S21CSE471: Statistical Methods in AI **Spring 2021**

Perceptrons and Back Propogation on Neural Networks

Prepared by: Varun Chhangani(2019121011), Siddarth Vijay(2018113020), Sushant Kumar(2020201003)

"Emotions are enmeshed in the neural networks of reason" - Antonio Damasio

In this scribe we will discuss the most trivial and basic types of Neural Networks, Perceptrons, of single and multiple layers; and then grow the idea to a Neural Network. We will see how we can train those Neural Networks using Back Propagation and discuss few other methods of training the Neural Networks.

1 Introduction - Artificial Neural Networks (skippable)

Some of the earliest learning algorithms we recognize today were intended to be computational models of biological learning, i.e. models of how learning happens or could happen in the brain. As a result, one of the names that deep learning has gone by is artificial neural networks (ANNs). The corresponding perspective on deep learning models is that they are engineered systems inspired by the biological brain (whether the human brain or the brain of another animal).

The brain is the fundamental part in the human body. It is the biological neural network which receives the inputs in the form of signals and processes it and send out the output signals. The fundamental unit of the brain is the Neuron. Experts define ANN as "a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs."

Similar to the brain, the Artificial Neural Network, imitates this biological Neural Network of human body. A neural network is a network or circuit of neurons, 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. As a result, we ought to know the basic building blocks of ANN which is the Perceptron.

2 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, $x_1, x_2, \ldots,$ and produces a single binary output.

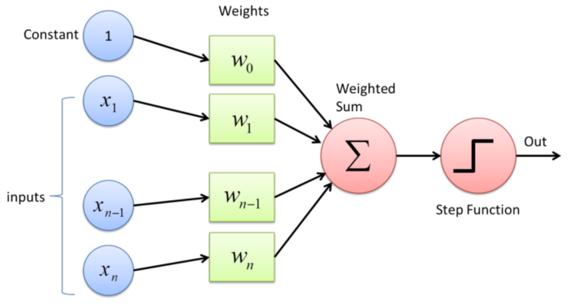


Figure 1: Perceptron

The perceptron has 4 parts:

- Input values / Input Layer
- Weights and Bias
- Net Sum
- Activation Function

In the figure 1, the perceptron has n inputs, $x_1, x_2, \ldots, x_n$ and a bias term. Rosenblatt proposed a simple rule to compute the output. He introduced weights, $w_1, w_2, ...$, real numbers expressing the importance of the respective inputs to the output; and bias w_0, the value to shift the value of net sum so as to shift the activation function curve.

Each input $x_0 (= 1), x_1, \ldots, x_n$ is multiplied with the respective weight and is summed.

$$net = \sum_{i=0}^{n} w_i x_i$$

This net sum is then passed through the activation function, that is signum in case of perceptron to get the final output.

$$\begin{aligned} out = sign(net) = sign\left(\sum_{i=0}^{n} w_i x_i\right) &= sign(W^T X) \\ W &= [w_0 \, w_1 \, \cdots \, w_n]^T \\ X &= [1 \, x_1 \, \cdots \, x_n]^T \\ \therefore out &= \begin{cases} 1 & W^T X > 0 \\ 0 & W^T X < 0 \end{cases} \end{aligned}$$

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.

2.1 Training Perceptrons

Let's first set up a Loss function for the Perceptron. What is loss functions will be later discussed in the section 7.

$$\begin{aligned} L_{\text{Perceptron}}(f(X), y) &= max(0, -yf(X)) \\ f(X) &= W \cdot X = W^T X, \, y \in \{0, 1\} \\ (X, y) &\in S \end{aligned}$$

Where S is the training set.

- if $yf(X) \geq 0$ (correct classification), then the loss is 0
- if $yf(X) < 0$ (misclassification), then the loss is $-yf(X)$
- Thus, the loss increases lineraly with the margin of misclassification

Thus, the gradient becomes:

$$\nabla_W L(f(X), y) = \begin{cases} -yX & yf(X) < 0 \\ 0 & yf(X) > 0 \end{cases}$$

And thus the update equation is:

$$W^{t+1} = \begin{cases} W^t + \eta y X^t & yf(X^t) < 0 \\ W^t & yf(X) > 0 \end{cases}$$

It is suggested to read the proof of convergence of perceptron for linearly separable data by yourself at [3].

2.2 Changing Activation Functions

Without any activation function, or say, activation function $g(z) = z$, a perceptron is basically a linear regressor. We need to have activation functions to decide the range of outputs of our perceptron.

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. This basically means that the activation function must have a gradient for the a huge variety of loss functions, unlike the hand-made loss function in the case of signum loss function.

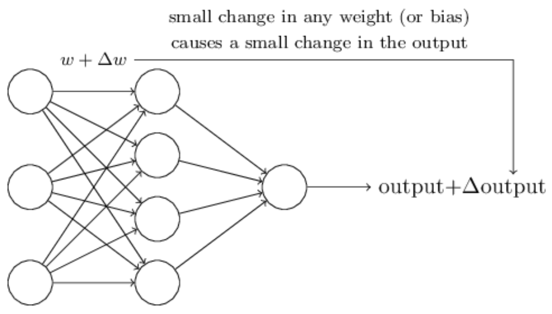

small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

Figure 2: Updating weights change Output

To emulate effects of signum activation function, but with gradient available, we use **Sigmoid activation function**.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Similarly, we have tanh, relu and more activations functions that help us train perceptrons for different ranges and according to different requirements,

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.

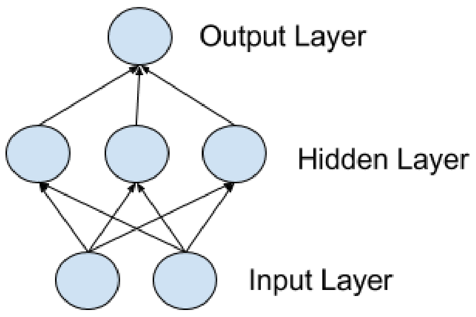


Figure 3: Network

Input Layer 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.

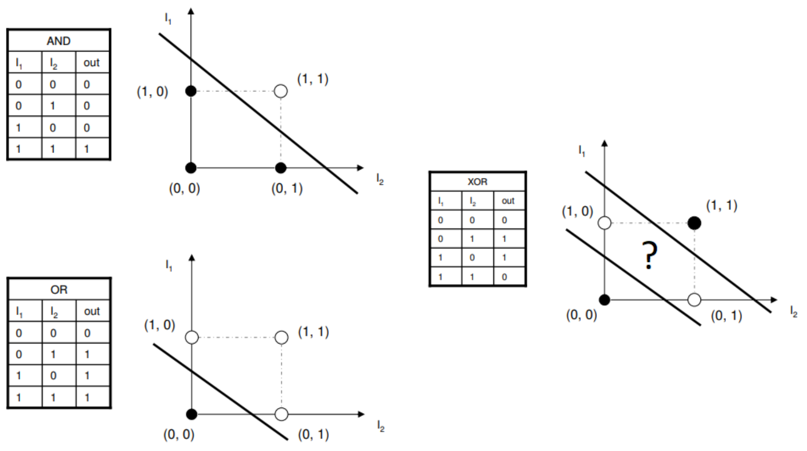
Hidden Layer(s) Layers after the input layer are called hidden layers because that are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value. Deep learning can refer to having many hidden layers in your neural network. Hidden Layers may be of several kinds, fully connected, convolutional, dense, etc. Furthermore, there may to several skip connections in the hidden layers so as to bypass the problem of diminishing gradient.

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.

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.

One such problem encountered in the Rosenblatt perceptron is inability to solve the XOR problem.



AND

I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

OR

I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

XOR

I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4: Single Layer can’t solve XOR

5 Perceptron Learning Algorithm

The algorithm for training MLP is as follow:

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 (W_h). 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).

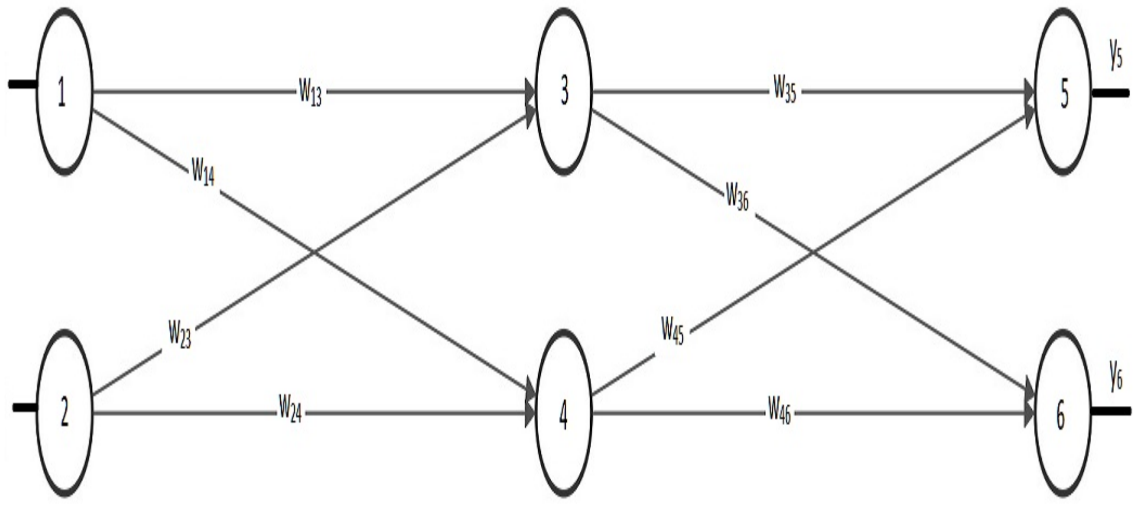


Figure 5: MLP Network

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 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.

$$y_5 = f_5(w_{35}f_3(w_{13}y_1 + w_{23}y_2) + w_{45}f_4(w_{14}y_1 + w_{24}y_2))$$
$$y_5 = f_5(w_{35}f_3(w_{13}f_1(x_1) + w_{23}f_2(x_2)) + w_{45}f_4(w_{14}f_1(x_1) + w_{24}f_2(x_2)))$$

6 Linear Regression vs Neural Networks

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:

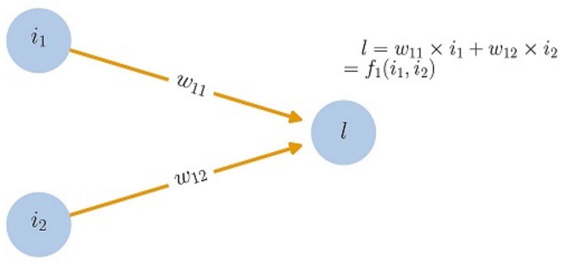


Figure 6: Linear Regression

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

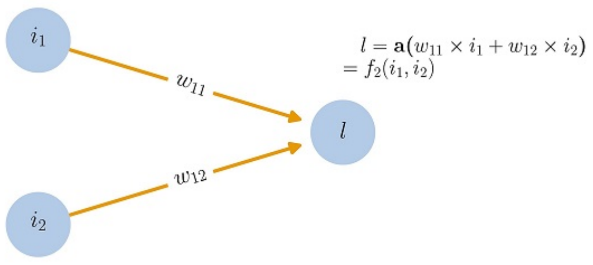


Figure 7: Non Linearity a

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

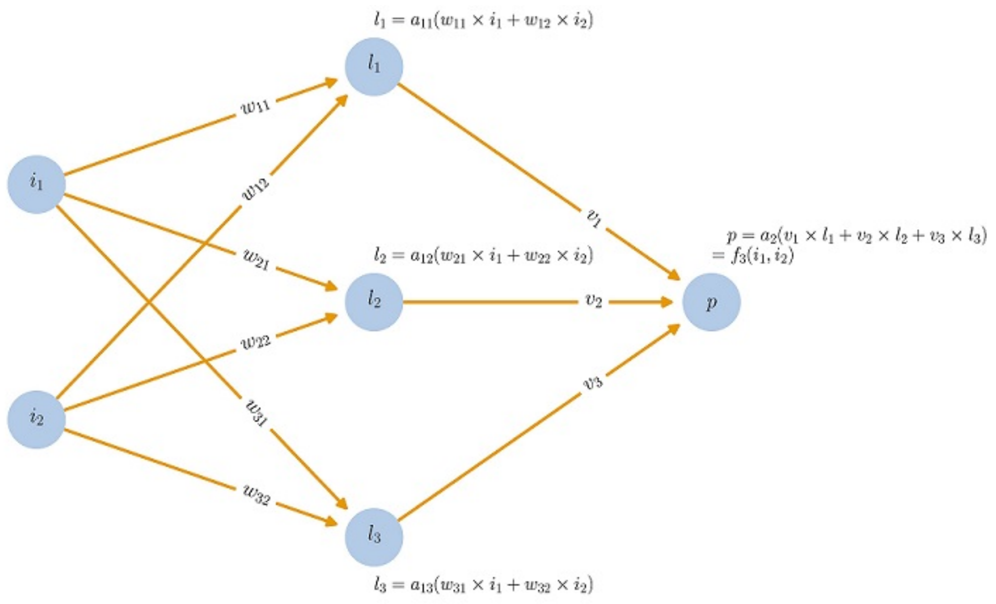

$l_1 = a_{11}(w_{11} \times i_1 + w_{12} \times i_2)$
l_1
w_{11}
w_{12}
i_1
w_{21}
$l_2 = a_{12}(w_{21} \times i_1 + w_{22} \times i_2)$
l_2
v_1
v_2
$p = a_2(v_1 \times l_1 + v_2 \times l_2 + v_3 \times l_3)$
$= f_3(i_1, i_2)$
p
w_{22}
i_2
w_{31}
v_3
w_{32}
l_3
$l_3 = a_{13}(w_{31} \times i_1 + w_{32}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

Figure 8: Neural Network with non-linearity a

7 Loss Function

Lets look at a simple loss function calculation for the case of regression task

- Regression
  The MSE or Mean square error is commonly used as the loss function for regression. We are gonna take the example of neural network with one neuron on the final layer and it returns a continuous numerical value, for which, we use a ReLU or Linear activation function.

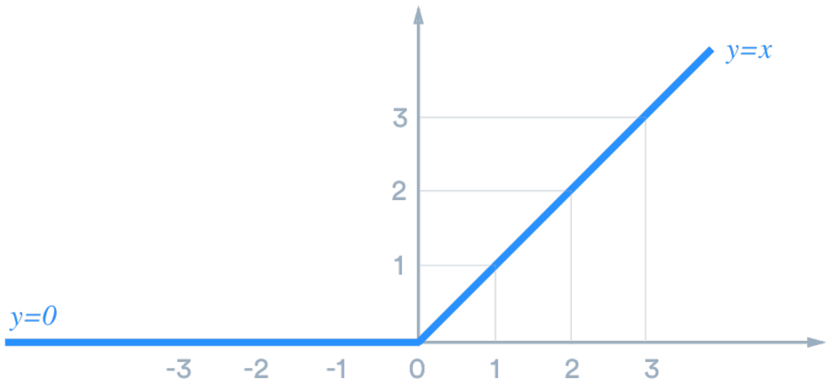


Figure 9: ReLU

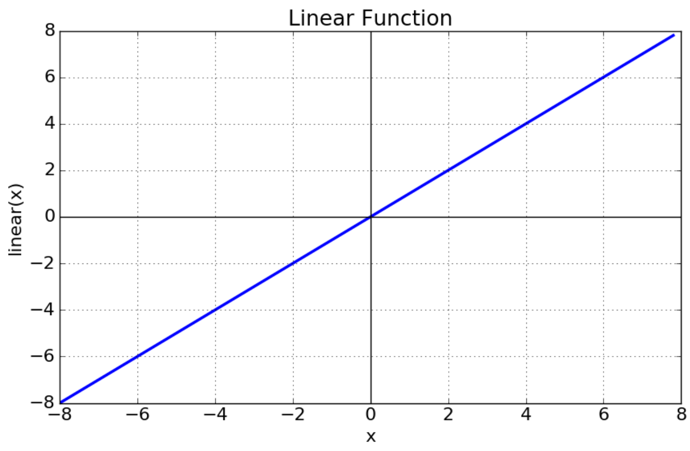


Figure 10: Linear Activation

The obtained value at output neuron is compared to the 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). While the main one is MSE, we have other loss functions too, like, MAE and MSLAE.

  - $MSE = \frac{1}{N}\sum_{i=0}^{N}\left(y_i^{\text{predicted}} - y_i^{\text{actual}}\right)^2$
  - $MAE = \frac{1}{N}\sum_{i=0}^{N}\mid\left(y_i^{\text{predicted}} - y_i^{\text{actual}}\mid\right.$
  - $MSLE = \frac{1}{N}\sum_{i=0}^{N}\left(\log\left(y_i^{\text{predicted}} + 1\right) - \log\left(y_i^{\text{actual}} + 1\right)\right)^2$

- Classification
  Classification IS regression in this case. Neural networks approximate a function which outputs the sufficient statistics of some probability mass/density function. It's your interpretation of the output of a neural network that gives rise to classification as being distinct from regression. Neural

Networks learn the distribution over classes in the feature space.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.

The softmax function, also known as softargmax or normalized exponential function, is a generalization of the logistic function to multiple dimensions. It is used in multinomial logistic regression and is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

But what is the use of softmax? Why not just use logits or probabilities?
Lets look at the graph of the logit function

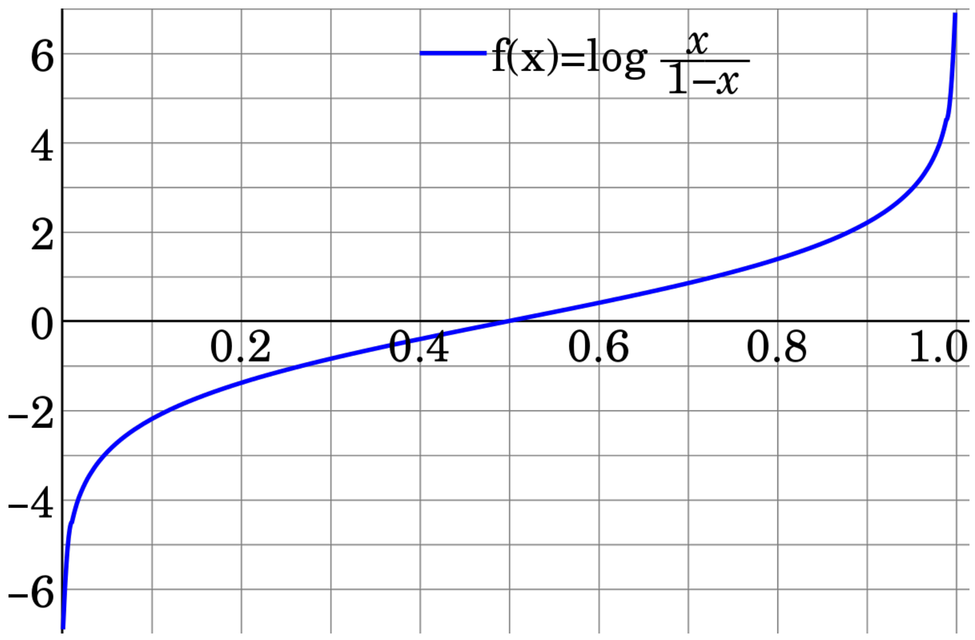


Figure 11: Logit Function

- We can clearly see that range of logit is from negative to postivie infinity and adding negative values does not give a correct normalization. But exponent of a logit lies in the range of zero to infinity.

- 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.

- So we first apply exponentiation on logits for a k class case, then in the next step we obtain probabilities from exponential vector is the prediction of the network.

- Where each ith element is the probability that given input sample belongs to the ith class of potential classes [C1 C2 C3 ... Ck]. And the output label is the Cj class where

- Now lets look at a simple multiclass classification task. The Softmax layer will give us the probability that the given test sample belongs to a class. It will be in its respective position in the output vector from the softmax.

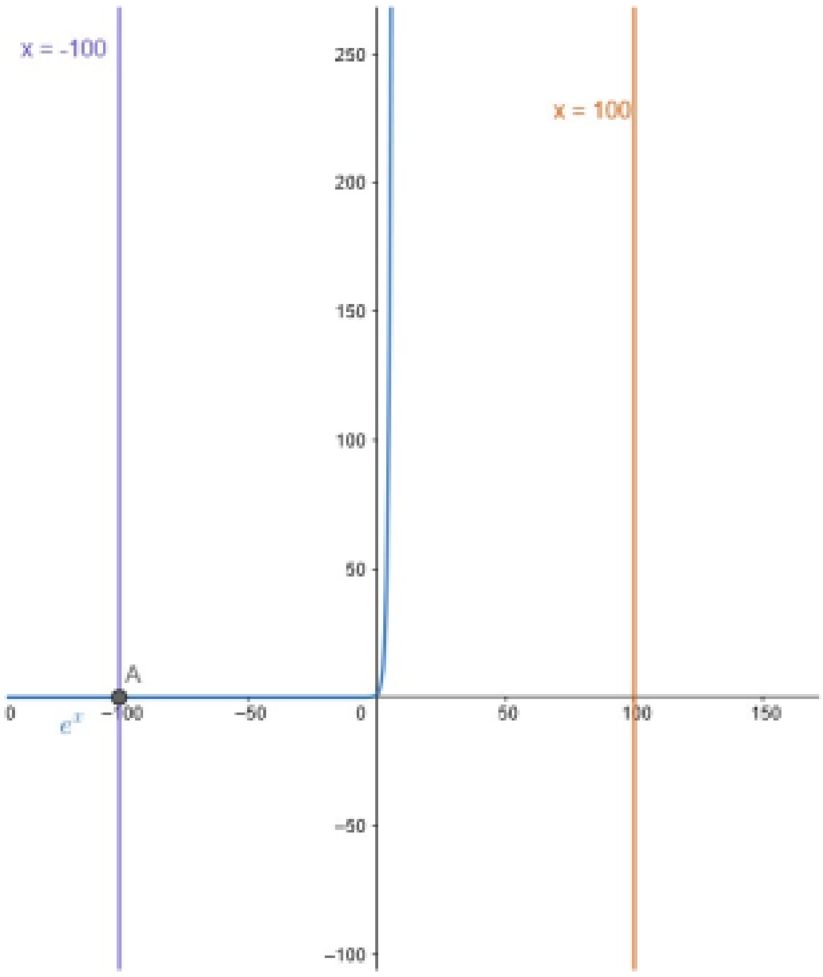


Figure 12: Exponentials

So suppose the vector is [0.4 0.3 0.05 0.05 0.2], you can understand this to mean that the Probability of class 1 being the input is 0.4, the probability of class 2 being the input is 0.3, the probability of class 3 being the input is 0.05, the probability of class 4 being the input is 0.05, the probability of class 5 being the input is 0.2. This is where cross entropy comes in.

But before that we need to understand the need for applying log on the probabilities obtained from softmax layer. Lets say two classes got the same probability, so now the vector is [0.4 0.3 0.05 0.05 0.2] we get [-0.916 -1.203 -2.995 -2.995 -1.609]

We can see that when probability tends to 0, the log tends to negative infinity. And to see it clearer, we take the negative log, and then miss-classification of a particular input sample incurs a high penalty. That is if the probability is low, the negative log is a huge positive number. In addition to this we need the ground truth as well.

So basically if the first sample is of class 1, then the Probability of class 1 being the input is 1.0, the probability of class 2 being the input is 0.0, the probability of class 3 being the input is 0.0, the probability of class 4 being the input is 0.00, the probability of class 5 being the input is 0.0.

Now this would just be the one hot encoding of the classes. So now, we need to make this $Y^{\text{predicted}}$ to be equivalent with Y^{actual} . We need to increase the correct class probability and decrease the wrong class probability.

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

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

Figure 13: Logits

$$Y^{predict} = \begin{bmatrix} \frac{e^{y_1}}{\sum_{i=1}^{k} e^{y_i}} & \frac{e^{y_2}}{\sum_{i=1}^{k} e^{y_i}} & \frac{e^{y_3}}{\sum_{i=1}^{k} e^{y_i}} & . & . & . & \frac{e^{y_k}}{\sum_{i=1}^{k} e^{y_i}} \end{bmatrix}$$

Figure 14: Probabilities from exponential vector

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

Now lets try to understand this,
The ground truth = $P = Y^{\text{actual}}$ = [1 0 0 0 0]
The prediction = $Q_1 = Y^{\text{predicted}}$ = [0.4 0.3 0.05 0.05 0.2]
$H\,(P, Q_1) = -(1.\log(0.4) + 0.\log(0.3) + 0.\log(0.05) + 0.\log(0.05) + 0.\log(0.2)) \approx 0.916$

We see that if the P for any class is high, cross entropy goes to zero.

The ground truth = P = Y^{actual} = [1 0 0 0 0]
The prediction = $Q^{\text{hypothetical}} = Y^{\text{predicted}}$ = [0.4 0.3 0.05 0.05 0.2]
H(P, $Q^{\text{hypothetical}}$) = 0 Here we see that as Q approaches P, cross entropy goes to 0

As the epochs increase we see that the loss decreases, the negative log of probability of class 1 approaches 1. Now, the cross entropy is,

The ground truth = P = Y^{actual} = [1 0 0 0 0]
The prediction = $Q^{\text{kth iteration}} = Y^{\text{predicted}}$ = [0.98 0.01 0 0 0.01]
H(P, $Q^{\text{kth iteration}}$) = 0
We can see the cross entropy is almost 0

8 Training & Updating Weights

The training step is like Linear Regression, basically involving learning of weights. In which the key 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 $node_j$). $\frac{\delta_J}{\delta_{w_{ij}}}$, 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:

$$\frac{\delta_y}{\delta_x} = \frac{\delta_y}{\delta_z}\frac{\delta_z}{\delta_p}\frac{\delta_p}{\delta_x}$$

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 $\phi(x)$

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

Figure 15: $Y_predicted$

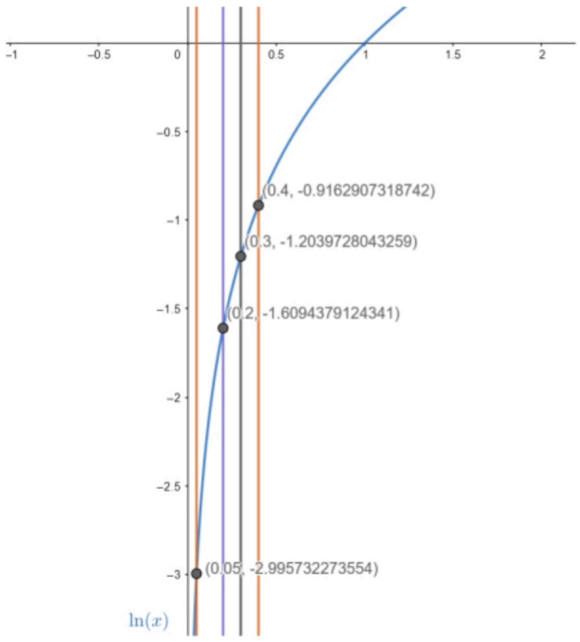


Figure 16: $Y_predicted$

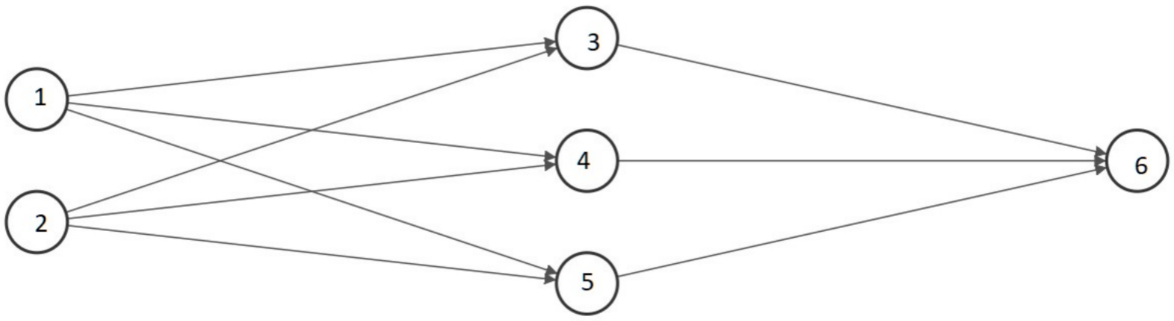


Figure 17: Simple MLP

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

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

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

$$\frac{\delta_J}{\delta_{w_{ij}}} = 2(Y_{predicted} - Y_{actual})$$

Let us 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,

$$\frac{\delta_J}{\delta_{W_1}} = \frac{\delta_J}{\delta_{Y_{predicted}}} \frac{\delta(W_2^T \phi(W_1^T X))}{\delta(\phi(W_1^T X))} \phi(W_1^T X)' X$$

In the above equation,

$$W_1 = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix}$$

and

$$W_2 = \begin{bmatrix} w_1^{(2)} \\ w_2^{(2)} \\ w_3^{(2)} \end{bmatrix}$$

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

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

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.

References

[1] Deep Learning, Ian Goodfellow and Yoshua Bengio and Aaron Courville

[2] Activation Functions in Neural Networks https://towardsdatascience.com/activation-functions-in-neural-networks-83ff7f46a6bd

[3] Convergence Proof, Mitesh M. Khapra, https://www.cse.iitm.ac.in/ miteshk/CS7015/Slides/Teaching/pdf/Lecture2.p

[4] http://neuralnetworksanddeeplearning.com/