# Deep Learning Assignment 1

### Karthik Murugadoss – R00183157

### PART A

#### Question 1\_1: Building a SoftMax Classifier(Vectorized Implementation)

The python implementation of the code is available in the file “Question1\_1.ipynb”

The overall aim of the code is to send in all training data into a SoftMax layer of 10 output units since there are 10 classes and get the respective probabilities for each of the training instance i.e. since there are 10 classes and 60000 training instances the output of the SoftMax layer would be 10 x 60000 which is 10 probabilities for each training instance.

The following are the functions used to build the SoftMax classifier:

**Forward\_pass:**

**Code:**

****

**Explanation:**

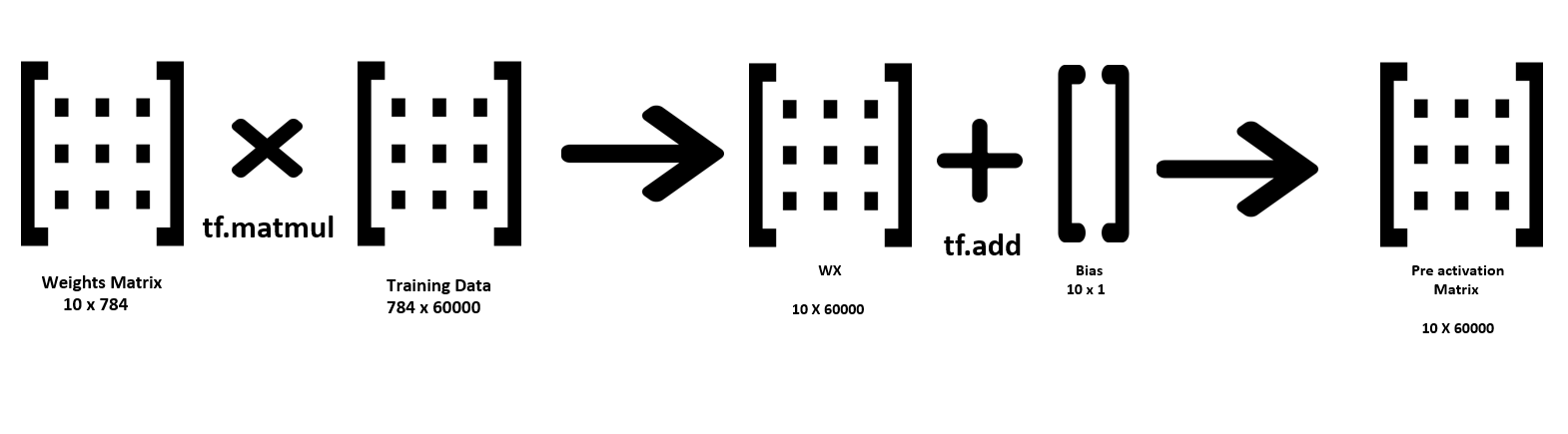
In the forward pass function the entire training data is sent to the SoftMax layer and output probabilities for all classes of each training instance is given out.

This function takes as arguments the weights(**W**), reshaped training data(**X**) and the bias(**b**).

First step is to create the pre-activation matrix (**A1**) which is W.X + b

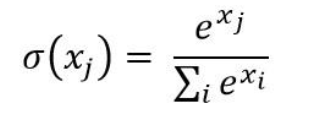
The weights for each of the output units is multiplied with the input data using tf.matmul and the result is added with the bias **b**  is added using tf.add ( tf.add by default does broadcasting and adds the values just as in numpy)

Below is the pictorial representation of the same



Second step is to implement the softmax activation function to A1

In detail, in the softmax function the below formula is to be implemented



Same can be explained using an example matrix and Let’s assume the below matrix is A1 each column vector represents one training instance.

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 3 | 1 | 1 |

The exponent value of each of the elements in A1 is calculated using tf.math.exp()

|  |  |  |
| --- | --- | --- |
| 2.718282 | 7.389056 | 20.08554 |
| 54.59815 | 148.4132 | 403.4288 |
| 20.08554 | 2.718282 | 2.718282 |

Then the summation of each column vector is calculated using tf.reduce\_sum(A1, 0) and stored as

|  |  |  |
| --- | --- | --- |
| 77.40197 | 158.5205 | 426.2326 |

Sum\_of\_t =

Then each element is divided by its appropriate column sum which would result in the probabilities which is achieved using tf.divide()

|  |  |  |
| --- | --- | --- |
| 0.035119 | 0.046613 | 0.047123 |
| 0.705385 | 0.93624 | 0.946499 |
| 0.259496 | 0.017148 | 0.006377 |

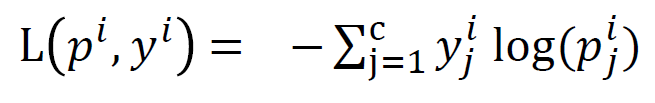
Each column represents each training instance and each row represent the class probability.

In this example we can see that for the first column the value for the second class is more which denotes that the probability of training instance 1 being Class 2 is more.

This is the final output of the softmax function. This value is then clipped by an upper bound of 1.0 and a lower bound of 1e-10 to avoid the probabilities reach 0 which would result in NaN values when log is calculated in the cross entropy function.

**Cross\_entropy:**

The function cross entropy is used to find the loss of the predicted probabilities from the softmax layer. This is achieved using the Loss function as shown below and then the cost is calculated by taking the mean of the Loss values.



The code implementation for the same is shown below



The cross entropy function takes two arguments H and y. H is the output of the softmax layer and y is the one hot encoded correct labels.

To explain the above code lets use the same example shown above. The output of the softmax layer is shown below.

|  |  |  |
| --- | --- | --- |
| 0.035119 | 0.046613 | 0.047123 |
| 0.705385 | 0.93624 | 0.946499 |
| 0.259496 | 0.017148 | 0.006377 |

Lets consider the one hot encoded y values are

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |

Using tf.math.log we calculate the log of all values in H

|  |  |  |
| --- | --- | --- |
| -3.34901 | -3.06588 | -3.05499 |
| -0.34901 | -0.06588 | -0.05499 |
| -1.34901 | -4.06588 | -5.05499 |

When this is multiplied with one hot encoded y values using tf.math.multiply the below output is produced which is then summed using tf.reduce\_sum(input, 0). We use tf.math.multiply because we need element wise multiplication here.

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | -0.06588 | -0.05499 |
| -1.34901 | 0 | 0 |

Once the column wise sum is found then the cost is calculated by estimating the mean of all the values susing tf.reduce\_mean() which is total\_loss

**Calculate Accuracy:**

**Code:**

****

This function takes in input arguments reshaped\_data **x**(training or test), the correct one hot encoded labels **y**(training or test) , weights **w**  and the bias values **b.**

The first step in the function is to find the output of the softmax layer which is nothing but the output probabilities. Then once that is obtained the index of the max probability is obtained using tf.math.argmax() for both the predicted probabilities and for the actual y labels. If these index value for each entry matches then the prediction is correct or it is considered to be wrong. The same can be explained using a sample.

|  |  |  |
| --- | --- | --- |
| 0.035119 | 0.046613 | 0.047123 |
| 0.705385 | 0.93624 | 0.946499 |
| 0.259496 | 0.017148 | 0.006377 |

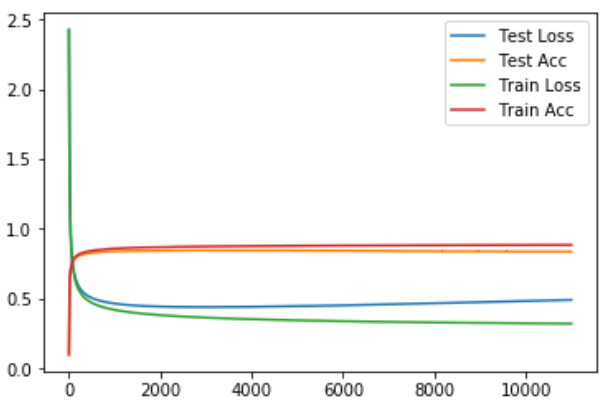
Lets assume the above matrix is the output probabilities of the soft max layer. When we do an argmax we get the output of vector [ 1 , 1, 1] . Similarly, the same is done for the y tensor. Let’s consider that the y tensor is as shown below

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |

From the above tensor when we get the argmax values to be [2 , 1 , 1 ] by using the two vectors the accuracy is calculated using the function tf.equal() 🡺 [0, 1, 1 ] and the mean is calculated using tf.reduce\_mean() 🡺 0.666 ( which is the accuracy)

**Evaluation:**

The below graph shows the performance of the softmax classifier on both training and test data. The X axis shows the epochs. To understand how the model performs and to see clearly when over fitting occurs the model was trained for 10000+ epochs. It can be interpreted that the model performed well and both training and test data values were identical until 500 epochs. In the first few epochs we can see that there is a tremendous reduction in the loss and good improvement in the accuracy for both training and test data. Post the 1000 epoch approx. we can see that the test and train curves start to separate. The training accuracy continues to increase at a slower rate, while the test accuracy starts to plateau and then reduces slightly. While the Test loss starts to increase, and the train loss continues to drop. This definitely means that the model has started to overfit to the training data. Since this is a simple dataset the difference is very less when a more complex dataset is used the overfitting can be observed very clearly.



#### Question1\_2: Introducing additional ReLu based layers

The goal of this task is to introduce hidden ReLu based layers and to analyse the performance of the neural network

**Network Architecture A:**

**Code:**

****

**Explanation:**

As shown in the pictorial representation for Question1\_1 a similar implementation is carried out here with addition layer introduced before the Softmax Layer. This function takes weights and bias of both the layers along with the reshaped input training data.

A hidden layer of 300 neurons with ReLu activation unit is added and tf.nn.relu is added to implement the function and the output of the hidden layer 1 **(H1)** is given as the input to the softmax layer and the final output is **H2**

**Network Architecture B**

**Code:**

****

**Explanation:**

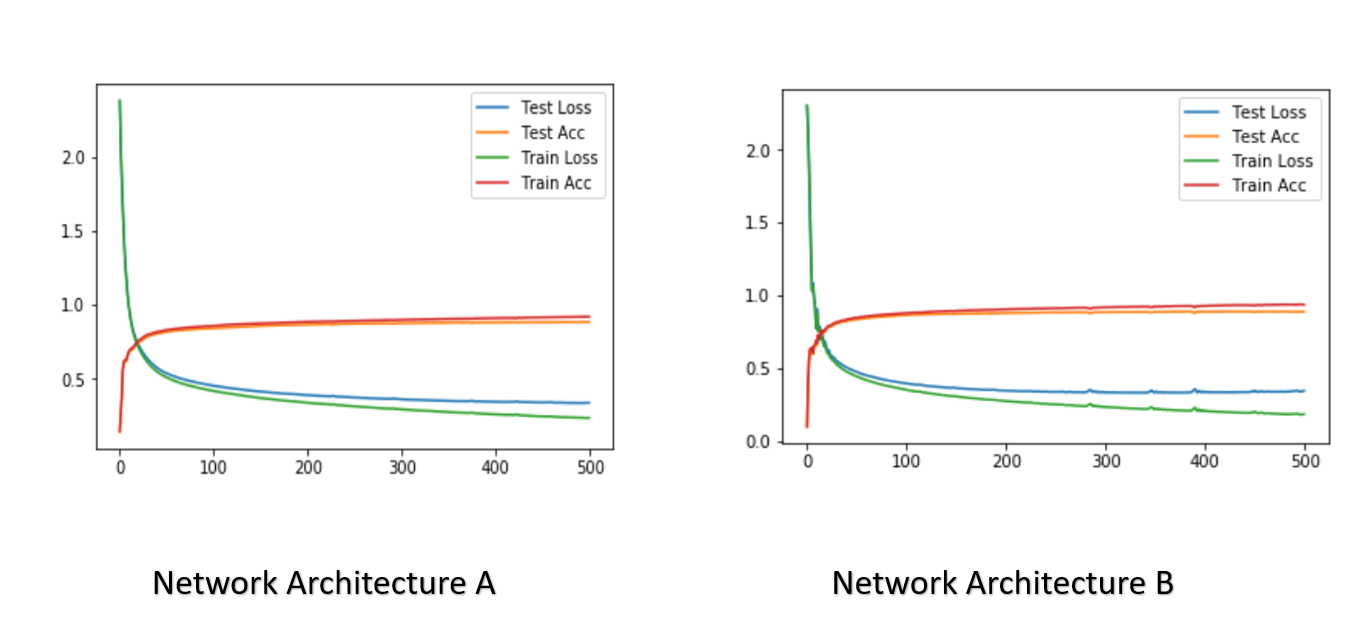
Similar to the network architecture A this is also implemented where an additional layer is added

The flow of the network goes like this:

W1.X\_reshaped +b1🡺 A1🡺ReLu(A1)🡺H1🡺W2.H1 + b2 🡺A2 🡺ReLu(A2) 🡺 H2 🡺 W3.H2 + b3 🡺 A3 🡺 Softmax(A3)🡺 Predicted value

The function takes all three weights and bias along with reshaped training data as arguments and returns the predicted value as output

**Evaluation of both the architecture**



From the graphs we can see that the more the number of hidden layers the more accuracy we get in lesser number of epochs.

In Network Architecture A after 400 iteration the accuracy was 90 percent where as in Network Architecture B the accuracy at 400th iteration is 92.8 percent.

And this also means that with more number of neurons overfitting can also happen aggressively as seen in the graph the test Loss seems to increase more than that shown in architecture A

#### Question1\_3 Applying L1 and L2 Regularization of Network Architecture B

**L1 Regularization:**

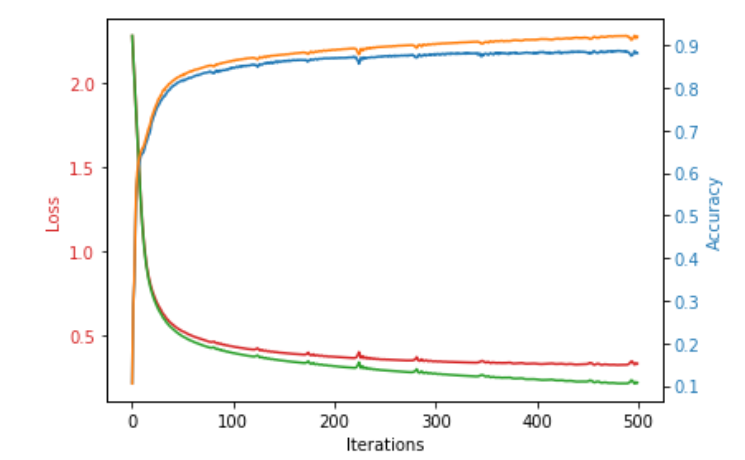
To implement L1 regularization, an extra step of calculating sum of individual weights in each weight matrix across all the weight matrix is added. This is calculated using the function reg\_value\_L1. This function takes in a list of weights as inputs and returns the sum of all elements in each matrix and a cumulative sum of all weight matrices given in a list. After which the reg\_value is calculated by multiplying the value returned from reg\_value\_L1 with regularization rate and added to the previously calculated cross\_entropy\_loss.

**Code:**



**Evaluation:**

From the below graph we can see that with a regularization rate of 0.000001 the model performs as shown below for 500 epochs. We can still see that the model seems to start overfitting, but the accuracy is far better than that of L2 regularization. (Orange Line – Training Accuracy, Blue Line – Test accuracy, Red Line- Test Loss, Green Line – Training loss).



**L2 Regularization:**

The code is implemented in a similar manner just that, instead of reg\_value\_L1 we will be using reg\_value\_L2. The difference in this function is that each element in the matrix is squared and then all the squared elements is summed. This is done across all the weight matrix and a total sum is returned. Once this is calculated the value returned is multiplied with regularization rate and this reg\_loss is added to the total loss

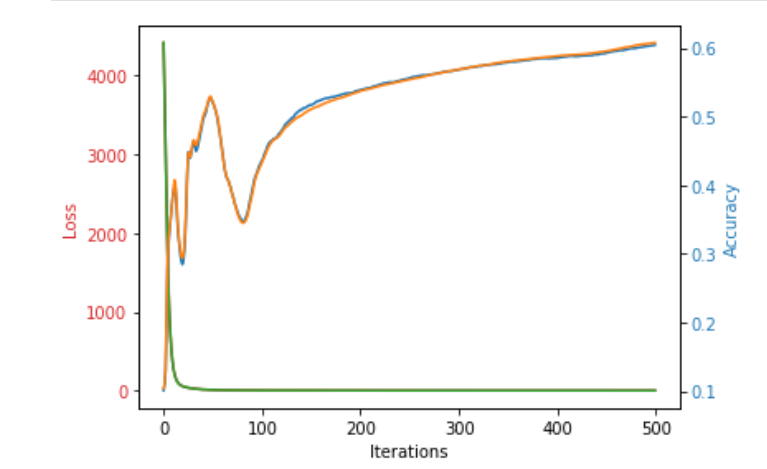
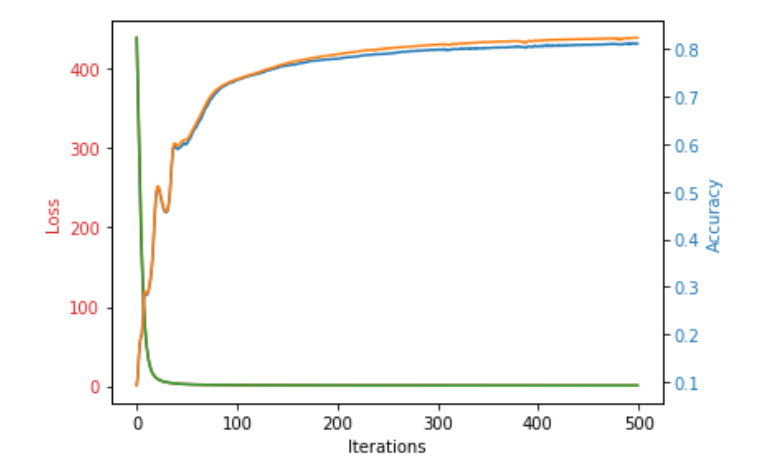
**Code**:



**Evaluation:**

From the below graph we can see that overfitting is reduced by a large extent and the training(orange line) and test(blue line) accuracy lines are almost following a similar trend. We can also observe that there are fluctuations in the initial few epochs this is because during the initial few epochs the weights assigned are random and not close to the ideal weight values. Since we have added weights also as part of the cost function, the model optimization is dependent on the weights which makes the fluctuations occur. To confirm, the regularization rate was increased to see if the fluctuations increase by adding more weightage to the weights which is shown in the second graph. Like expected the

Oscillations are more.



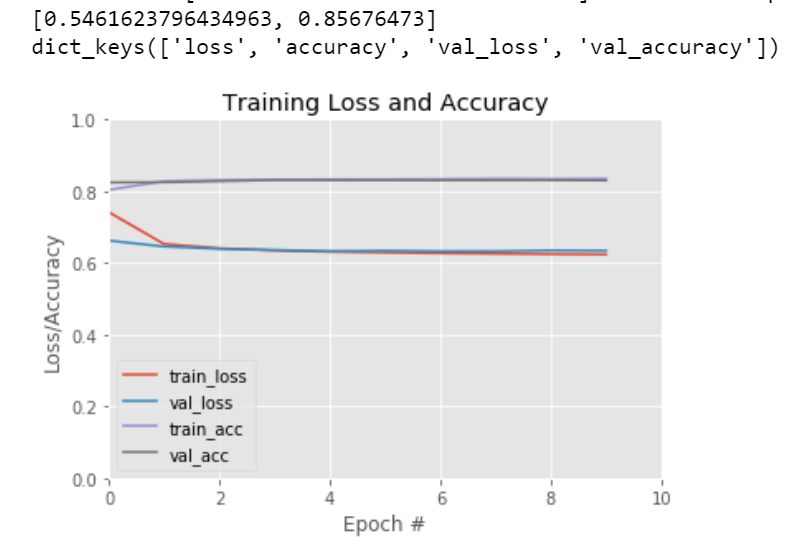
Regularization rate 0.001 Regularization value 0.01

#### Part B

#### Keras – High Level API

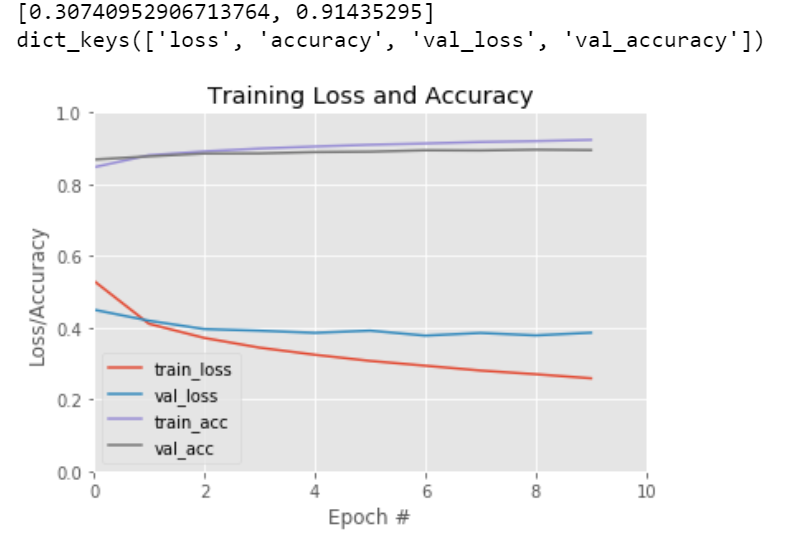
1. Building a softmax classifier using keras with batch size of 256

Accuracy obtained is 85.67%



**2 Layer model output:**

Accuracy achieved is 91 percent approx. for 10 epochs and we can observe from the graph that the validation loss has already started to increase as we added a extra layer and showing signs of overfitting after the 2 epoch



**3 Layer model output**

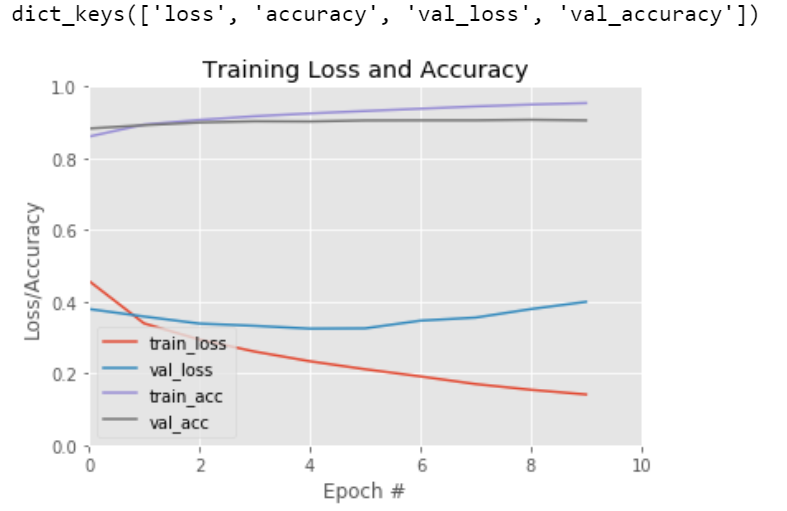
From the below screen shot we can say that by increase an additional layer has increased the accuracy but it also resulted in a drastic increase in the validation loss and from the trend we can tell that since this is a basic dataset having more layers or increasing the epochs would result in overfitting on the training data.



**4 Layer model output:**



**5 Layer Output:**

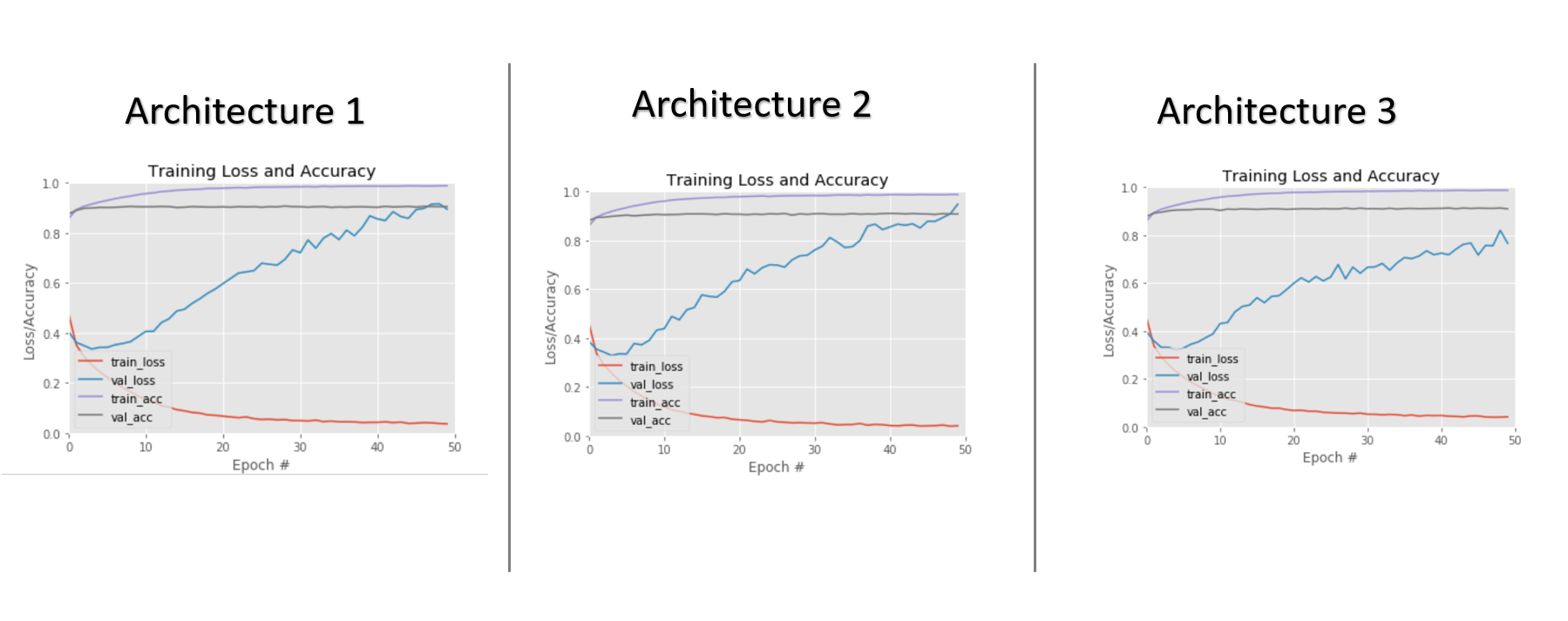


**Comparison of three different network configuration:**

The following are the network architecture which were taken for comparison.

1. L1 400 Neurons L2 200 Neurons L3 SoftMax
2. L1 600 Neurons L2 400 Neurons L3 200 Neurons L3 SoftMax
3. L1 800 Neurons L2 600 Neurons L3 400 Neurons L4 200 Neurons L5 SoftMax

To get a better understanding and clear picture the models were trained to 50 epochs so that we can clearly see the overfitting pattern.



One common observation in all the three architectures is that the more deeper the layer becomes than the required amount the model starts to overfit, which can be interpreted from the above graphs clearly.

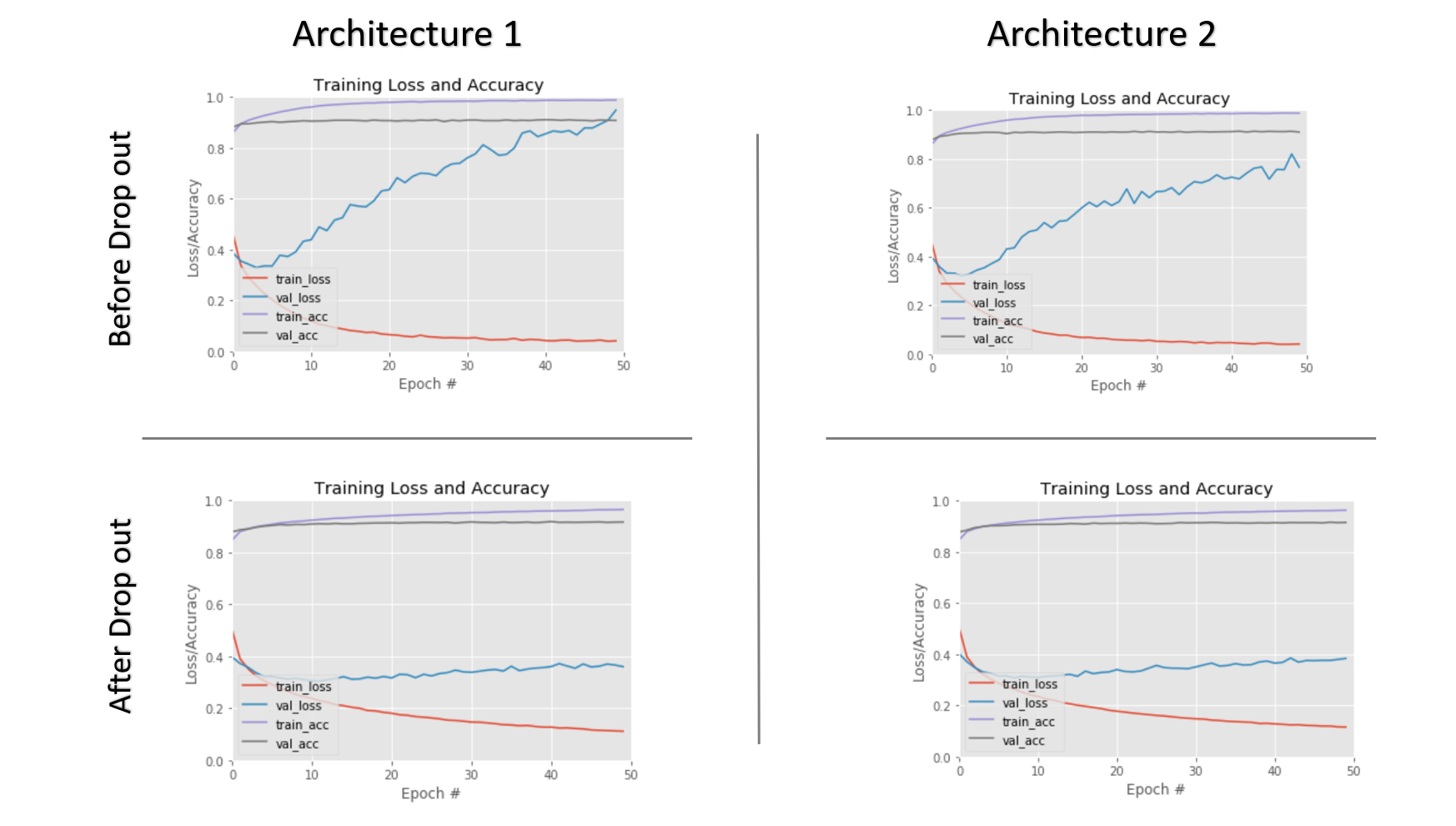
Secondly, just by increasing the number of layers the accuracy has not improved by a large extent. In all the three graphs we can see that the accuracy has plateaued after the 20th epoch.

**(ii)Drop Out Regularization**

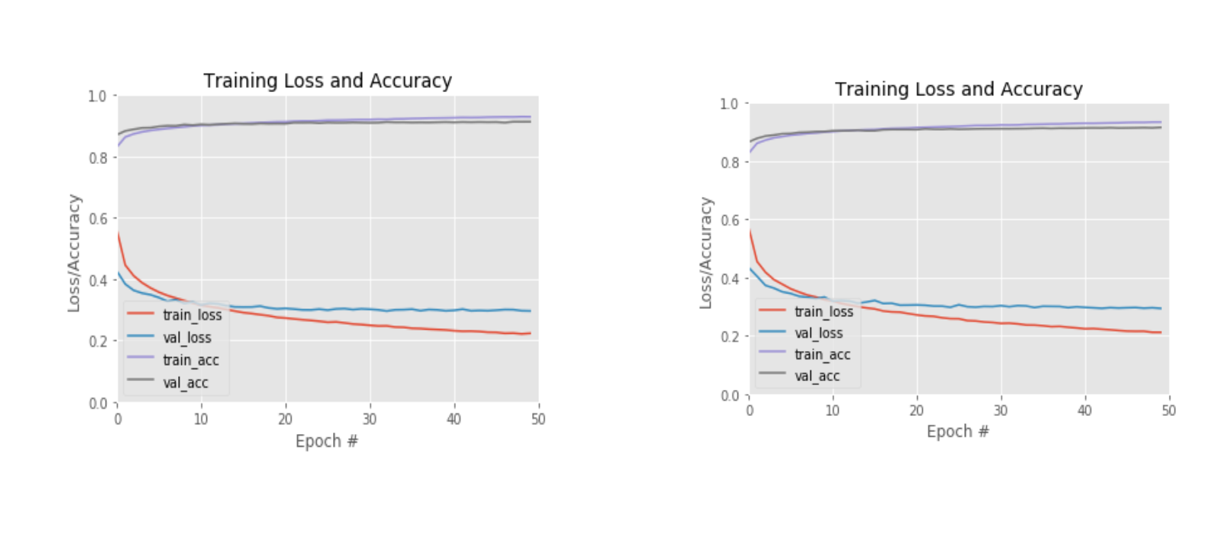
To see the functionalities of the drop out regularization, the following two deep network architecture is used

1. L1 600 Neurons L2 400 Neurons L3 200 Neurons L3 SoftMax
2. L1 800 Neurons L2 600 Neurons L3 400 Neurons L4 200 Neurons L5 SoftMax

The graph below shows the comparison between the performance of the model before drop out regularization and after that and also with two values of drop out 0.2 and 0.4 we can see that the more we increase the drop out value the more the validation loss decreases and smoothens out, which is almost equivalent to reducing the layers If we set the drop out value to be 1 then the entire layer is not considered.



0.2 Drop Out



0.4 Drop out

#### Part C

#### Research topic: Batch Normalization

While there are multiple ways in increasing the performance of the neural networks like increasing the number of layers, increasing number of neurons, adding more data etc. Batch Normalization is one of the key things which ensure that the model performs well especially while training the model with mini batches. The purpose of Batch normalization is to improve the learning process of the neural network by stabilizing the inputs. This is achieved by mean subtracting the inputs and dividing it by the standard deviation before passing it to the activation function. The same can explained using the following process flow. Consider a network of two hidden layers and hence it will have its own weights W1 and bias b1 similarly layer 2 will have W2 and b2. The input to the first layer would be X

Normal Process:

X 🡺 W1X +b1 🡺A1 🡺 act(A1) 🡺H1 🡺 W2H1+b2 🡺 A2 🡺act(A2) 🡺H2

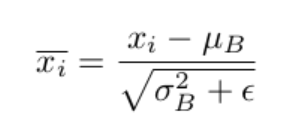
Layer1 Layer 2

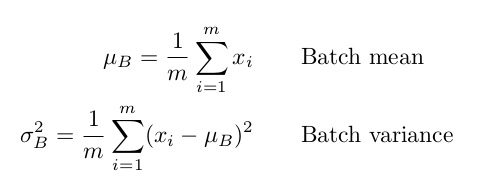
Introducing Batch Normalization:

X 🡺 W1X +b1 🡺BatchNorm(A1) 🡺N1🡺 act(N1) 🡺H1 🡺 W2H1+b2 🡺 A2 🡺 BatchNorm(A2) 🡺N2🡺act(N2) 🡺H2

As seen in the above process the batch normalization is introduced before the applying the activation function and the output of Batch Normalization is N1, N2 respectively for layer 1 and layer 2.

The batch normalization is calculated using the formula





Each training instance is subtracted by the mean of the bath and divided by the sqrt of the standard deviation. To avoid division by zero an extra constant is added in the denominator.

Once it is normalized the final output is multiplied with gamma and added with beta. These are scaling parameters.

Output = (gamma \* normalised value) + Beta

The problem Batch Normalization addresses is the internal covariance shift. Even after normalizing the training data, after each layer the distribution of values vary due to which each layer should constantly adapt to changing distributions in the incoming data which makes the training process a lot slower and slower convergence rates. After every epoch the weights are adjusted in each of the layer based on the distribution of data in that particular epoch. If there is change in the distributions in each epoch they weights will constantly be changing trying to adjust to current distribution and will take a long time to converge. This is where Batch Normalization comes for the rescue. By the normalizing the before sending it to the activation layer in each layer ensures that the weights are constantly adjusted in such a way that the accuracy increases constantly rather than oscillation. The scaling and shifting of data is taken care by two parameters gamma and beta which gets updated after every epoch and based on which the incoming data is normalized rather than only normalizing based on the current epoch’s distribution.

References:

<https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad>

<https://arxiv.org/pdf/1805.11604.pdf>

<https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/>

<https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/>