CSE/ECE-471: Statistical Methods in AI **Spring 2020**

Neural Networks

Prepared by: Tom Sabu(2019201005), Dhawal Sirikonda(2019201089)

"*Life is like a box of Neural-Nets. You never know what you gonna get.* "

1 Perceptron

A neural network is a network or circuit of **neurons**, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes.Thus a neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems.The area of our concern is the Artificial Neural network(ANN). As a result, we ought to know the basic building blocks of ANN which is the Perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. So how do perceptrons work? A perceptron takes several binary inputs, x1,x2,..., and produces a single binary output:

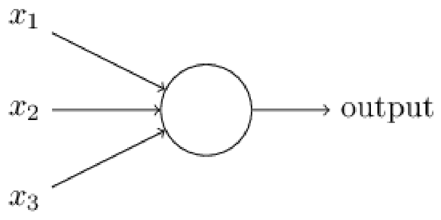


Figure 1: Perceptron

In the example shown the perceptron has three inputs, x1,x2,x3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced weights, w1,w2,..., real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

By varying the weights and the threshold, we can get different models of decision-making. Making a few changes to the above equation :

(i) Writing $\sum_j w_j x_j$ as a dot product, $w \cdot x \equiv \sum_j w_j x_j$
(ii) Moving the threshold to the other side of the inequality, and to replace it by what's known as the

perceptron's bias,
$b \equiv -\text{threshold}$

Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as AND, OR, and NAND.

2 Sigmoid Perceptron

Learning algorithms sound terrific. But how can we devise such algorithms for a neural network? Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. As we'll see in a moment, this property will make learning possible. Schematically, here's what we want (obviously this network is too simple to do handwriting recognition!):

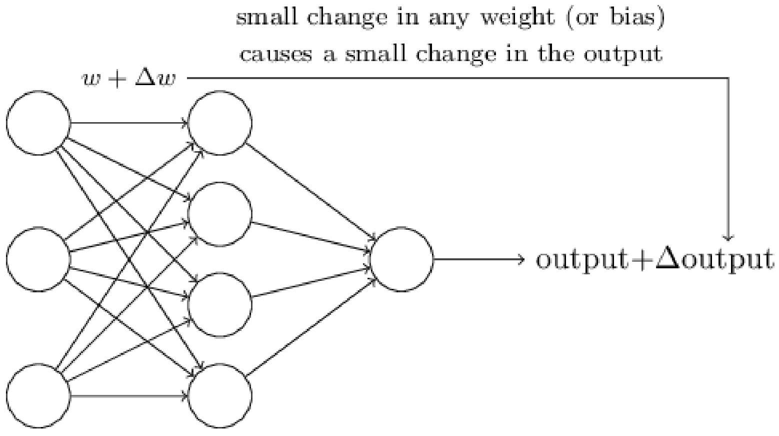
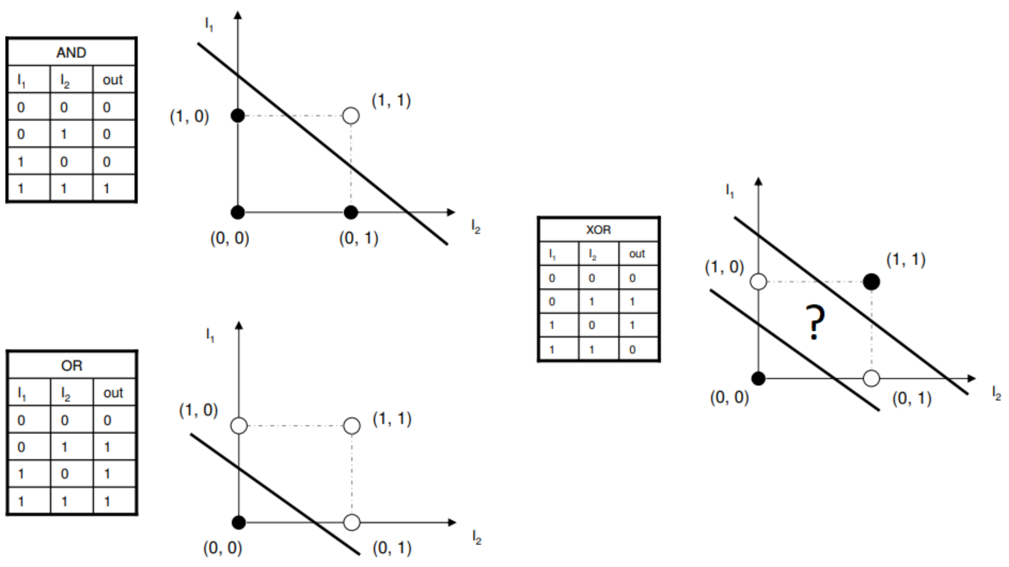

small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

Figure 2: Perceptron

We want our network to learn, which doesn't happen in case of perceptrons.We can overcome this problem by introducing a new type of artificial neuron called a sigmoid neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn. Just like a perceptron, the sigmoid neuron has inputs, x1,x2,.... But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1. So, for instance, 0.638... is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w1,w2,..., and an overall bias, b. But the output is not 0 or 1. Instead, it's $\sigma(w \cdot x + b)$, where is called the sigmoid function defined by :

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

This sigmoid function or any other function used is called an activation function. Use of more than one Hyperplanes will easily solve the issue faced in XOR which is not a linearly-separable data.



AND		
I1	I2	out
0	0	0
0	1	0
1	0	0
1	1	1

OR		
I1	I2	out
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
I1	I2	out
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3: XOR Hyperplanes

3 Network

A row of neurons is called a layer and one network can have multiple layers. The architecture of the neurons in the network is often called the network topology.

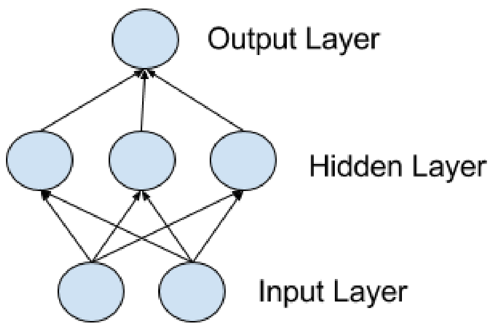


Figure 4: Network

Input or Visible Layers The bottom layer that takes input from your dataset is called the visible layer, because it is the exposed part of the network. Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset. These are not neurons as described above, but simply pass the input value though to the next layer.

Hidden Layers Layers after the input layer are called hidden layers because that are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value. Deep learning can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically, but may take seconds or minutes to train using modern techniques and hardware.

Output Layer The final hidden layer is called the output layer and it is responsible for outputting a value or vector of values that correspond to the format required for the problem i.e. those which are not lineraly separable.The **MultiLayer Perceptron (MLPs)** breaks this restriction and classifies datasets which are not linearly separable.

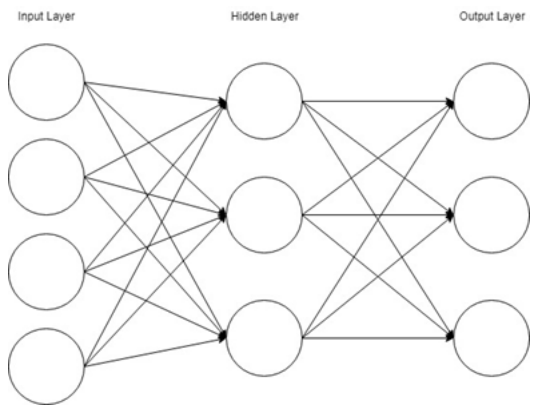


Figure 5: MLP

4 MultiLayer Perceptrons

The perceptron is very useful for classifying data sets that are linearly separable. They encounter serious limitations with data sets that do not conform to this pattern. The Perceptron consists of an input layer and an output layer which are fully connected. MLPs have the same input and output layers but may have multiple hidden layers in between the aforementioned layers, as seen below.

5 Perceptron Learning Algorithm

The algorithm for the MLP is as follows:

1. Just as with the perceptron, the inputs are pushed forward through the MLP by taking the dot product of the input with the weights that exist between the input layer and the hidden layer (WH). This dot product yields a value at the hidden layer. We do not push this value forward as we would with a perceptron though.

2. MLPs utilize activation functions at each of their calculated layers. There are many activation functions to discuss: rectified linear units (ReLU), sigmoid function, tanh. Push the calculated output at the current layer through any of these activation functions.

3. Once the calculated output at the hidden layer has been pushed through the activation function, push it to the next layer in the MLP by taking the dot product with the corresponding weights.

4. Repeat steps two and three until the output layer is reached.

5. At the output layer, the calculations will either be used for a backpropagation algorithm that corresponds to the activation function that was selected for the MLP (in the case of training) or a decision will be made based on the output (in the case of testing).

For our example, let the inputs be x_1 and x_2 the outputs be y_5 and y_6, the weights among the $node_i$ and $node_j$ be w_{ij} leading to $w_{13}, w_{14}, w_{23}, w_{24}, w_{35}, w_{36}, w_{45}, w_{46}, f_j$ being the activation activation at the particular node.

Thus we have, $y_5 = f_5(w_{35}.y_3 + w_{45}.y_4)$ which can be written in terms of original inputs x_1 *and* x_2.

$$y_5 = f_5(w_{35}.f_3(w_{13}.y_1 + w_{23}.y_2) + w_{45}.f_4(w_{14}.y_1 + w_{24}.y_2))$$

$$y_5 = f_5(w_{35}.f_3(w_{13}.f_1(x_1) + w_{23}.f_2(x_2)) + w_{45}.f_4(w_{14}.f_1(x_1) + w_{24}.f_2(x_2)))$$

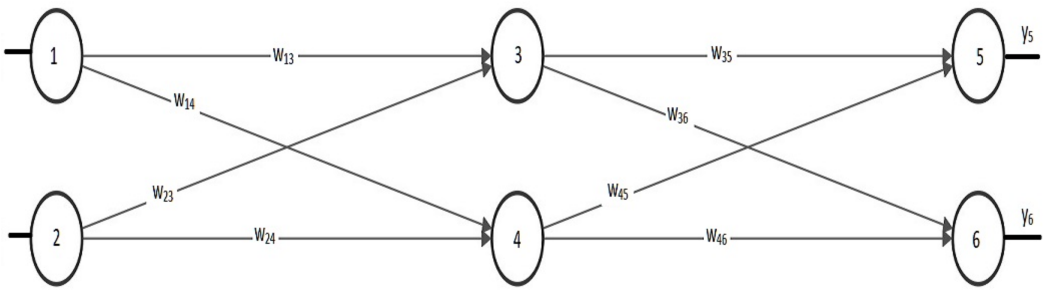


Figure 6: MLP

6 Linear Regression Vs Neural Networks

From the above result it looks more like Linear regression which is a linear combination of inputs coupled with some weights.
How is Neural Network different from Linear regression? In linear regression the output is just a linear combination of inputs. If we see the Neural network in a crude scenario, even the Neural Network tries to find a linear combination of inputs. But, We actually use a non-linearity which is not present in Linear regression.

The following is the Linear regression

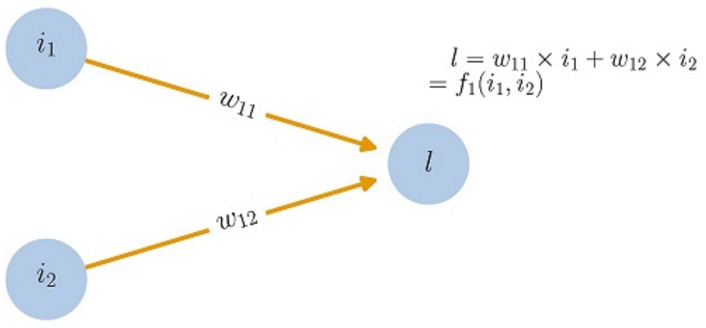


Figure 7: Linear Regression

On that output if we add a non-linearity a, we obtain the following

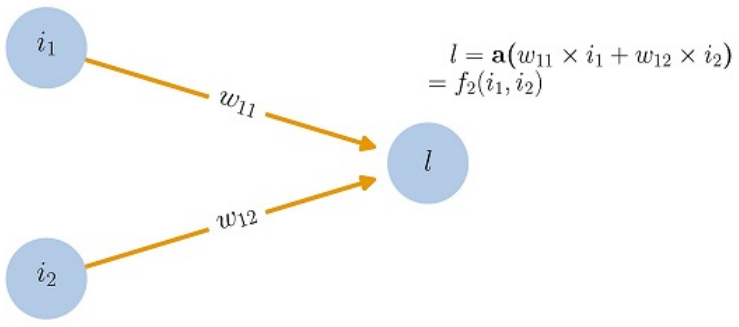


Figure 8: Non Linearity a

The a can be a logistic function which makes turns the linear regression into a logistic regression.

This non-linearity is applied along all the neuron in the network whereby the output need not be a linear combination of inputs.

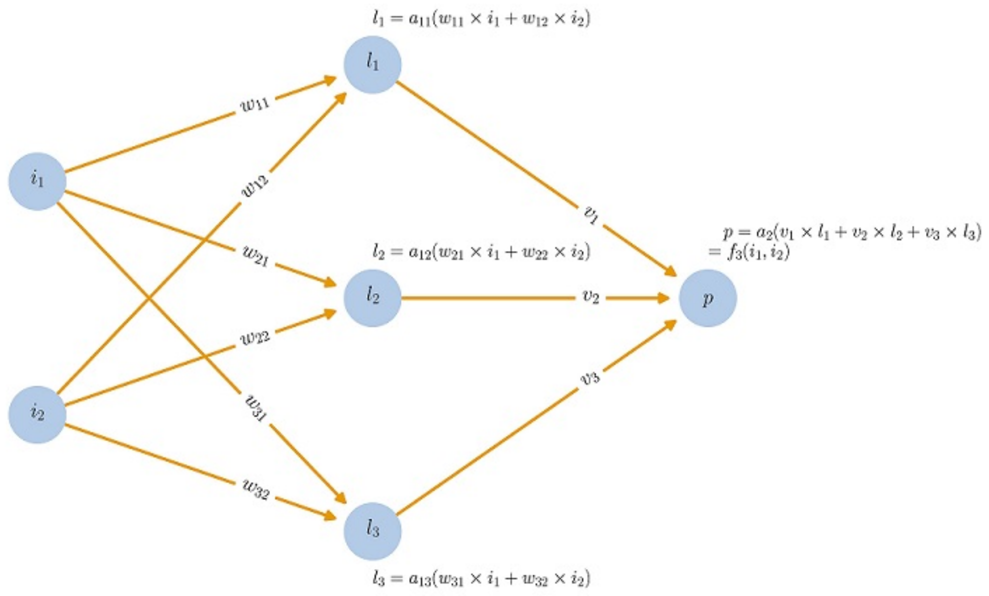

$l_1 = a_{11}(w_{11} \times i_1 + w_{12} \times i_2)$
$l_2 = a_{12}(w_{21} \times i_1 + w_{22} \times i_2)$
$l_3 = a_{13}(w_{31} \times i_1 + w_{32} \times i_2)$
$p = a_2(v_1 \times l_1 + v_2 \times l_2 + v_3 \times l_3)$
$= f_3(i_1, i_2)$
i_1
i_2
l_1
l_2
l_3
p
w_{11}
w_{12}
w_{21}
w_{22}
w_{31}
w_{32}
v_1
v_2
v_3

Figure 9: Neural Network - (Non Linearity) a

In usual scenarios the same activation function a is same through-out the network.

7 Loss Function

Lets us start with an easy case: Calculating a loss function for the regression task.

7.1 Regression

We often settle to use a mean square error as the loss function of the Regression task. In the following setup the Neural Network has a single neuron in the final layer and value it returns is a continuous numerical value.

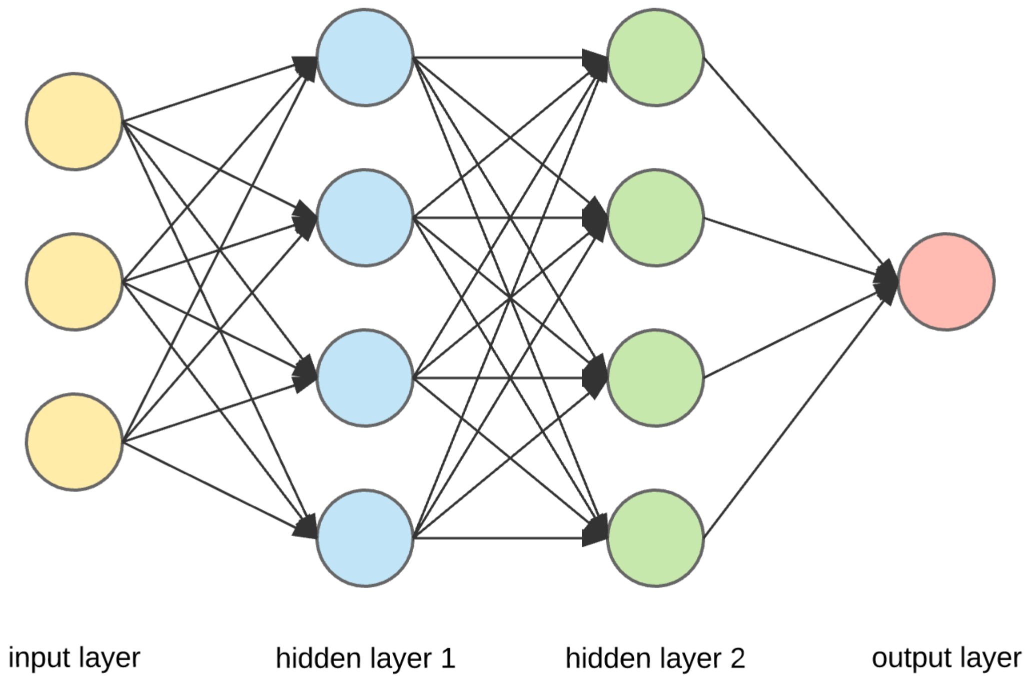


Figure 10: Neural Network Setup for Regression

The last neuron generally uses a Linear activation function or a ReLU activation function whereby we obtain a continuous value at the output.

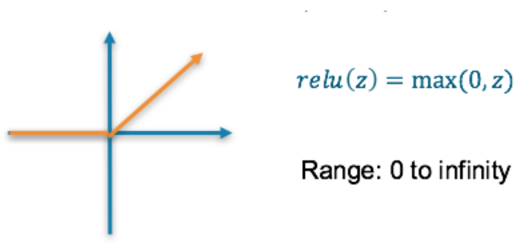


Figure 11: *Relu*

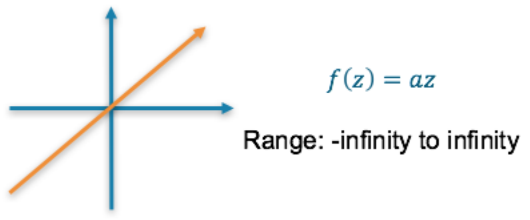


Figure 12: *Linear*

The obtained value at output neuron is compared with actual value. The deviation of predicted value from the actual is quoted as the Loss*(J)* and the goal is to reduce this Loss*(J)*. In most cases we stick to the Mean Squared Error (*MSE*) which is

$$MSE = \frac{1}{N} \sum_{i=0}^{N} (y_i^{predicted} - y_i^{actual})^2$$

There other loss functions which are also used along side *MSE*, are Mean Absolute Error (*MAE*) and Mean Squared Logarithmic Error (*MSLE*). Given by

$$MAE = \frac{1}{N} \sum_{i=0}^{N} |(y_i^{predicted} - y_i^{actual}|$$

$$MSLE = \frac{1}{N} \sum_{i=0}^{N} (\log(y_i^{predicted} + 1) - \log(y_i^{actual} + 1))^2)$$

7.2 Classification

In Neural-Networks Classification is Regression. Neural Networks learn the distribution over classes in the feature space. Without Regression we can not perform classification. Neural networks approximate a function which outputs the sufficient statistics of some probability mass/density function. It's our interpretation of the output of a neural network that gives rise to classification as being distinct from regression.

Let us dive deep into what it means. Say we have a multi-class classifier. In such classification, the assumption we typically make about the target data is that they are distributed according to categorical distribution. In such cases we use a Softmax function that generalizes the logistic function.

What is Softmax? The Softmax activation function turns numbers (ie logits) into probabilities that sum up to one. The function outputs a vector that represents the probability distribution of a list of possible outcomes. Softmax has two components:

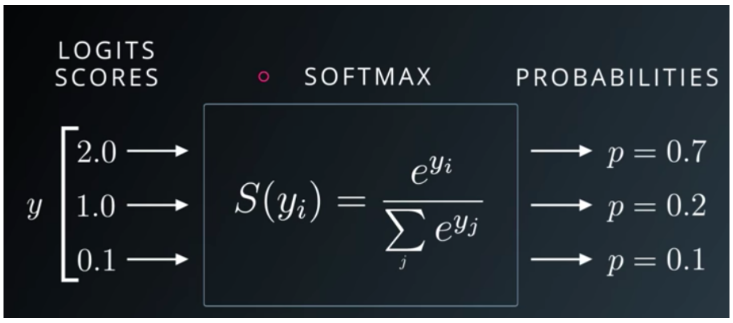


Figure 13: Softmax Flow

- Transform the components to e^x. This allows the neural network to work with logarithmic probabilities, instead of ordinary probabilities. This turns the common operation of multiplying probabilities into addition, which is far more natural for the linear algebra based structure of neural networks.
- Normalize their sum to 1, since that's the total probability we need.

Why do we need Softmax? Why can we just use a logits or probabilities(i.e; divide logits/probabilities by sum of logits/probabilites)?

Observe the Logit function $logit(x) = \log(\frac{x}{1-x})$ and its graph given below.

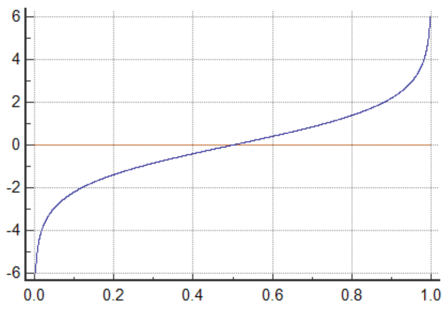


Figure 14: Logit Function

We can clearly see that $Range(logit(x)) = [-\infty, \infty]$ and adding negative values does not give a correct normalization. But exponent of a logit lies in the range of $[0, \infty]$. Also the gap between two logits increases when transformed using a exponential function, giving scope for a better classification (which is fine tuned by the minimization of loss function). e^{100} being very large value, while e^{-100} being very small positive value, which can be clearly seen in the below diagram.

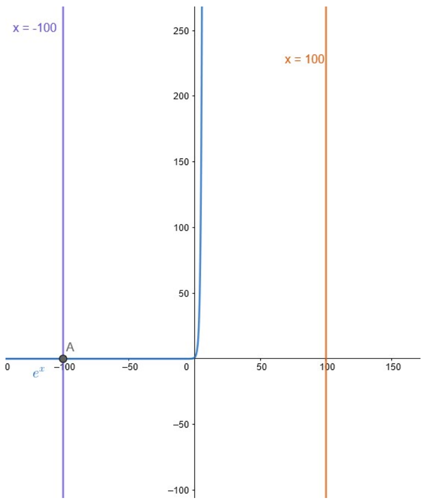


Figure 15: e^x graph

So we first apply exponentiation on logits for a k class case i.e;

$$Logits(L) = \begin{bmatrix} y_1 & y_2 & y_3 & . & . & . & y_k \end{bmatrix}$$
$$L = \begin{bmatrix} y_1 & y_2 & y_3 & . & . & . & y_k \end{bmatrix} \xrightarrow{exponentiation} e^L = \begin{bmatrix} e^{y_1} & e^{y_2} & e^{y_3} & . & . & . & e^{y_k} \end{bmatrix}$$

Now obtaining probabilities from exponential vector is the prediction of the network. Which is given by

$$Y^{predict} = \begin{bmatrix} \frac{e^{y1}}{\sum_{i=1}^{k} e^{y_i}} & \frac{e^{y2}}{\sum_{i=1}^{k} e^{y_i}} & \frac{e^{y3}}{\sum_{i=1}^{k} e^{y_i}} & . & . & . & \frac{e^{y_k}}{\sum_{i=1}^{k} e^{y_i}} \end{bmatrix}$$

Where each i^{th} element/value of the $Y^{predicted}$ is the probability that given input sample belongs to the i^{th} class of potential classes $\begin{bmatrix} C_1 & C_2 & C_3 & . & . & . & C_k \end{bmatrix}$. And the output label is the C_j class where

$$y_j^{predicted} = max(y_i^{predicted}), \forall i \in [1, k]$$

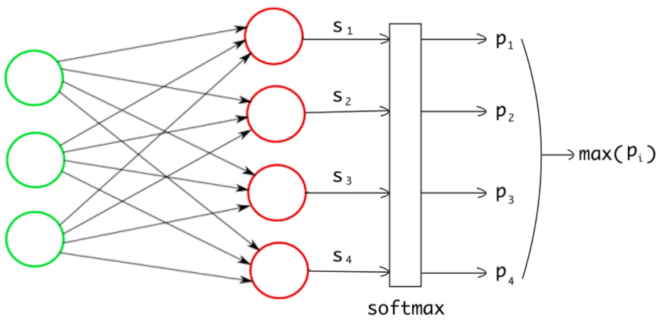


Figure 16: Output for Classification in case of Single Label

What about Loss Function? For simplicity assume that we are doing a classification for 5 class problem. Say we need to to classify whether the given input sample is one of [dog, fox, horse, eagle, squirrel].

Figure 17: Multi-Class Classification Example

Softmax layer will give us the probability the given sample belongs to [dog, fox, horse, eagle, squirrel] in their respective positions in the softmax output vector. That is

$$\begin{aligned}
&Softmax_{output} = \begin{bmatrix}0.4 & 0.3 & 0.05 & 0.05 & 0.2\end{bmatrix} \text{ which states}\\
&\text{Probability that input being } dog = P_{dog}(input) = 0.4\\
&\text{Probability that input being } fox = P_{fox}(input) = 0.3\\
&\text{Probability that input being } horse = P_{horse}(input) = 0.05\\
&\text{Probability that input being } eagle = P_{eagle}(input) = 0.05\\
&\text{Probability that input being } squirrel = P_{squirrel}(input) = 0.2\\
&\text{which sum to 1}
\end{aligned}$$

From these probability value we need to obtain a loss function, as stated earlier Classification in Neural Nets is Regression. For this we introduce Cross Entropy. Before that we need to look into log loss for which we apply a log on the probabilities obtained from softmax layer, we will discuss why are doing this?. For the above example two probabilities collided (that is for input being horse and eagle, both turning out to be 0.05), when applying log on the softmax we obtain the following

$$\begin{aligned}
&\text{Probabilities(P)= Q} = \begin{bmatrix}P_{y_1} & P_{y_2} & . & . & . & P_{y_k}\end{bmatrix} \xrightarrow{log} Log(Q) = \begin{bmatrix}log(P_{y_1}) & log(P_{y_2}) & . & . & . & log(P_{y_k})\end{bmatrix}\\
&\text{Here } Q_1 \text{ being}\\
&Q_1 = \begin{bmatrix}0.4 & 0.3 & 0.05 & 0.05 & 0.2\end{bmatrix} \implies Log(Q_1) = \begin{bmatrix}-0.916 & -1.203 & -2.995 & -2.995 & -1.609\end{bmatrix}
\end{aligned}$$

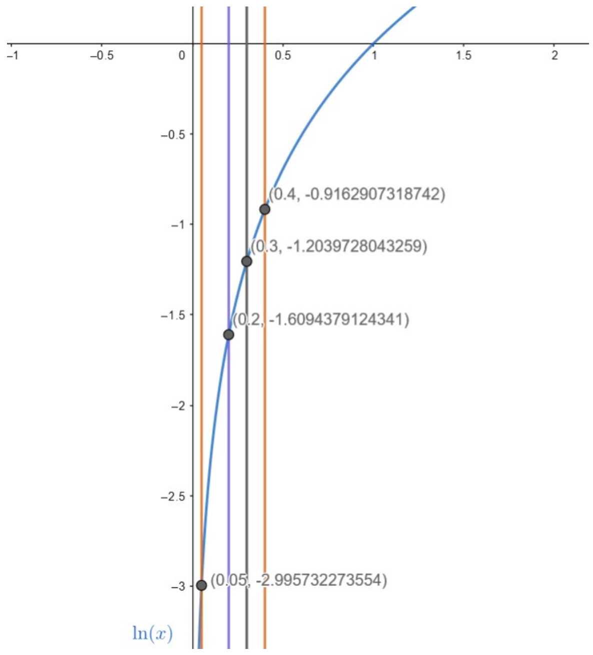


Figure 18: Positive log of prediction probabilities

We can clearly observe that if the probability approaches near to zero the log value of the concerned probability approaches very large negative value that is $-\infty$. But to see it more easily we take negative log whereby miss-classification of a particular input sample incurs a high penalty (as it's $-log(x)$ value is large positive number). That is if input image is a dog and our network gave a probability of *0.05* that input image is is a dog. Then $-log(0.05)$ is a very large positive value stating loss is high. But this alone is not sufficient for obtaining a loss function to train a Neural Network. We also need the ground truth.

$$\text{Probabilities(P)= Q} = \begin{bmatrix} P_{y_1} & P_{y_2} & . & . & . & P_{y_k} \end{bmatrix} \xrightarrow{log} -Log(Q) = \begin{bmatrix} -log(P_{y_1}) & -log(P_{y_2}) & . & . & . & -log(P_{y_k}) \end{bmatrix}$$
Here Q_1 being
$$Q_1 = \begin{bmatrix} 0.4 & 0.3 & 0.05 & 0.05 & 0.2 \end{bmatrix} \implies -Log(Q_1) = \begin{bmatrix} 0.916 & 1.203 & 2.995 & 2.995 & 1.609 \end{bmatrix}$$

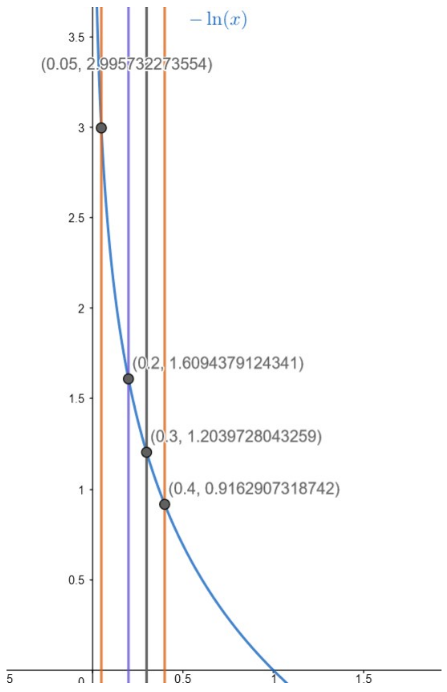


Figure 19: Negative log of prediction probabilities

Categorical Cross Entropy*(H)* As discussed earlier in case of input sample being a dog, the ground reality should be

Probability that input being $dog = P_{dog}(input) = 1$
Probability that input being $fox = P_{fox}(input) = 0$
Probability that input being $horse = P_{horse}(input) = 0$
Probability that input being $eagle = P_{eagle}(input) = 0$
Probability that input being $squirrel = P_{squirrel}(input) = 0$
which sum to 1
Ground Truth $= P = Y^{actual} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}$
Predicted $= Q_1 = Y^{predicted} = \begin{bmatrix} 0.4 & 0.3 & 0.05 & 0.05 & 0.2 \end{bmatrix}$

Y^{actual} is nothing but one-hot encode of class labels. Our Goal here is to make the $Y^{predicted}$ to be equivalent to Y^{actual}. That is increase the correct class probability and reduce the wrong class probability. Lets us look at the function which deals with this we will look into the idea why it works later.

$$H(P, Q) = E_{x\sim P}[-log(Q(x))]$$
$$H(P, Q) = \sum_i P(i)(-log(Q(i))) = -\sum_i P(i)log(Q(i))$$

Which measures how similar is Q w.r.t P (other way around need not be necessarily equal). Lets us see how and why it works.

Ground Truth $= P = Y^{actual} = \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}$
Predicted $= Q_1 = Y^{predicted} = \begin{bmatrix}0.4 & 0.3 & 0.05 & 0.05 & 0.2\end{bmatrix}$
$H(P, Q_1) = -(1.log(0.4) + 0.log(0.3) + 0.log(0.05) + 0.log(0.05) + 0.log(0.2)) \approx 0.916$
which states the the dissimilarity between Q and P is 0.916 and our job is to reduce it.

That is if the $P_{dog}(input)$ is high (i.e, is near to 1). Cross Entropy goes to zero.

$P = Y^{actual} = \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}$
$Q_{hypothetical} = Y^{predicted} = \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}$
$H(P, Q_{hypothetical}) = -(1.log(1) + 0.log(0) + 0.log(0) + 0.log(0) + 0.log(0)) = 0$
that is, as the Q approaches P [Q becomes exactly similar to P] cross entropy falls to zero.

It can also be seen that a dog being classified as eagle won't directly effect loss but sum of probabilities should be 1. So, if $P_{eagle}(input)$ is near to 1, it implies that $P_{dog}(input)$ is supposed to be near to zero making the $P_{dog}(input)$ a very high value which when multiplied with ground truth $\begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}$ gives a large positive value which is basically the loss.

As the number of epoch increase the loss decreases and the $-log(P_{dog}(input))$ approaches 1 say it approached a value of 0.98 and rest of the values be $\begin{bmatrix}0.98 & 0.01 & 0 & 0 & 0.01\end{bmatrix}$. Then the Cross Entropy of $P\ and\ Q$ is given by

$P = Y^{actual} = \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}$
$Q_{k^{th}iteration} = Y^{predicted} = \begin{bmatrix}0.98 & 0.01 & 0 & 0 & 0.01\end{bmatrix}$
$H(P, Q_{k^{th}iteration}) = -(1.log(0.98) + 0.log(0.01) + 0.log(0) + 0.log(0) + 0.log(0.01)) \approx 0.02$
which shows the Cross Entropy is near to zero

There are also cases where we try to do a multi-label classification(i.e; for a input sample more than one class label are possible). In this scenarios the Sofmax can not be used anymore we move towards sigmoid function which is given by $Sigmoid(x) = \frac{1}{1+e^x}$ and even in this case we use one-hot encoded ground truths and output layer looks similar to the following.

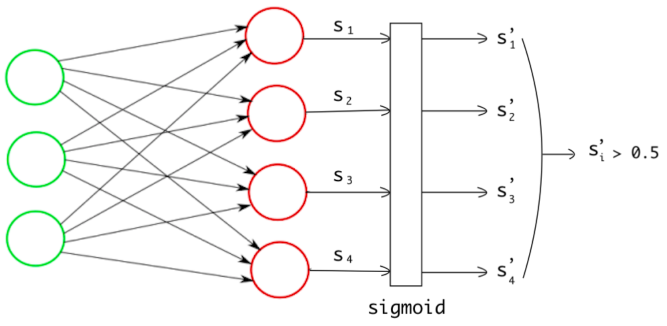


Figure 20: Output for Classification in case of Multi-label

8 How does it Train and update weights?

The training step is like Linear Regression, basically involving learning of weights. In which the keys terms involved are *Learning Rate* (α), a weight matrix (W) and a bias (b this can be adjusted in weight matrix *W*). Just like usual Linear Regression even here we tend to apply Gradient decent on the Loss Function (for simplicity consider *MSE*).

Here one point that needs to be kept in mind while fetching Gradient w.r.t a particular weight($node_i$ and $node_j$). $\frac{\delta J}{\delta w_{i,j}}$, Loss function is basically not just a linear combination of weight and inputs but there a non-linear function which activates the input at every neuron in the network. So simple calculation of gradient might be a tedious task. But, we could use some thing called a chain rule which is as follows

$$\frac{\delta y}{\delta x} = \frac{\delta y}{\delta z} \cdot \frac{\delta z}{\delta p} \cdot \frac{\delta p}{\delta x}$$

Lets use this on a simple Neural Network. Let the inputs of the $neuron_1$ and $neuron_2$ be x_1 and x_2 and non-linearity function used is $\phi(x)$

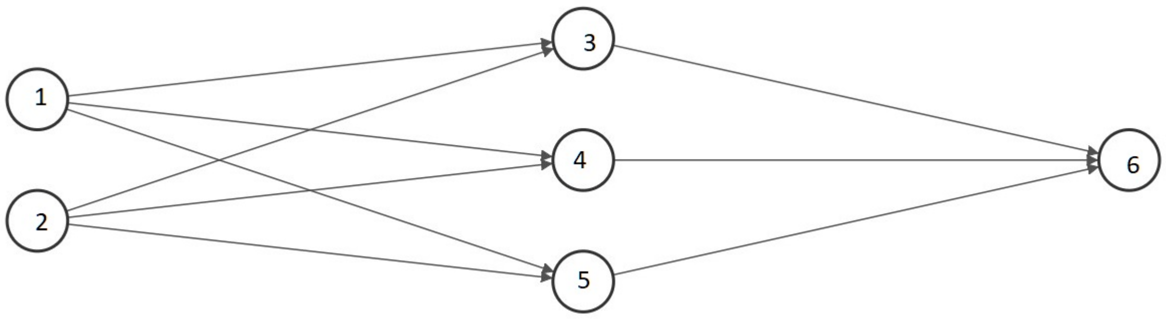


Figure 21: Simple MLP

Since it is a regression task, *MSE* is the right Loss function to be used.

$$\text{Loss} = J = \frac{\sum(y^{predicted} - y^{actual})^2}{N}$$

But for now let us ignore the $\sum$ and division by sample size.

$$\frac{\delta J}{\delta w_{ij}} = 2(y^{predicted} - y^{actual})$$

(lets ignore 2 for now.)

The equation gives us the contribution of weight w_{ij} towards the loss of the model.
But,

$$y^{predicted} = W_2^T \phi(W_1^T X),$$

where the X is the input vector and W_1, W_2 is weight matrix of layer 1 and 2 respectively. Which basically means $y^{predicted}$ is a linear combination of activation's of previous layer. As seen above $y^{predicted}$ can be written as $W_2^T \phi(W_1^T X)$,which is a combination of W_2^T and $\phi(W_1^T X)$. The term $\phi(W_1^T X)$ is a non linearity over the output of Layer-1 ($W_1^T X$). So we can clearly see that chain rule can be applied here to fetch contribution of weights of Layer-1 (W_1) towards the Loss J, which is given by,

$\frac{\delta J}{\delta W_1} = \frac{\delta J}{\delta y^{predicted}} \frac{\delta y^{predicted}}{\delta \phi(W_1^T X)} \frac{\delta \phi(W_1^T X)}{\delta (W_1^T X)} \frac{\delta (W_1^T X)}{\delta W_1}$

Substituting the concerned values as stated above, the above equation transforms into

$\frac{\delta J}{\delta W_1} = \frac{\delta J}{\delta (W_2^T \phi(W_1^T X))} \frac{\delta (W_2^T \phi(W_1^T X))}{\delta \phi(W_1^T X)} \ \phi(W_1^T X)' \ X$

In the above equation

$$W_1 = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix} \text{ and } W_2 = \begin{bmatrix} w_1^{(2)} \\ w_1^{(2)} \\ w_3^{(2)} \end{bmatrix}$$

which brings us to the equation

$\frac{\delta J}{\delta W_1} = (y^{predicted} - y^{actual}) \ W_2^T \ \phi(W_1^T X)' \ X$

As we have obtained gradient it is pretty straight forward to get the updated weights, which is given by

$$W_1^{(new)} = W_1^{(old)} - \alpha \ ((y^{predicted} - y^{actual}) \ W_2^T \ \phi(W_1^T X)' \ X)$$

The above is the method to update the weights of the network, moving towards the output from the given input is coined as feed-forward network. While moving backwards to update weights is referred as back-propagation. During this back-propagation the weights of the network are update based on their contribution towards the loss.

Prime thumb rule in using back-propagation is that the activation function should be differentiable. And if the activation function we choose leads to very small gradient of loss then we might end up in a situation of vanishing gradient, where it is very hard to optimize weight, whereby hard to train the network.

Click on the following reference links

References

[1] Youtube Playlist on Neural Networks by 3blue1brown

[2] Back-propagation and Working of neural net with example

[3] Convergence Proof

[4] Linear Regression Vs Neural Network

[5] Multi Label Classification

[6] More into Neural Networks