# Neural Network

*Steven Loaiza*

*January 29, 2019*

## Neural Networks Introduction

WHAT IS THIS! It sounds like this should be in a science class. Essentially, we are borrowing concept from science and applying it to Machine Learning (or Deep Learning concepts).

Disclaimer: By no means I am an expert in this topic, but I will explain the concepts the same way I was able to learn it, minus the countless hours of internet searches.

Back to the science. The concept builds on the set up a neuron cell. The dendrites recieve signals and then transmit the signals to the body of the cell. Then the body decides to send the signal to other neuron cells. The axon will trasmit the signal to other cells.

For our intent an purposes we are going to model a mathematical formula, that will take input values, transform the data via "signals", and give us a desired output.

Before we go deep into the definition and math lets start somewhere simple.

## Linear and Logistic Regression

Let us return to our Linear and Logistic Regression.

Recall that we can represent our dependent variable as:

$$Y = X'\beta$$

where $Y$ is an (nx1) dimentional, $X$ is (mxn) and $\beta$ is (mx1), where (m) is the number of features.

Suppose we have one feature $x_1$ and our graph of points look like this.

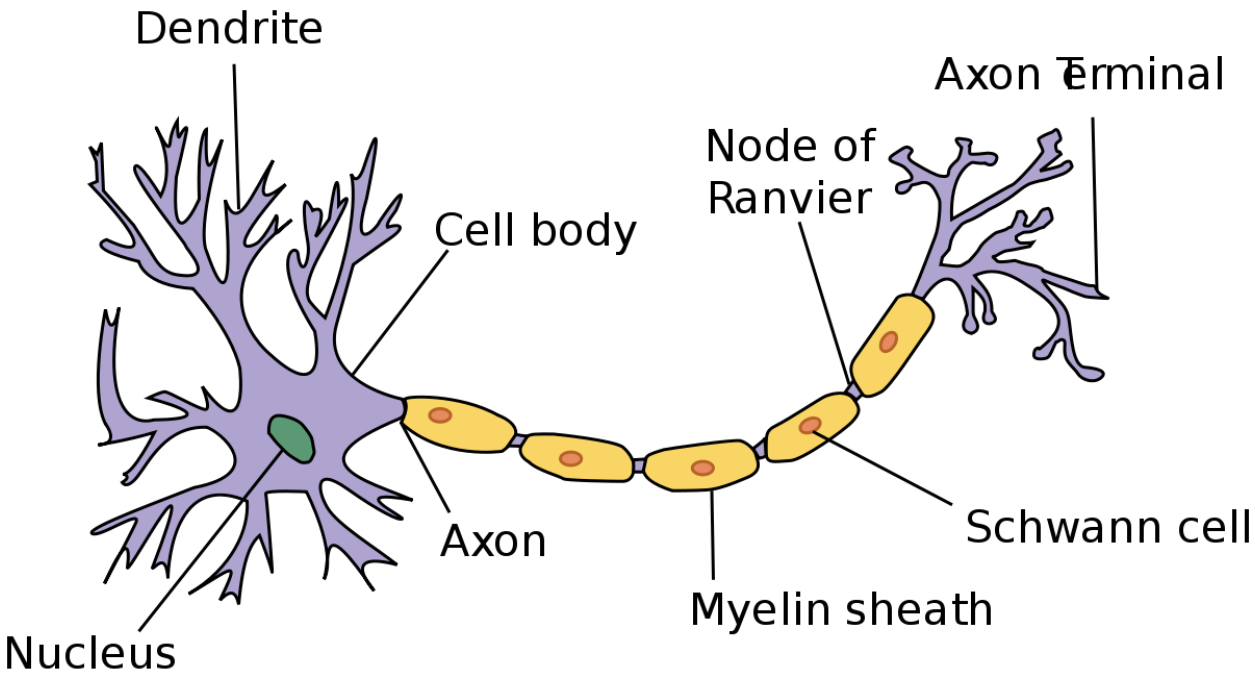
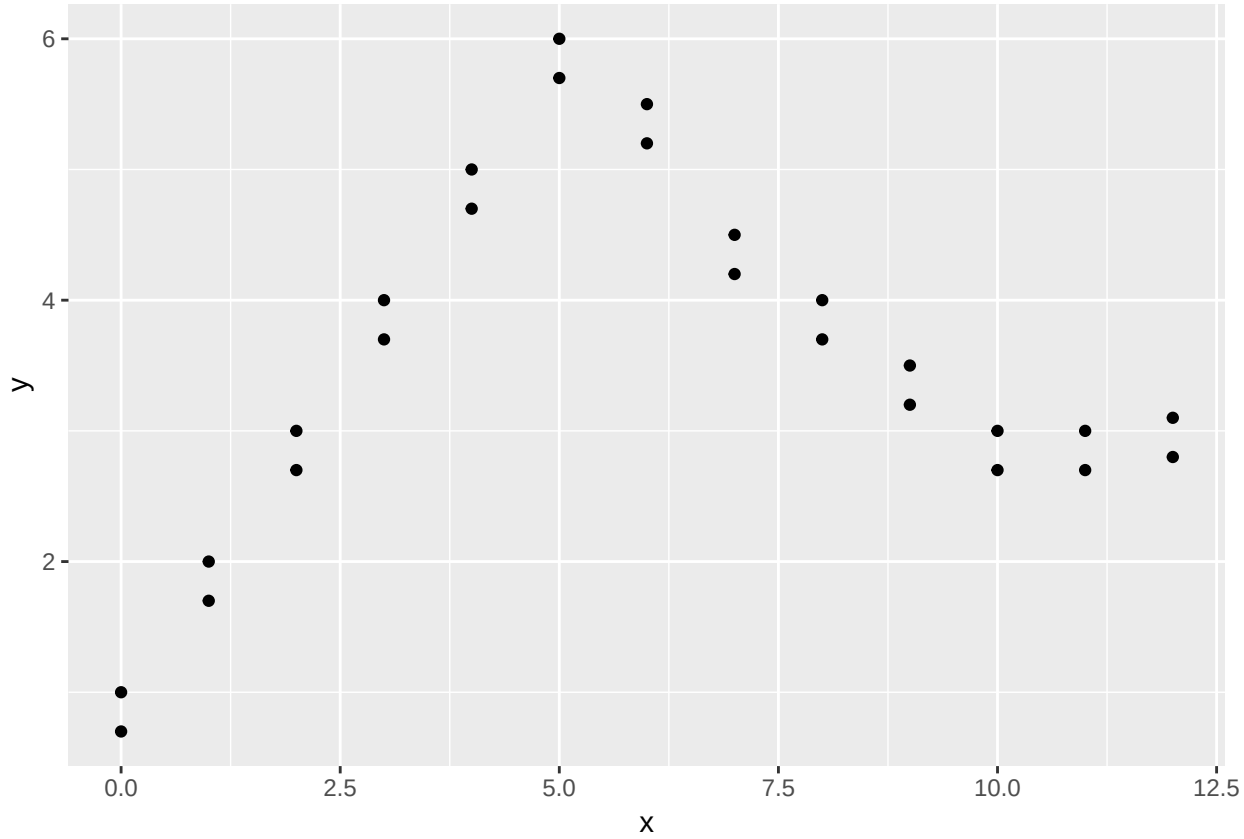


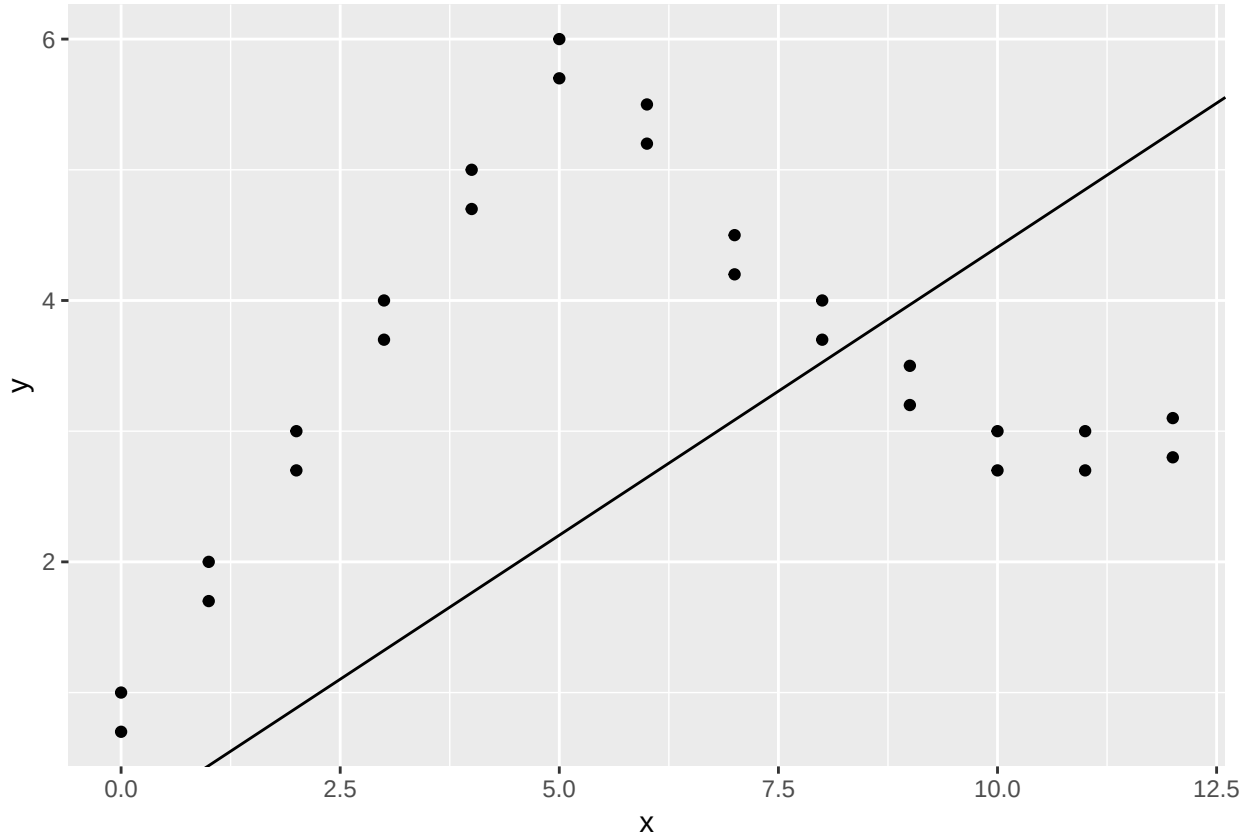

Figure 1: Original Source Wikimedia




Its obvious that a linear regression may not be the best fit for this data. Using linear regression we note that *beta* is 0.440, using the following code:

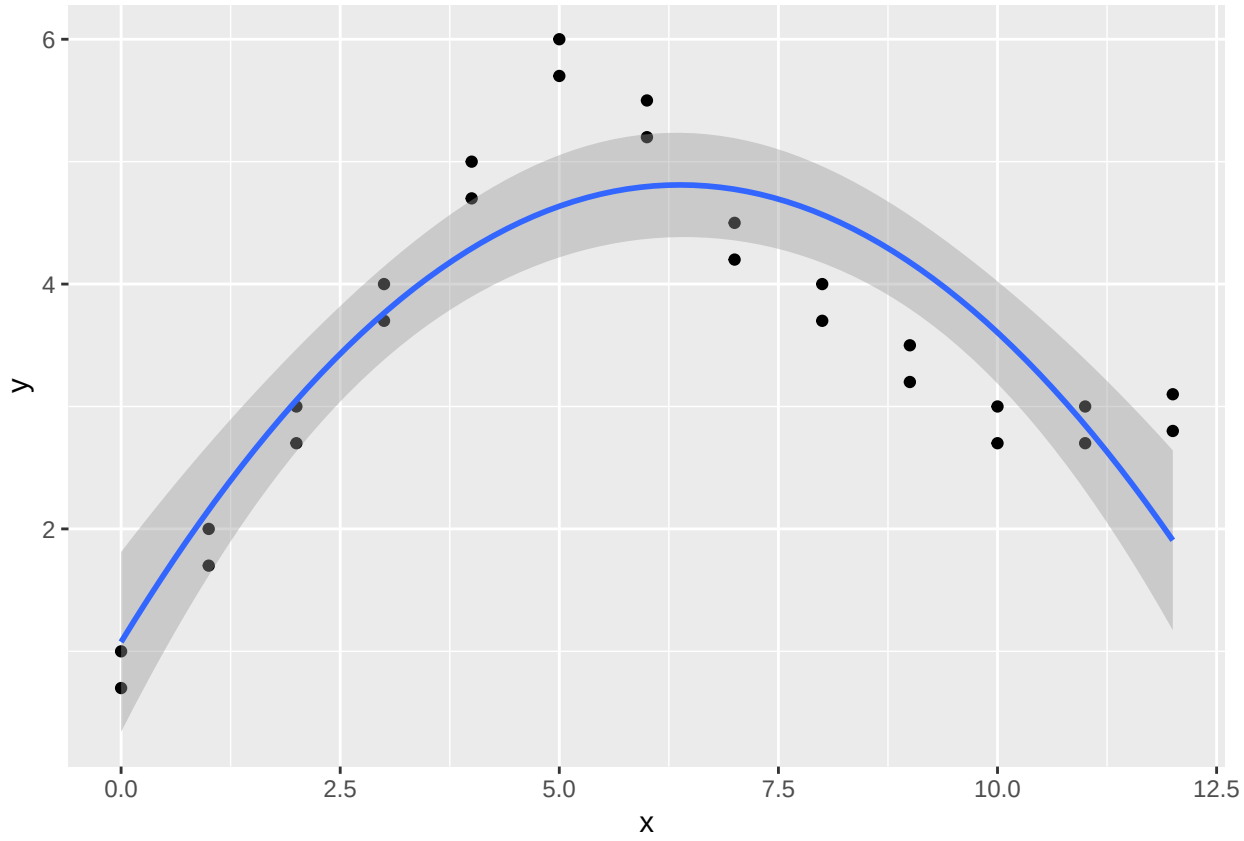
```
#we use negative one to omit the bias
reg<-lm(y~x-1)
```

Lets view the plot with our linear regression




This linear regression did not do so well fitting the data, therefore we may need to use a different feature, perhaps a polynomial.

```
fit<-lm(y~poly(x,2))
ggplot(,aes(x,y))+geom_point()+stat_smooth(method="lm",formula=y~poly(x,2))
```

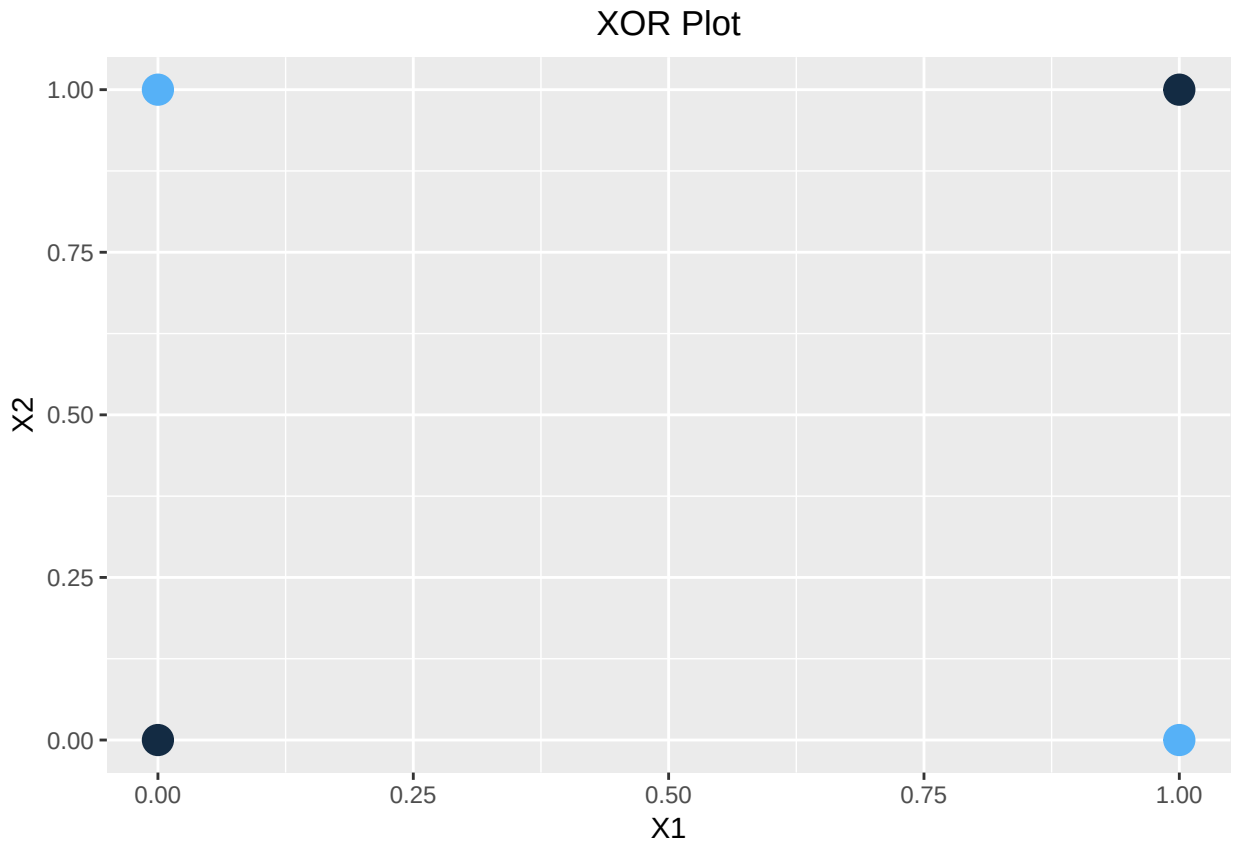



This looks like a much better representation.

## XOR

XOR is an exclusive "or"" problem. It essentially means that given two variables one of them has a different value.

```
##   X1 X2 Y
## 1  0  1 1
## 2  1  0 1
## 3  0  0 0
## 4  1  1 0
```




Recall in our ML problem, we want our algorithm to disginguish between two group, that being the Light blue and Dark Blue dots above.

We obviosuly can't use a linear regression, because a line will not be able to properly seperate out the two groups, we will need a non-linear regression.

We can use the logit regression which is non linear since $Y^* = g(z)$[1], where g is the sigmoid function and z is our features and weights, $g = \frac{1}{1+e^{-z}} \quad ; \quad z = X'\beta$.

Although a logistic regression has a non-linear function, its boundary region is still linear. Let say we have a threshold that $if \quad Y^* \quad > 0.5 \quad Y = 1 \quad otherwise \quad Y = 0$ that means that $\frac{1}{1+e^{-z}} > 0.5 \quad -> \quad e^{-z} > 1 \quad -> \quad -z > log(1) \quad -> \quad -X'\beta > 0$. Therefore our decision boundary is still linear in nature, even though our model is non-linear.

Let us add a nonlinear feature say $X_1 * X_2$ to our model.

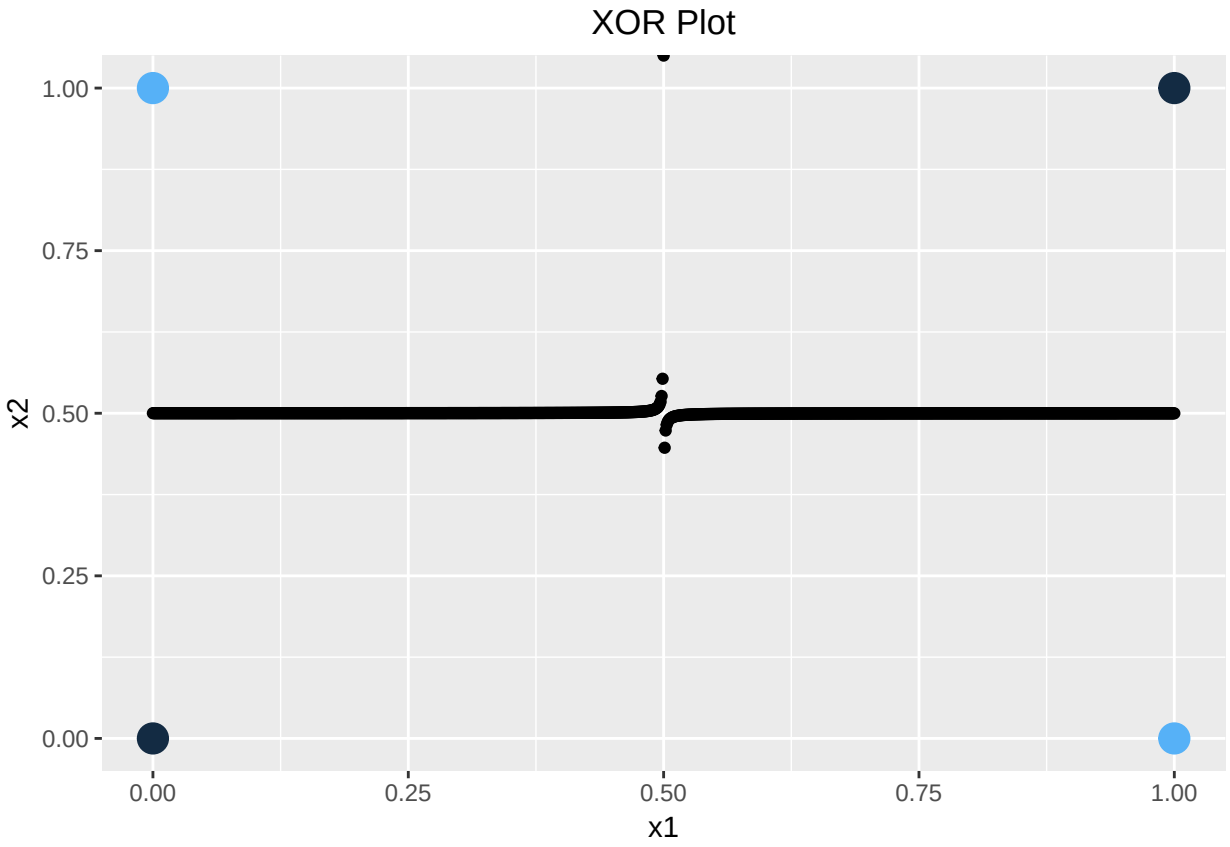
```
model<-glm(Y~q$V1+q$V2+q$V1*q$V2,family=binomial(link='logit'))
model
```

```
##
## Call:  glm(formula = Y ~ q$V1 + q$V2 + q$V1 * q$V2, family = binomial(link = "logit"))
##
## Coefficients:
## (Intercept)         q$V1         q$V2    q$V1:q$V2
##      -23.57        47.13        47.13       -94.26
##
```

[1]The asterisk above the Y, means that this variable is a latent variable, we dont observe this quantity, we only observe the output of Y being either a one or zero. Recall that a logit regression outputs values (0,1) and then using a threshold we determine if the output will be a one or a zero.

```
## Degrees of Freedom: 3 Total (i.e. Null);  0 Residual
## Null Deviance:       5.545
## Residual Deviance: 4.661e-10     AIC: 8
```

Now our decision boundary should be $-X'\beta > 0 \quad -> \quad -(-23.57 + 47.13X_1 + 47.13X_2 - 94.26X_1X_2) > 0$, if we solve for one of the variables we will get the formula for the decision boundary $(X_2 > -(\frac{\beta_0+\beta_1X_1}{\beta_2+\beta_3X_1}))$




It is difficult to tell that the decision bounday above , correctly classifies the XOR , due to it being non-differential at $x_1 = 0.5$. Below you can see the output of the prediction.

```
##   X1 X2 XOR Prediction
## 1  0  1           TRUE
## 2  1  0           TRUE
## 3  0  0          FALSE
## 4  1  1          FALSE
```

## Neural Networks

Neural Networks are learning methods that provide robust approches to approximating valued functions. Along with a concept called Back Propogation (which we will discuss in detail later) this method has been successful in handwriting ,voice and image recognition.

The basic concept of a neural network is this, we have a black box with some mathematical formulas inside of it. We also have a training data set, meaning that we have input values $(X)$ and we also have the output value $(Y)$. We can show the black box the inputs and the answers and it will return a function $f(X) \approx Y$. This function can now be used on unrealized $\hat{X}$ inputs and predict its output value $(Y)$.

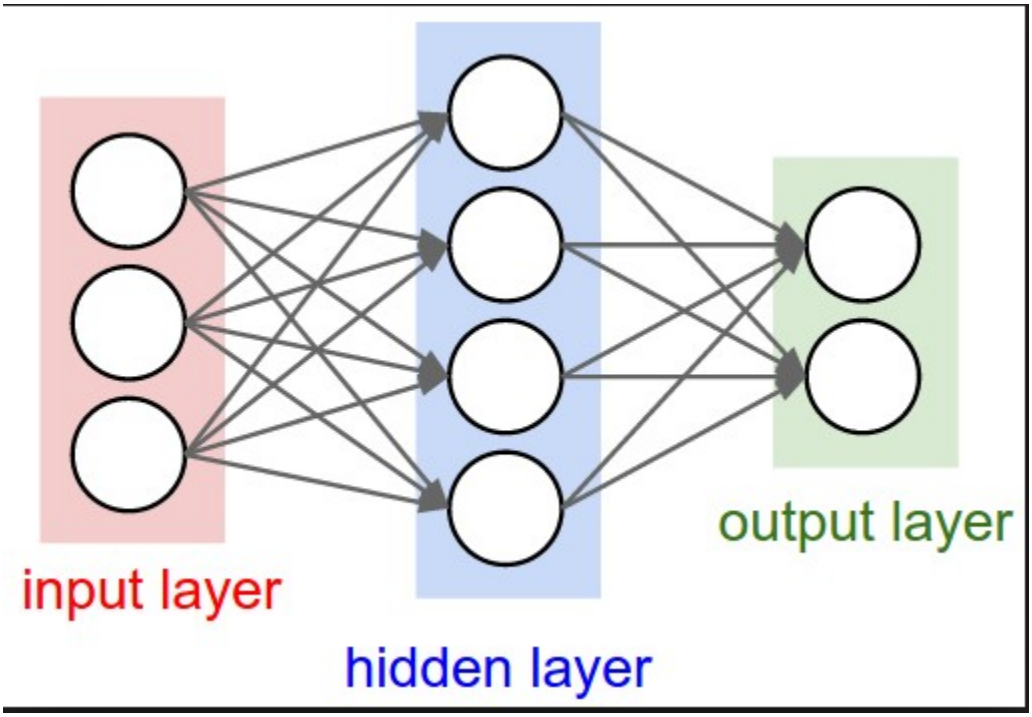



Figure 2: Network Layer

## Structure

There is a visual structure that can help us with the flow of a neural network. There is a input layer, hidden layer and output layer. The input layer will take the feature space for each observation. From the photo below we have 3 features, namely $x_{i1}, x_{i2}, x_{i3}$ where the subscript {i} is for the observation. The hidden layer, which we will go into greater detail later, is what most people call the black box, which perfroms operations unseen in the output space. The hidden layr will tranform our original feature space and create other complex features (linear, non-linear) to help get the best prediction of our output. Finally the output space is the probability the neural network places on the {k} output vector. For example the image below shows two items, in the output space thus the range of our data is 2. So our neural network will place a probabilistic value on which output value this particular observation {i} should be.

## Notation

We will start working on a neural network with 3 inputs features, 3 hidden layer features and 3 output values. Its important to note that the input and output size is determined by the data and problem you are trying to solve. The amount of hidden layers (yes there can be more than one) and hidden features is determined by YOU!

Notation:

$Y_k^d$ - Desire output for observation (d), for the (k)th output choice.

$X_i = a_i^1$ - (i)th feature in the training set, where 1 indicated the layer in the neural network.

$W_(ij)^q$ - Weight from the (i)th feature to the (j)th feature, from the (q)th layer.

$z_i^q = \sum_{i \epsilon I} w_{ij}^{q-1} a_i^{q-1}$ - summation of weights and features from the previous layer.

$\sigma(z_i^q) = a_i^q$ - (i)th feature in the (q)th layer.

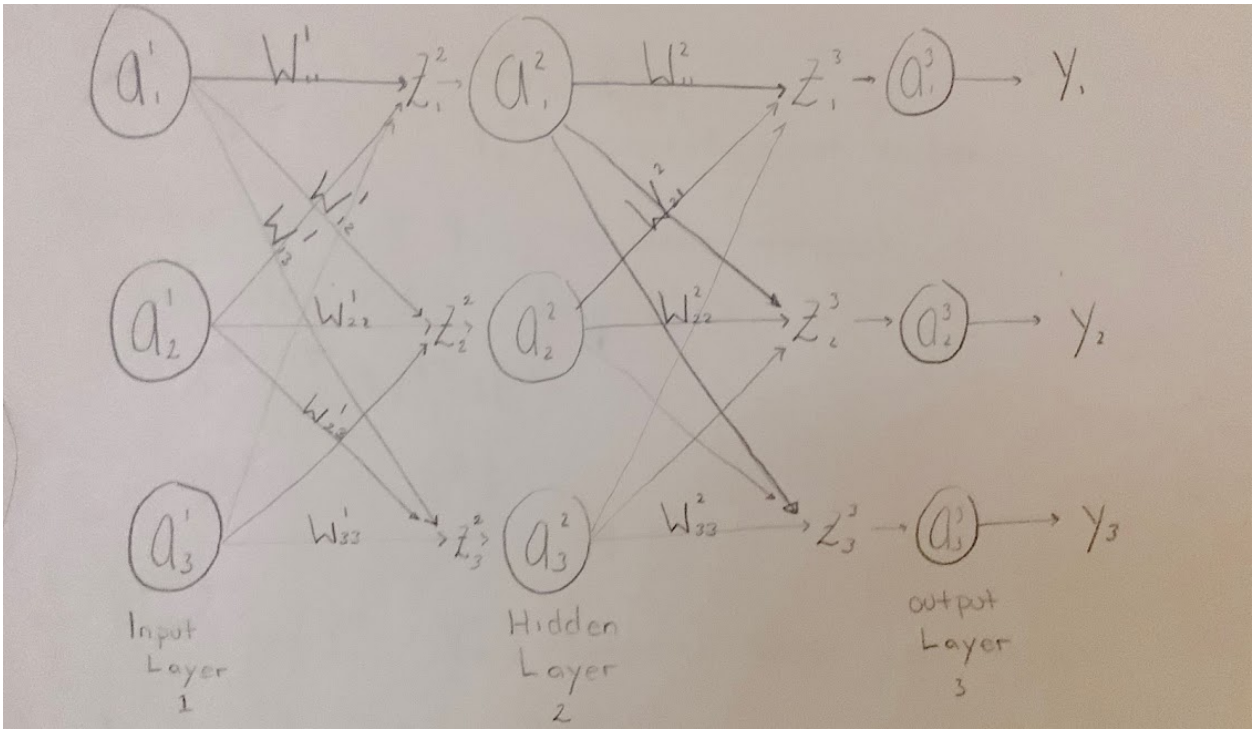



Figure 3: Network Layer 2

$h_\theta(.) = \sigma(.)$ - sigmoid function

L - number of layers

d - the index of the training example

M - the number of features in the training example

n - the number of observations

$\delta_i^q$ - Error in Layer (q) for feature (i). There is no error for q=1.

## Forward Propogation

We will use for loops in this introduction to Neural Networks. Using loops is computationally expensive, but for learning purposes it helps us grasp a better understanding of what is happening in the algorithm.

Let,

$X^d = < x_1^d \ldots \ x_m^d >$

$Y^d = < y_1^d \ldots \ y_m^d >$ Desired output vector

Initialize w_{ij}, $\forall$ q, i, j to some value(s)calculate $\forall$ q, i (recall a_i^1 =x_i) \

$z_i^q = \sum_{i \epsilon I} w_{ij}^{q-1} a_i^{q-1} \backslash$

$\sigma(z_i^q) = a_i^q$ \

obtain, $Y^o =< Y_1^o ... Y_k^o >$ output vector from our NN, where o = output.

## Cost Function

Since we made an initial guess as to the values of the weights, we know that our output value will not be close to the desired values.

Recall from calculus or economics , we can use a cost function to minimize it , with respect to our weights, by updating our weights.

There are many options to choose as a cost function. We will use:

$$C_d = -\sum_k [Y_k^d log(h_\theta()^d) + (1 - Y_k^d) log(1 - h_\theta()^d)]$$

Given our Neural Network we have 18 weights to Update.

## Error Function

We can easily compute the error in our output layer.

$$\delta_k^L = Y_k^o - Y_k^d$$

It is impossible to compute direct error functions for our hidden layer, because the desired intermediate values are unknown.

We can use a method called back propagation to propagate errors $\sigma$ back through all the neurons.

## Back Propagation

Intuitively, we are perforing a feed forward propagation starting from our output errors back to the feature space $X$.

$$\delta_k^L = Y_k^o - Y_k^d$$

and

$$\delta_i^q = \sum_{k \epsilon K} \frac{\partial C^d}{\partial Z_k^{q-1}} \frac{\partial Z_k^{q-1}}{\partial Z_i^q}$$

,$\forall i$ where $q \neq L$

Note: From the NN figure, for e.g, $a_1^2$ is an input for $a_1^3$,$a_2^3$,$a_3^3$. Therefore, $z_1^q$ is an input for $Z_i^{q+1} \forall i$, but only affects it through $a_1^q$.

$$\delta_i^q = \sum_{k \epsilon K} \frac{\partial C^d}{\partial Z_k^{q-1}} \frac{\partial a_i^q w_{ik}^q}{\partial Z_i^q}$$

It is easy to note the recursive nature:

$$\delta_i^q = \sigma'(z_i^q) \sum_{j \epsilon J} \delta_j^{q+1} w_{ij}^q$$

This is the weighted sum of our errors in the Last Layer. We can continue to perform this for each layer.

What we are most concern with is $\frac{\partial C^d}{\partial w_{ij}^q}$ $\forall$ q,i,j. We want to know this partial derivative to update our weights and minimize the cost function.

**Case 1:**

Where the output is the terminal node L. Therefore $W_{ij}^{L-1}$ only has effect through $Z_j^L$.

$$\frac{\partial C^d}{\partial W_{ij}^{L-1}} = \frac{\partial C^d}{\partial Z_j^L}\frac{\partial Z_j^L}{\partial W_{ij}^{L-1}}$$

We know:

$$\frac{\partial C^d}{\partial Z_j^L} = \delta_j^L$$

Derive:

$$\frac{\partial Z_j^L}{\partial W_{ij}^{L-1}} = \frac{\partial \sum_{i\epsilon I} W_{ij}^{L-1} a_i^{L-1}}{\partial W_{ij}^{L-1}} = a_i^{L-1}$$

Therefore:

$$\frac{\partial C^d}{\partial W_{ij}^{L-1}} = \delta_j^L a_i^{L-1} = (Y_j^o - Y_j^d) a_i^{L-1}$$

**Case 2:**

Where the output is an internal hidden layer (q). Going back to our Figure 3 we see that $W_{21}^1$ affects $C^d$ through $Z_1^2$ and $Z_1^2$ affects $Z_1^3, Z_2^3, Z_3^3$.

$$\frac{\partial C^d}{\partial W_{ij}^{q-1}} = \sum_{k\epsilon K} \frac{\partial C^d}{\partial Z_k^{q+1}}\frac{\partial Z_k^{q+1}}{\partial Z_j^q}\cdot\frac{\partial Z_j^q}{\partial W_{ij}^{q-1}}$$

where $q \neq L$

We have already derived the summation as $\delta_j^q$ from the back propogation section.

$$\frac{\partial Z_j^q}{\partial W_{ij}^{q-1}} = a_i^{q-1}$$

Putting it all together we obtain:

$$\frac{\partial C^d}{\partial W_{ij}^{q-1}} = a_i^{q-1}\cdot\sigma'(z_i^q)\sum_{j\epsilon J}\delta_j^{q+1} w_{ij}^q = a_i^{q-1}\cdot\delta_j^q$$

# Weight Update Rule

## Stochastic Gradient Descent ( Online, Single-setp learning)

Updates alter seeing only a single training example, or even batches of samples to reduce the variance.(This will not be proven here)

Our update rule is going to be

$$\Delta w_{ij}^q = \eta\cdot\delta_j^{q+1}\cdot a_i^q$$

as we go through each example in the training set.

$\eta$ is the learning rate of our update rule.

$$w_{ij}^q = w_{ij}^q + \Delta w_{ij}^q$$

## Batch Learning

Run through the entire traning set, keep track of our gradient.

$$\Delta_{ij}^q = \Delta_{ij}^q + \delta_j^{q+1} \cdot a_i^q$$

it can be shown that:

$$\frac{\partial J(\theta)}{\partial W_{ij}^q} = D_{ij}^q = \frac{1}{m}\Delta_{ij}^q$$

### Algorithm 1

Initialize weights, learning rate.

For $< X^d, y^d >$ in the training set.

- Feed forward
- for each k in the output set
  - $\{\delta_i^q = Y_k^o - Y_k^d\}$
- for each layer $q\epsilon[2, L)$ and feature $i$
  - $\delta_i^q = \sigma^{'}(z_i^q) \sum_{j\epsilon J} \delta_j^{q+1} \cdot W_{ij}^q$
- Use (SGD) update for all (i,j) pairs, $\forall q$
  - $w_{ij}^q = w_{ij}^q + \Delta w_{ij}^q$
  - $\Delta w_{ij}^q = \eta \cdot \delta_j^{q+1} \cdot a_i^q$
- Continue until (Add some stop criteria here) ### Algorithm 2

Follow all the steps for algorithm 1 and use batch learning instead. Once we have run through the entire training set, we will use an optimization method which will take as inputs our cost function and gradients $\frac{1}{m}\Delta_{ij}^q = D_{ij}^q$.
* R Code : Optim()

### Random Weight Initialization

Note that our cost function is not strictly convex, meaning it can have a global minimum as well as local minimum. Therefore, our choice of weights can cause our algorithm to converge to a local optimum.

- E = 10^{-5}
- $\theta = rand(0, 1) \cdot (2 \cdot E) - E$

where rand(0,1) is a random variable between 0 and 1

Additional reference He-et al. method.

### Bias + Update Rule

In our network we will have a bias in (L-1) layers with corresponding weights.

```
[Graphic w/ bias]
```

Notation $b_j^q$: $q^{th}$ layer connection to the $j^{th}$ feature in the q+1 layer. $b_j^q$ affects the cost function through $Z_j^{q+1}$.

Update:

$$\frac{\partial C^d}{\partial b_j^q} = \frac{\partial C^d}{\partial Z_j^{q+1}} \cdot \frac{\partial Z_j^{q+1}}{\partial b_j^q} = \delta_j^{q+1} \cdot \frac{\partial(\sum_{i \epsilon I} a_i^q \cdot W_{ij}^q) + b_i^q}{\partial b_j^q}$$

Note: $\frac{\partial(\sum_{i \epsilon I} a_i^q \cdot W_{ij}^q) + b_i^q}{\partial b_j^q} = 1$ where i=j.

Algorithm 1 Rule:

- $b_j^q = b_j^q + \eta \cdot \delta_j^{q+1}$

Algorithm 1 Rule:

- $\Delta_{bj}^q = \Delta_{bj}^q + \delta_j^{q+1}$
- Then take the weighted sum
  - $\frac{1}{m}\Delta_{bj}^q$

No need for a learning rate since we are using an optimization method to minimize the cost function here.

# Additional Information

## Adding Regularization

We add the following to the end of our cost function and focus on it.

$$\frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{S_l} \sum_{j=1}^{s_{l-1}} (w_{ij}^l)^2$$

* We do not regularize our bias unit, because we are concerned with over fitting the data points (x), the bias is merely a scaling parameter.

Recall we want to know $\frac{\partial C^d}{\partial w_{ij}^q}$ and $\frac{\partial C^d}{\partial b_j^q}$, since the bias is not regularize we can ignore it. The partial derivatives of a sum is the sum of partial derivatives. Therefore, we can take the $\frac{\partial C_{rhtside}^d}{\partial w_{ij}^q} = \frac{\partial \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{S_l} \sum_{j=1}^{s_{l-1}} (w_{ij}^l)^2}{\partial w_{ij}^q}$, then add it to our previous gradient.

$$\frac{\partial \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{S_l} \sum_{j=1}^{s_{l-1}} (w_{ij}^l)^2}{\partial w_{ij}^q} = \lambda W_{ij}^q$$

We are doing this for training examples (d), so we do not have to average it over (1/m).

Additionally, since this will happen over all tranining examples and always result in $\lambda W_{ij}^q$ we will have $m \cdot \lambda W_{ij}^q$. Thus our new $D_{ij}^q = \frac{1}{m}\Delta_{ij}^q + \frac{1}{m}(m \cdot \lambda W_{ij}^q)$.

## Gradient Checking

Let $J(\cdot)$ be out cost function, $\theta \epsilon R^n$, $\theta = [\theta_1 ... \theta_n]$.

$$grad \approx \frac{J(\theta + E) - J(\theta - E)}{2 + E}$$

REcall our apprximation of a tangent line using 2 points close by to get the slope.

If our back propagation is similar, then we can proceed.

- Make sure to turn off gradient checking due to computational cost.