Machine Learning Standford course - Summary

Gi Wah Dávalos Loo

November 25, 2018

1 Introduction

1.1 Definition

Modern definition by Tom Mitchell:

> "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**."

Example: Email span classifier

$E =$ Watching email labeling spam or not
$T =$ Classifiying emails as spam or not
$P =$ Fraction of emails correctly classified as spam or not

1.2 Types

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

1.2.1 Supervised Learning

We are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output. Supervised learning problems are categorized into:
1.*Regression*: tries to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function.

2.*Classification*: tries to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Regession	**Classification**
Linear Regression	Logistic Regression

1.2.2 Unsupervised Learning

Allows us to approach problems with little or no idea what our results should look like. 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.
With unsupervised learning there is no feedback based on the prediction results.

2 Linear Regression

enri The main idea of linear regression is to predict some continuous value from input (x_i) using a function that has a linear behaviour ($\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n = \theta^T x$). The **goal** is to find the most accurate coefficients for the linear function we must reduce the error as much as possible, in other words, we gotta minimize the cost function.

2.1 Types

2.1.1 Univariable

When we have only 1 feature. The linear function for univariable regression should look like this:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1}$$

2.1.2 Multivariable

When we have more than 1 feature. The multivariable linear regression function should look like this:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n \tag{2}$$

2.2 Cost Function

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

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

2.3 Notation (Matrixes and Vector approaches)

In order to make operations more code/cpu "friendly" we represent our set of data and variables on matrixes and vectors.

$$\begin{aligned} x_j^{(i)} &= \text{value of feature } j\text{in the } i^{th} \text{ training example} \\ x^{(i)} &= \text{the input (features) of the } i^{th} \text{ training example} \\ m &= \text{The number of training examples} \\ n &= \text{The number of features} \end{aligned}$$

$$\theta_{(n\times 1)} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \quad x_{(n\times 1)} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad X_{(m\times n)} = \begin{bmatrix} x_0^1 & x_1^1 & x_2^1 \dots x_n^1 \\ x_0^2 & x_1^2 & x_2^2 \dots x_n^2 \\ x_0^3 & x_1^3 & x_2^3 \dots x_n^3 \\ \vdots & \vdots & \vdots \\ x_0^m & x_1^m & x_2^m \dots x_n^m \end{bmatrix} \quad y_{(m\times 1)} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

$$\text{definition format: } h_{(\theta)} = \begin{bmatrix} \theta_0 & \theta_1 \dots \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

$$\text{solution format: } h_{(\theta)} = \begin{bmatrix} x_0^1 & x_1^1 & x_2^1 \dots x_n^1 \\ x_0^2 & x_1^2 & x_2^2 \dots x_n^2 \\ x_0^3 & x_1^3 & x_2^3 \dots x_n^3 \\ \vdots & \vdots & \vdots \\ x_0^m & x_1^m & x_2^m \dots x_n^m \end{bmatrix} \begin{bmatrix} \theta_0 & \theta_1 \dots \theta_n \end{bmatrix} = X\theta \qquad (3)$$

$$\text{cost function: } J(\theta) = \frac{1}{2m}(X\theta - y)^2 \tag{4}$$

Note 1: x_0 is a *"hack"* and it's value is always 1. It's for making both (θ and x) same size in order to multiply them.

Note 2: The univariable and multivariable $h_{(\theta)}$ can be solved using $X\theta$

2.4 How to obtain θ_j

The idea is to minimize the **Cost Function** (minimize the error). In order to do that, we need to build an equation by comparing the derivative of J (gradient) to cero.

$$min(J(\theta)) = \frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} = 0$$

We got 2 ways to approach this problem:

1. *Gradient Descent*
2. *Normal Equation*

2.4.1 Gradient Descent

It is an iterative algorithm that, with a fixed learning rate (α), updates all values of θ simultaneously and decreases the Cost Function. The learning rate (α) is adjusted by try/error. The initial values to test should be on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on).

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

Since it's an iterative algorithm we want the steps to be really small in order to make the computation time a lot faster. We care about the relationship between entries, not the actual values; that why we tend to make the mean zero. For that we got 2 methods:

1. *Feature Scaling*: Involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1.

2. *Mean normalization*: Involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero.

$$x_i = \frac{x_i - \mu_i}{s_i}$$

Where μ_i is the average of all the values for feature (i) and s_i is the range of values (max - min), or s_i is the standard deviation.

Note that dividing by the range, or dividing by the standard deviation, give different results.

2.4.2 Normal Equation

This is the analytical way to tackle this problem. We actually solve the equation $\frac{\partial J(\theta)}{\partial \theta_j} = 0$. There is no need to do feature scaling with the normal equation.

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

2.4.3 Comparison

The main differences between this two approaches:

Gradient Descent	**Normal Equation**
Need to choose alpha	No need to choose alpha
Needs many iterations	No need to iterate
$O(kn^2)$	$O(n^3)$, need to calculate inverse of $X^T X$
Works well when n is large	Slow if n is very large

3 Logistic Regression

The main idea of logistic regression is to predict from a set of discrete binary values (ex: have cancer or not [0, 1]). In order to acomplish that, we use a linear function (Decision boundary, can be non-linear) to map all predictions greater than 0.5 as 1 and all less than 0.5 as a 0. This **prediction** is done by a **sigmoid** $(g(x))$ function that normalizes the output of the **Decision boundary** $(\theta^T x)$ to just 1 or 0.

So in conclusion, there is a **hypotheses** ($h_\theta(x)$) function that is the sigmoid ($g(x)$) function applied to the Decision boundary; and the **goal** is to find the most accurate parameters for the Decision boundary ($\theta^T x$).

3.1 Concepts

3.1.1 Sigmoid function

Also called *Logistic function*, 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. The function has this form:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

And looks like:

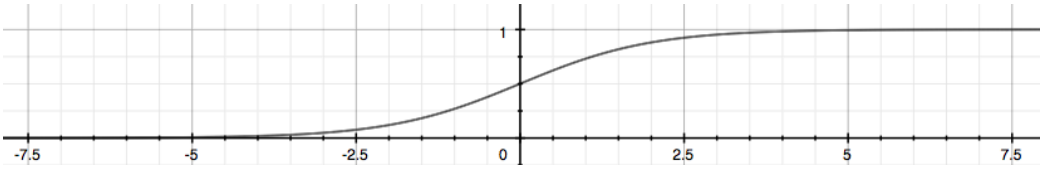

Figure 1: Sigmoid function

3.1.2 Hypotheses function

This hypotheses function will predict, using some input data (x_i) and a set of precomputed coefficients (θ), a number between 0 and 1.

$$0 \leq h_\theta(x) \leq 1$$

In order to achieve this, we use the sigmoid function to map our output to 0 - 1 interval on the decision boundary. The function has this form:

$$h_\theta(x) = g(\theta^T x) = g(X\theta) \tag{6}$$

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

The result of this function can be interpreted as *the probability that the 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%).

3.1.3 Decision boundary

This function describes the line (can be polynomial) that determines the limit of the features in order to predict whether 0 or 1. The representation for the decision boundary is:

$$\theta^T x$$

In order to visualize it, we need to separate the ones that are $y = 0$ and the ones that are $y = 1$, then plot all features. But all axis must be the features $(x^{(j)})$ displayed separately the *0's* from the *1's*
And it can have many forms, also polynomial.

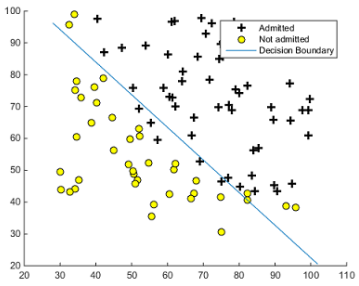

Figure 2: Linear Decision boundary

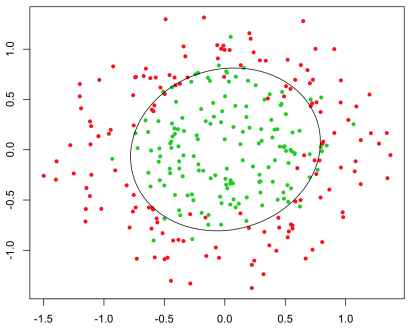

Figure 3: Cuadratic Decision boundary

We can interpret the output of the hypotheses function as:

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

If we observe Figure 1 we can observe that in order to get $g(z) \geq 0.5$ we need z to be greater than 0, in other words:

$$h_\theta(x) = g(\theta^T x) \geq 0.5 \rightarrow y = 1$$
$$\text{when } z = \theta^T x \geq 0$$

Remember:

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

3.2 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.

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

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 $J(\theta)$ is convex for logistic regression.

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

But there is a much simpler and compat way to represent that conditional:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} -y^{(i)}log(h_\theta(x^{(i)})) - (1 - y^{(i)})log(1 - h_\theta(x^{(i)})) \qquad (7)$$

And this is the vectorized implementation:

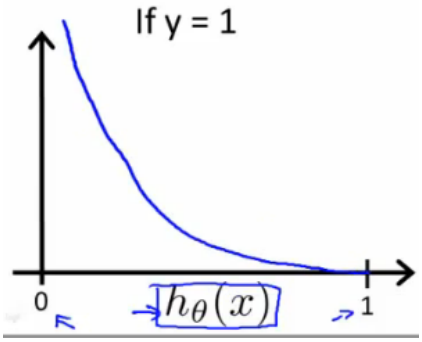

Figure 4: $-log(h_\theta(x))$

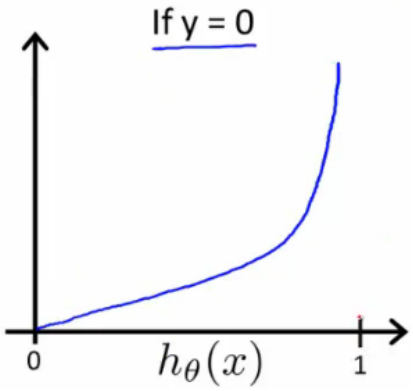

Figure 5: $-log(1 - h_\theta(x))$

$$J(\theta) = \frac{1}{m}\left(-y^T log(g(X\theta)) - (1-y)^T log(1 - g(X\theta)))\right) \tag{8}$$

3.3 How to obtain θ_j

The idea, as in linear regression, is to minimize **Cost Function** (minimize the error). To do that we got many methods:

- *Gradient Descent*
- *Conjugate Gradient*
- *BFGS*
- *L-BFGS*

In order to use this any of these methods we need our function to return 2 things:

1. *Value of cost*: It is just the scalar value of the cost function.
2. *Gradient*: It is the derivative of the cost function ($\frac{\partial J(\theta)}{\partial \theta_j}$)

Note: For academic and learning purposes, we are going to only see the Gradient Descent.

3.3.1 Gradient Descent

The technique is the same as linear regression, and it has the form:

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

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

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:

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

3.4 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 $y = \{0, 1 \dots n\}$.

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

$$\begin{aligned}
&y \in \{0, 1 \dots n\} \\
&h_\theta^{(0)}(x) = P(y = 0 | x; \theta) \\
&h_\theta^{(1)}(x) = P(y = 1 | x; \theta) \\
&h_\theta^{(2)}(x) = P(y = 2 | x; \theta) \\
&\vdots \\
&h_\theta^{(n)}(x) = P(y = n | x; \theta) \\
&\text{prediction } = \max_i (h_\theta^{(i)}(x))
\end{aligned}$$

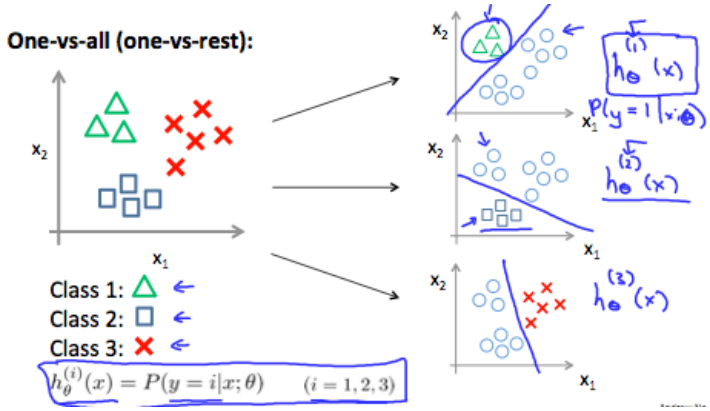

Figure 6: One-vs-all multiclass classification

3.5 Underfitting & Overfitting

When we are choosing the *Decision Boundary* or the parameters of the *Linear Regression*, and we realize that the function does not fit very well the data. In order to make a better function, we decide to add extra features with different polynomial behaviours; for example, a 5^{th} order polynomial. We observe that even though the fitted curve passes through the data perfectly, we would not expect this to be a good predictor. Now, we got 2 cases: **underfitting** and **overfitting**.

1. **Underfitting**: Also called *high bias*, is when the form of the hypotheses function (h_θ) maps poorly to the trend of the data, it means, the function is too simple to predict an accurate value. One reason is because there are very few parameters and cannot generalize very well, thus it cannot make an accurate prediction.

2. **Overfitting**: Also called *high variance*, is when the function fits perfectly the available data but does not generalize very 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.

There are 2 ways to tackle this issue:

1. Reduce the number of features

   - Manually select wich features to keep.
   - Use model selection algorithm.

2. Regularization: Keep all the features, but reduces magnitude of parameters θ_j. Works well when we have a lot of slightly useful features.

3.6 Regularization

The main goal of regularization is to make the curves less sharp, because we got overfitting problems. In order to do that we gotta reduce the magnitude of θ. As we can see in Figure 7, the function is a more simple with lower curves, which means that now the model can predict with better results by making the parameters to weight more in the cost function (later penalized by the gradient descent algorithm for being too high) and as a result it will reduce it's magnitude.

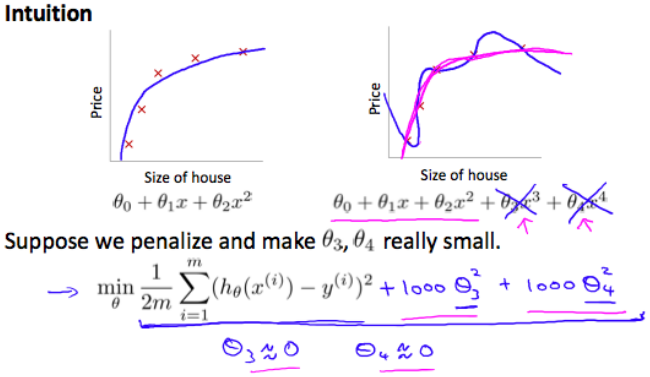

Figure 7: Regularization concept

To generalize the way to add weight to the parameters (θ), we use a **regularization parameter** lambda (λ) multiplied by the sum of the square of the value of the parameters (θ).

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

Using the cost function with the extra summation above, 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. If lambda is too small it will cause overfitting as well.

3.6.1 Regularized Linear Regression

The form of the regularization for the cost function is: $\frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$. We **won't apply regularization on the first parameter (bias)**. So the gradient descent looks like this:

Repeat{ (9)

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \tag{10}$$

$$\theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \qquad j \in \{1, 2 \dots n\} \tag{11}$$

} (12)

And the normal equation should look like this (vectorized):

$$\theta = (X^T X + \lambda \dot{L})^{-1} X^T y$$

$$\text{where } L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

3.6.2 Regularized Logistic Regression

The form of the regularization for the cost function is: $\frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$. We **won't apply regularization on the first parameter (bias)** for this case either. The gradient is the same as the linear.
When regularization is applied (to solve the overfitting issue), we gotta be careful because if we pick a very low regularization parameter (λ close to zero), we might still got the overfitting problem. On the other hand, if we pick a really high parameter, then there might be an underfitting problem.

4 Neural Networks

Why should I learn a new method if I already got *linear regression* and *logistic regression* (for classification). We saw that, in order to fit our model to the data, we can add more variables or increase the order of the polynomial but to achieve that we gotta do a lot of try/error. Wouldn't it be nice to have a method that self-learns the shape of the decision boundary based on all combinations of input data? or do the same for the shape of the polynomial regression ? That's where **neural networks** are for !!. Generally, we use neural networks to solve classification problems.

4.1 Representation and Concepts

The representation for neural networks is:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow h_\theta(x) \tag{13}$$

The first vector (x_j) is called **input layer** (x_i), it's length will be $n+1$ (n is the number of features, we add 1 because of the unit bias x_0).

The last item is the **output layer**($h_\theta(x)$) and can be a vector too, it represents the final answer which is the prediction (whether 0 or 1).

The layer(s) between *input* and *output* layers are called **hidden layer**. Each node in the layer is called **activation unit** that is a sigmoid function.

The relationship between layers is a matrix of weights ($\Theta^{(j)}$). Since it's a relationship it has to map all the combinations between the layers j and $j+1$, but we only include the unit bias of the layer j (add one to the number of the layer j), the bias unit is always 1 (because it will be directly multiplied by the activation unit). So it's length will be:

$$\Big[\#\text{ of units of layer } (j+1)\Big] \times \Big[1 + \#\text{ of units of layer } (j)\Big] \tag{14}$$

In summary, these are the representations:

s_j = number of units of layer j
$a_i^{(j)}$ = "activation" of unit i in the layer j
$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

Another way to present the neural network would be:

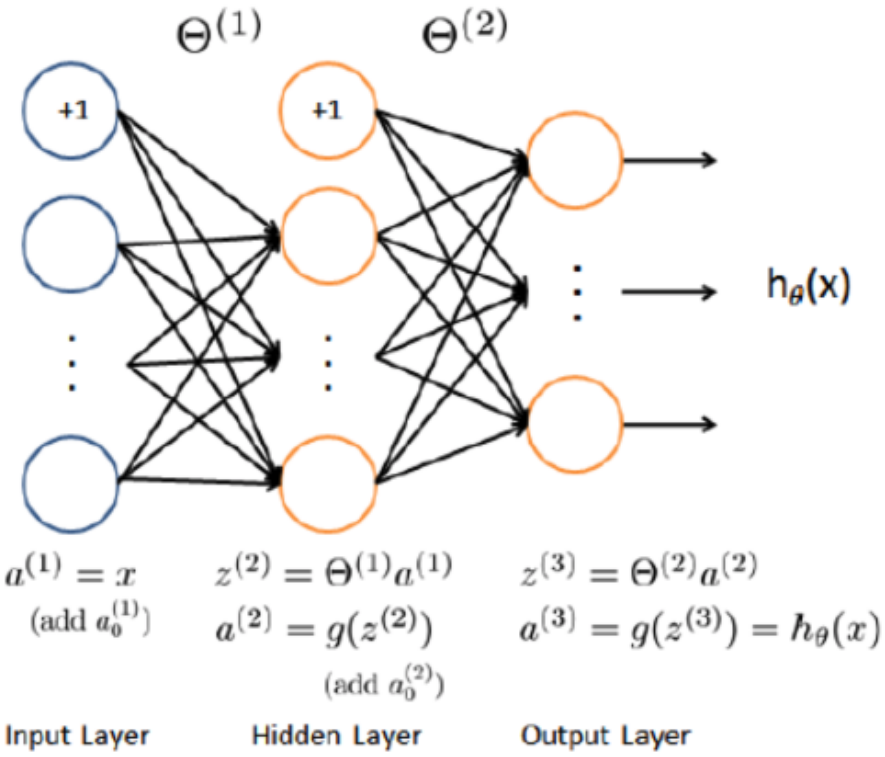

Figure 8: Neural Network

In the representation (15) the values for the activation units are:

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2)$$

And the value of the hypotheses function will be:

$$h_\Theta(x) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)}) = \begin{cases} 1 \\ 0 \end{cases}$$

This is saying that we compute our activation nodes by using a 2×3 (formula in 14) matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes. Each layer gets its own matrix of weights, $\Theta^{(j)}$.

We are going to introduce a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:

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

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

$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \ldots + \Theta_{k,2}^{(1)} x_2$$

4.2 Intuition - (what does a neural network actually learns)

A simple example of applying neural networks is by predicting $x_1 AND x_2$, which is the logical 'and' operator and is only true if both x_1 and x_2 are 1. The graph of our functions will look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \left[g(z^{(2)}) \right] \rightarrow h_\Theta(x) \tag{15}$$

(Remember that x0 is our bias variable and is always 1).
Let's set our first theta matrix as:

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

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

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

$$\begin{aligned} x_1 = 0 \text{ and } x_2 = 0 \text{ then } g(-30) \approx 0 \\ x_1 = 0 \text{ and } x_2 = 1 \text{ then } g(-10) \approx 0 \\ x_1 = 1 \text{ and } x_2 = 0 \text{ then } g(-10) \approx 0 \\ x_1 = 1 \text{ and } x_2 = 1 \text{ then } g(10) \approx 1 \end{aligned}$$

So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates.

For example, let's create a made-up gate: XNOR.

The $\Theta^{(1)}$ matrices for AND, NOR, and OR are:

$$\begin{aligned} &AND: \\ &\quad \Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix} \\ &NOR: \\ &\quad \Theta^{(1)} = \begin{bmatrix} 10 & -20 & -20 \end{bmatrix} \\ &OR: \\ &\quad \Theta^{(1)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix} \end{aligned}$$

We can combine these to get the XNOR logical operator (which gives 1 if x_1 and x_2 are both 0 or both 1).

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} a^{(3)} \end{bmatrix} \rightarrow h_\theta(x) \tag{16}$$

For the transition between the first and second layer, we'll use a $\Theta^{(1)}$ matrix that combines the values for AND and NOR:

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$$

For the transition between the second and third layer, we'll use a $\Theta^{(2)}$ matrix that uses the value for OR:

$$\Theta^{(1)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$$

Let's write out the values for all our nodes

$$\begin{aligned} a^{(2)} &= g(\Theta^{(1)} \cdot x) \\ a^{(3)} &= g(\Theta^{(2)} \cdot a^{(2)}) \\ h_\Theta(x) &= a^{(3)} \end{aligned}$$

And there we have the XNOR operator using a hidden layer with two nodes! The following summarizes the above algorithm:

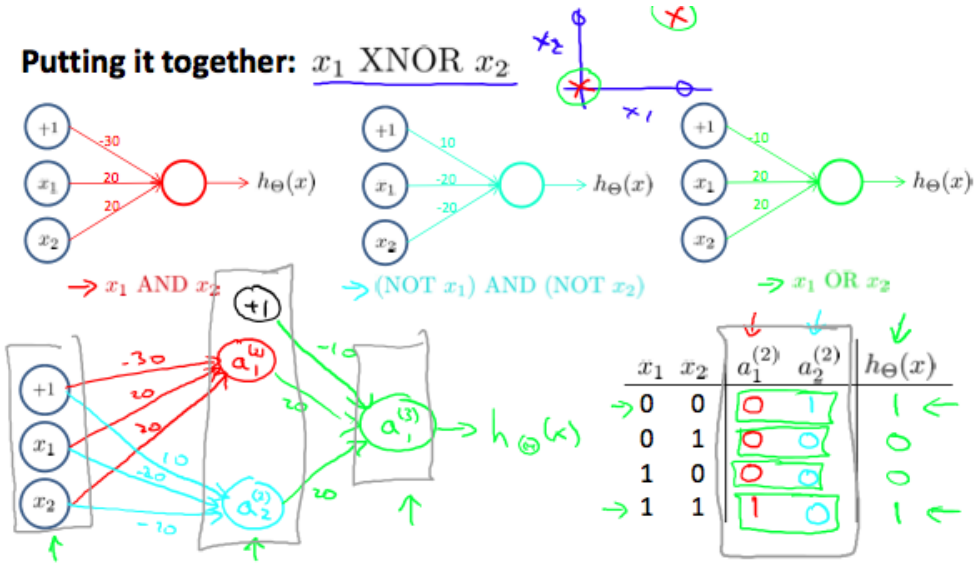

Figure 9: Neural Network x_1 XNOR x_2

4.3 Multiclass Classification

To classify data into multiple classes, the output of the neural network will be a vector that has the length of the number of classes (K). Something like this:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ \vdots \end{bmatrix} \rightarrow \ldots \rightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ h_\Theta(x)_3 \\ h_\Theta(x)_4 \end{bmatrix} \tag{17}$$

4.4 Cost function

To understand all of the notations in the formula, we define a few variables:

- 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} th output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression showed in formula (7):

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

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.

4.5 How to obtain $\Theta^{(l)}$