^{(2)} & \dots & (\mathbf{\Theta}^{(n_u)})^T x^{(2)} \\ \vdots & \vdots & \dots & \vdots \\ (\mathbf{\Theta}^{(1)})^T x^{(n_m)} & (\mathbf{\Theta}^{(2)})^T x^{(n_m)} & \dots & (\mathbf{\Theta}^{(n_u)})^T x^{(n_m)} \end{pmatrix}$$

$$\mathbf{X} = \begin{pmatrix} \dots & \dots & (\mathbf{x}^{(1)})^T & \dots & \dots \\ \dots & \dots & (\mathbf{x}^{(2)})^T & \dots & \dots \\ \vdots & \vdots & \dots & \vdots & \\ \dots & \dots & (\mathbf{x}^{(n_m)})^T & \dots & \dots \end{pmatrix}$$

and,

$$\mathbf{\Theta} = \begin{pmatrix} \dots & \dots & (\mathbf{\Theta}^{(1)})^T & \dots & \dots \\ \dots & \dots & (\mathbf{\Theta}^{(2)})^T & \dots & \dots \\ \vdots & \vdots & \dots & \vdots & \\ \dots & \dots & (\mathbf{\Theta}^{(n_u)})^T & \dots & \dots \end{pmatrix}$$

The vectorized format of predictions is: $\mathbf{PRED} = \mathbf{X}\mathbf{\Theta}^T$. In the linear algebra world, the matrix **PRED** bears a property called *low rank*.

13.6 Mean normalization

Let's consider a new user, who has not rated any movies yet. In the collaborative filtering algorithm according to equation 83, $\mathbf{\Theta}^{(j)}$ will turn out to be all zeros(also, think about a movie that has not been rated by anyone) and we will be unable to recommend movies to the new user(or, recommend a new movie to any user). This can be over come by normalizing all the ratings across the mean for that movie, like so:

$$y(i,j) \leftarrow y^{(i,j)} - \mu^{(i)} \tag{86}$$

and then perform predictions, using

$$prediction = (\mathbf{\Theta}^{(j)})^T x^{(i)} + \mu^{(i)} \tag{87}$$

Now, all new users will have predicted ratings equal to the mean rating for that movie. Which does make sense.

14 Large scale machine learning

Often the amount of data which a machine learning scientist deals with is large enough that it warrants special techniques, both to save time and for us to be able to iterate faster.

14.1 Stochastic gradient descent

In our study of gradient descent we were updating *all* the model parameters after computing the cost and the gradient across *all* the inputs. In organizations which have a lot of data, this is a problem since one cannot monitor progress until the algorithm has run through all thirty million examples at least once. Stochastic gradient descent solves this problem. The only change is that we update *all* model parameters with the evaluation of cost for *each* input example.

while Model parameters have not converged(change $< \epsilon$) **do**
  $\mathbf{x} \leftarrow$ next example
  $\Theta_j \leftarrow \Theta_j - \alpha \frac{\partial J}{\partial \Theta_j} \forall j$
end while

where we set the cost(for linear regression) for a single example according to:

$$J(\mathbf{\Theta}; \mathbf{x}, y) = cost(\mathbf{\Theta}; \mathbf{x}, y) = \frac{1}{2}(h_{\mathbf{\Theta}}(\mathbf{x}) - y)^2 + \frac{\lambda}{2}\sum_{j=1}^{n} \Theta_j^2 \tag{88}$$

- This cost function is not convex and the descent can get stuck in a local optimum. However this is usually not a problem. This algorithm however, will not show a *constantly* decreasing cost for each iteration. In order to monitor it, plot the cost, averaged for every 1000 iterations. If that curve is constantly decreasing, you're golden.
- Also, since it does one example at a time, stochastic gradient descent doesn't quite converge at the minimum. It keeps hovering at a very acceptable solution. If we want more convergence, one thing to do is to reduce the learning rate α with the iterations with something like: $\alpha \leftarrow \frac{const}{number + const}$

Another variant of the stochastic gradient descent is the *mini-batch gradient descent* where instead of *one* example at a time or *all* examples each time, we choose a *mini-batch* of b examples, such that $1 <= b <= m$.

14.2 Online learning

A lot of business models like information retrieval (search), dynamic pricing, etc. require that a machine learning system be able to learn from data as soon as it comes into the system. Batch learning algorithms fail here. But, stochastic gradient descent work in this scenario.

14.3 Batch gradient descent and mapreduce

Another way to speed up batch-gradient descent is to use mapreduce. One can shard all the inputs, and compute the *gradient* and *cost* (both of which are summations over the examples) distributed over the shards. Then these can be combined to move the algorithm one step forward(model parameters are changed) on a master shard. This distributed algorithm can also benefit from multi-core architecture on modern machines.

The thing to remember is - if the primary step in an algorithm is distributable, like a summation, it can likely benefit a speed-up from distributed computing.