Deep Learning - Assignment 1

Name : Pavithra Ramasubramanian Ramachandran
Student ID : R00183771

Part A : TensorFlow and the Low Level API

- All solutions implemented with a vectorized approach.
- Tensorflow low level functions used throughout the solutions.
- All tensors are of dtype 'float32'
- Training Data: 60,000 samples with 784 features each (60000,784)
- Testing Data : 10,000 samples with 784 features each (10000,784)

Question1_1 : Building a Softmax Classifier

Code Flow:

1. **Importing Packages and Data**
   The tensorflow version is set to 2.x and all required packages and data are imported as given in the assignment specification PDF.

2. **Reshaping Data for Vectorized Solution**
   - Casting the feature values in both train and test data to 'float32' tensors and reshaping these 2D tensors in such a way that the features become the rows and samples become columns. This reshaping enables a vectorized approach to the solution implementation.
   - The final shape of the features 2D tensor would be (784,60000) for the training samples and (784,10000) for the test samples.

3. **Coefficients(Weights) and Bias**
   - A 2D tensor of shape (10,784) is created using tf.Variable for storing the coefficients (weights) of the features data for all the 10 neurons(classes) and then initialized with random numbers using tf.random.normal such that their mean is 0.0 and standard deviation is 0.05.
   - A tensor variable with initial value of 0.0 is created.

4. **Forward_pass function**
   - This function takes three inputs as parameters, the features tensor, the coefficients tensor and the bias.
   - First, the pre-activation tensor is calculated using the formula :

   (coefficients*features data) + bias

   which is implemented using the low-level tensorflow functions as:

   tf.math.add(tf.matmul(coefficients, tr_x), bias)

   Although the forward_pass function is used to compute both train and test data's predictions, illustration on how the function works on the training data is as follows.

The tf.matmul operation performs the matrix multiplication operation on the coefficients and tr_x tensors of shape (10, 784) and (784,60000) and gives as output a 2D tensor of shape (10,60000) . This output tensor is then added to a bias using the tf.math.add tensorflow function. The tf.math.add function broadcasts the bias tensor to the shape(10,60000) therfore enabling addition of the bias to all elements. The resultant 2D tensor of shape (10,60000) is the pre-activation tensor.

- Then the pre-activation tensor is passed as an argument to another function called the **softmax_activation()** funcion.
- Here the softmax activation performs the softmax activation and outputs a 2D tensor depicting the probabilities with which each of the training samples may belong to the ten classes.
- Softmax activation includes two steps:

  - Calculate e to the power of the pre-actiavtion tensor.

$$t = e^{(x)}$$

where x is (coefficients*features data)+bias, the pre-actiavtion output and is implemented in low level tensorflow functions as:

```
step1 = tf.math.exp(x)
```

- Next we normalize each element by dividing it by the summation of the respective sample's pre-activation outputs.

$$t/\Sigma t$$

where t is the exponentiated pre-actiavtion tensor. The above operation is implemented in low level tensorflow functions as:

```
tf.math.divide(step1, tf.math.reduce_sum(step_1, axis=0))
```

In the above formula, the tf.math.reduce_sum function is used to compute the sum of the values across the 10 neurons for each of the 60000 training samples. For this reason, axis is set to 0. This ensures that the reduce_sum function computes the sum only across the columns and not for the entire 2D tensor. On doing so we get one value for each of the 60000 samples. We then divide the step1 by this to obtain 10 probability values for every training sample depicting the probability with which a training sample may belong to a particular class out of the 10 classes.

- The final prediction tensor with shape (10,60000) is returned as the output of the forward_pass function.

5. **Cross_entropy function**

- The cross_entropy function takes as input two parameters the predictions tensor and the training labels of shape (10,60000).
- Loss for each sample of the data for softmax layer is calculated using the formula:

For every training sample,

$$\text{Loss(predicted value, train label)} = -\Sigma_1^{10}(\text{train label})*\log(\text{predicted vale})$$

which is implement using tensorflow functions as:

```
Loss = tf.math.reduce_sum(tf.math.negative(
tf.multiply(tr_y,tf.math.log(predictions))),axis = 0)
```

The tf.multiply function is used to compute the product of the train label and the log of the prediction element-wise. A negative sum of the products is calculated for every sample of the data. So the resultant of the above operation give a tensor of shape (1,60000) each column representing the loss of that specific sample. Following this, the average softmax loss across all samples is calculated by using the tf.reduce_mean function.

```
tf.math.reduce_mean(loss)
```

- The output of the cross_entropy function is a single loss value.

6. **Calculate_accuracy function**
   - This function takes the predictions tensor and the training labels of shape (10,60000) to compute the accuracy.
   - The values in the predictions tensor are all probabilities such that every column(training sample) sums up to 1 whereas the training labels array is a one hot encoded array where in every sample(column), the correct class has a 1 and all the other wrong classes are 0s. It is difficult to compare these two matrices.
   - However, the **tf.math.argmax** function is used to identify the indices which have the maximum values in the predictions and training labels only across columns so that the class with the highest probability is identified for every training sample.
   - After this, the tensors consisting of these indices are compared with each other using **tf.math.equal** to figure out if the maximum value is at the same index in both the predictions tensor and the training labels tensor. This function outputs a boolean tensor.
   - We now convert this boolean tensor to float32 using **tf.float32** so that all the true values become 1s and all the false values become 0s.
   - Finally the sum of all the 1s is computed using the **tf.reduce_sum** tensorflow function to obtain the accuracy of the model.

7. **Epochs and Gradient Tape**
   - The model is built by using a '**for**' loop to iterate over the forward_pass, cross_entropy and calculate_accuracy functions as many times such that the model is trained well enough to predict the output when given an unknown sample. This is known as epochs in Deep Learning terminology. We choose to run through the function 200 times, that is, the maximum iterations is set to 200.
   - The **GradientTape()** function from tensorflow is used to record the **predictions(forward_pass function) and the loss(cross_entropy function)** in every iteration and the gradients, that is, the **derivatives of the coefficients and bias** are calculated using the loss.
   - The **Adam Optimizer** is then used to **update** the coefficients and the loss with the gradients, thereby, enhancing the prediction ability of the model.

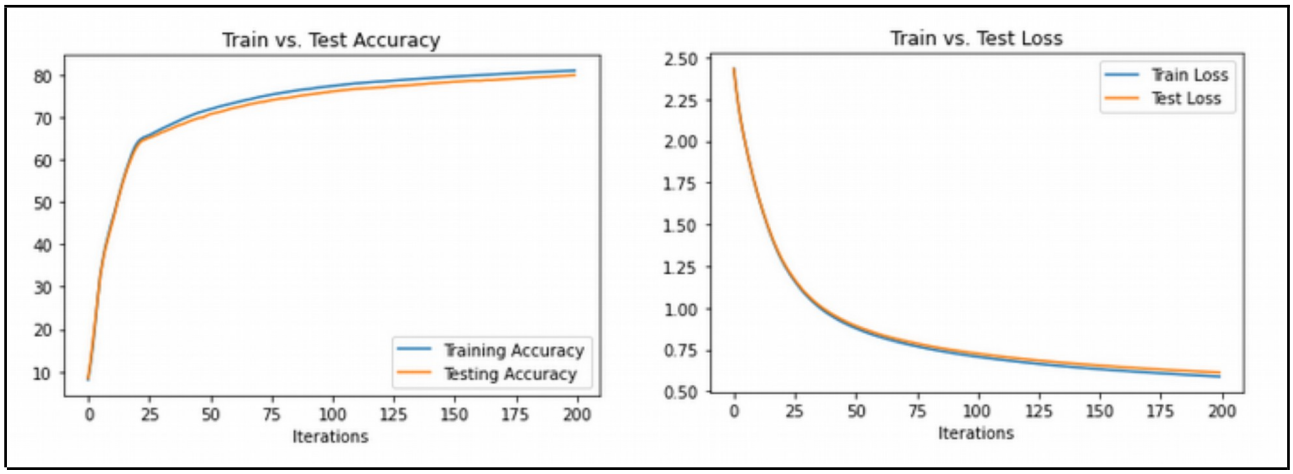
Analysis of the model

On training the model with 200 epochs, the metrics obtained are as follows:

Data/Metrics	Accuracy	Loss
Train Data (60,000)	80.93666%	0.58843166
Test Data (10,000)	79.89%	0.6124331

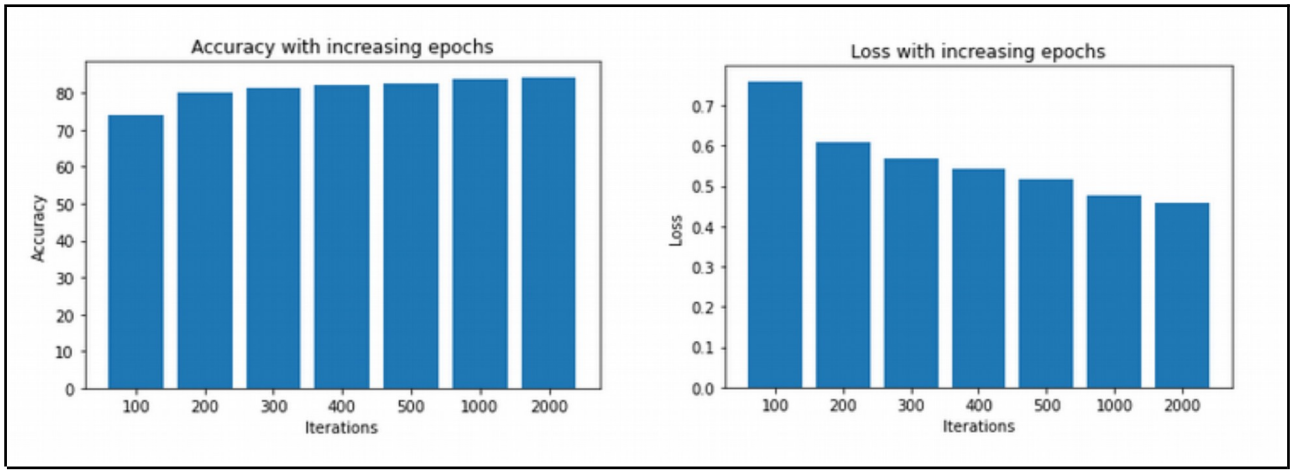
From the above table, it can be inferrred that a softmax layer model can perform fairly well with a test accuracy of 79.89% .

Below are the graphs depicting the accuracy and loss of the train and the test data over iterations.



From the above graphs, it can be observed that the accuracies for both the train and test data increase rapidly until the 25th epoch, then gradually increase until the 100th epoch and the begin to stabilize from there on. Similarly, the loss for the train and test data dips rapidly until the 25th epoch, then gradually reduces until the 100th epoch and then begins to stabilize.

On the training the model for more number of epochs(500,1000,2000etc.), it was observed that the graph lines for both the accuracy and the loss for both the train and test data seemed to run parallel to the X-axis with no drastic variation(<5% increase in accuracy and <0.2 dip in loss) indicating that the model is not good enough to predict any better on training further. The below bar graphs depict this observation on the test data.



Question1_2_1: Network Architecture A

a. Layer 1: 300 neurons (ReLu activation functions).
b. Layer 2: Softmax Layer (from Question1_1)

1. **Forward_pass function**
    - In the forward_pass() function everything remains the same as in Question1_1 expect that an additional relu_layer() function is added and 2 coeefficients and 2 biases (one each for the 2 layers) are taken as input along with the features data.
    - Since the Relu layer is the first layer in the model, the first operation in the forward_pass function would be to implement the relu_layer() function.
    - The training features data, coefficients1 and bias1 is passed as input arguments to the relu_layer() function.
    - A similar x = (coefficients*features data) +bias is implemented to obtain the pre-activation output of the relu_layer using the same tensorflow low level functions as used in Question1_1.
    - Then, to compute the Relu actiavtion we use the formula which implies that all values that are negative, on actiavtion will become 0.0 and all values that are positive remain the same on activation:

      H1 = max(0.0, x)

      which is implemented using tensorflow functions as:

      tf.math.maximum(H,0.0)

    - Here, the shape of H would be the shape of the pre-actiavtion output which is (300,60000) since the number of neurons in the Relu layer was specified to be 300. On performing the Maximum function between H and 0.0, 0.0 will be broadcasted by te tf.math.max function to the same shape as H. Therefore, the output of the relu_layer would have the shape (300,60000) and would be fed to the next layer, that is the softmax layer which operates in the same manner as in Question1_1.

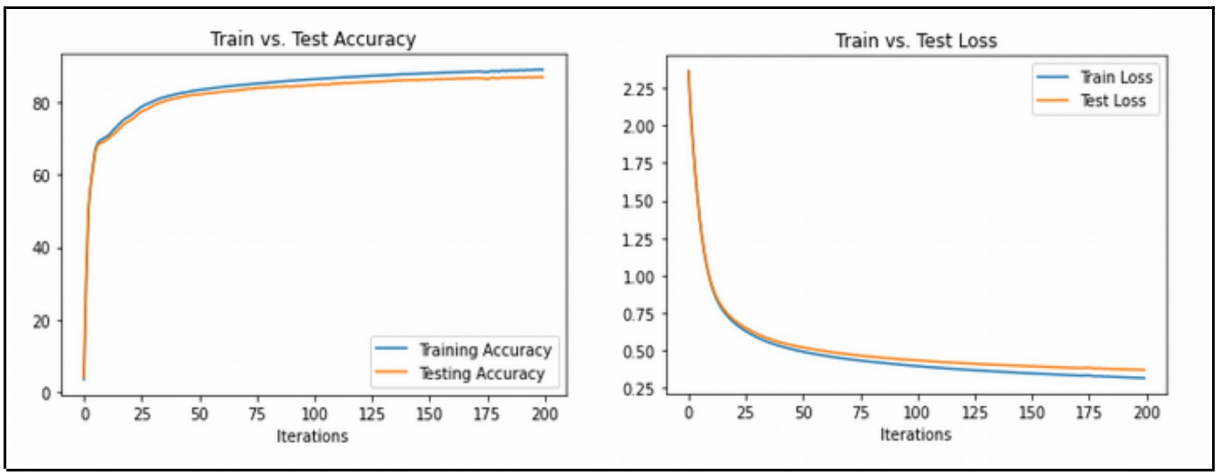
2. **Evaluation of the model**

On training the model with 200 iterations, the metrics obtained are as follows:

Data/Metrics	Accuracy	Loss
Train Data (60,000)	89.025%	0.31700405
Test Data (10,000)	86.93%	0.3722629

From the above table, it can be inferrred that model can perform really well with a test accuracy of 86.93% .

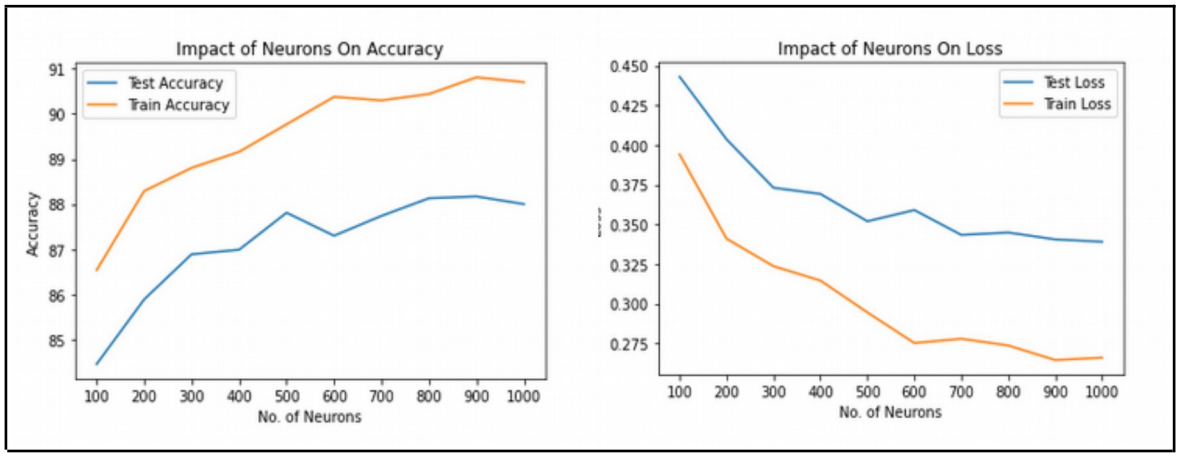
Below are the graphs depicting the accuracy and loss of the train and the test data over iterations.



From the above graphs, it can be observed that the accuracies for both the train and test data increase rapidly until the 10th epoch, then gradually increase until the 50th epoch and the begin to stabilize from there on. Similarly, the loss for the train and test data dips rapidly until the 10th epoch, then gradually reduces until the 50th epoch and then begins to stabilize.

3. Impact of the number of neurons in Layer1

An experiment was conducted on the model with increasing number of neurons starting from 100 to 1000. Below are the graphs depicting the impact of the increasing number of neurons in the Relu layer of the model when run for 200 iterations.



From the above graph, it can be observed that on increasing the number of neurons, the accuracy increases and loss decreases for the model intially. This is because the model is starts reading the minute details from the train data and trains the model such that every detail is captured. However, as the number of neurons increses from 500 onwards, both the accuracy and loss starts stabilizing and more importantly the gap between the train and the test curves starts to increase. This implies that the model has begun to overfit, that is, the model has got so accustomed to the fine details in the training data that it would no longer be able to perform well on the test(unknown) data. In other words the model looses it ability to be a generalized one. It cannot perform well on other datasets as the number of neurons in the layers increases. Choosing an optimum number of neurons for a layer in the model would be the best way to get a model to work most efficiently.

Question1_2_2: Network Architecture B

a. Layer 1: 300 neurons (ReLu activation functions).
b. Layer 2: 100 neurons (ReLu activation function)
c. Layer 3: Softmax Layer (from Question1_1)

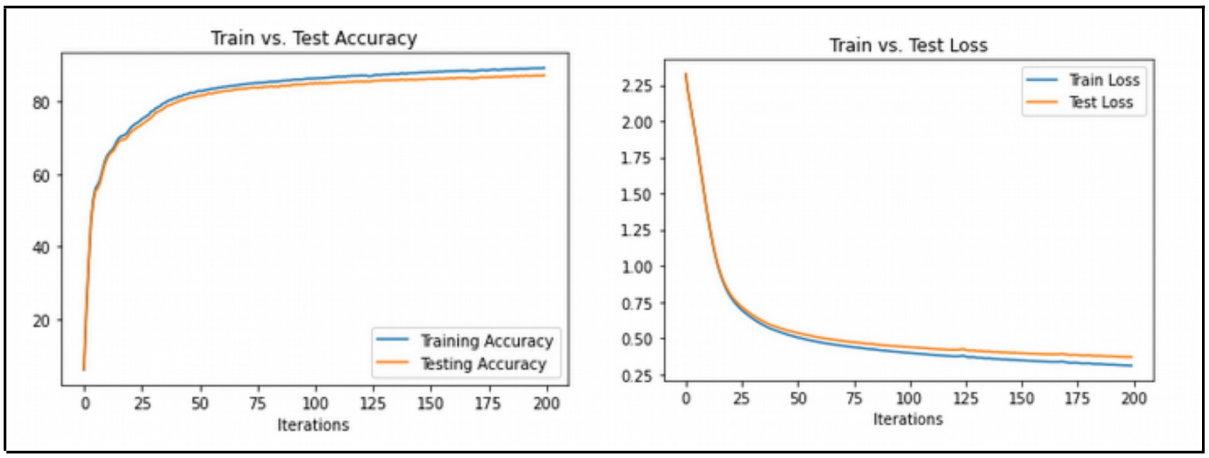
1. **Forward_pass function**
   - The forward_pass function is very similar to the one that was created in Question1_2_1 with the only difference that the forward_pass function would take 3 coefficients and 3 biases(one each for the 3 layers) as input along with the features data.
   - Also, the relu_layer ()function would be called twice, once for the first layer with 300 neurons and the second time for the second layer with 100 neurons.

2. **Evaluation of the model**
   On training the model with 200 iterations, the metrics obtained are as follows:

Data/Metrics	Accuracy	Loss
Train Data (60,000)	89.10667%	0.30891177
Test Data (10,000)	87.01%	0.36892897

From the above table, it can be inferrred that the model can perform really well with a test accuracy of 87.01% .

Below are the graphs depicting the accuracy and loss of the train and the test data over iterations.

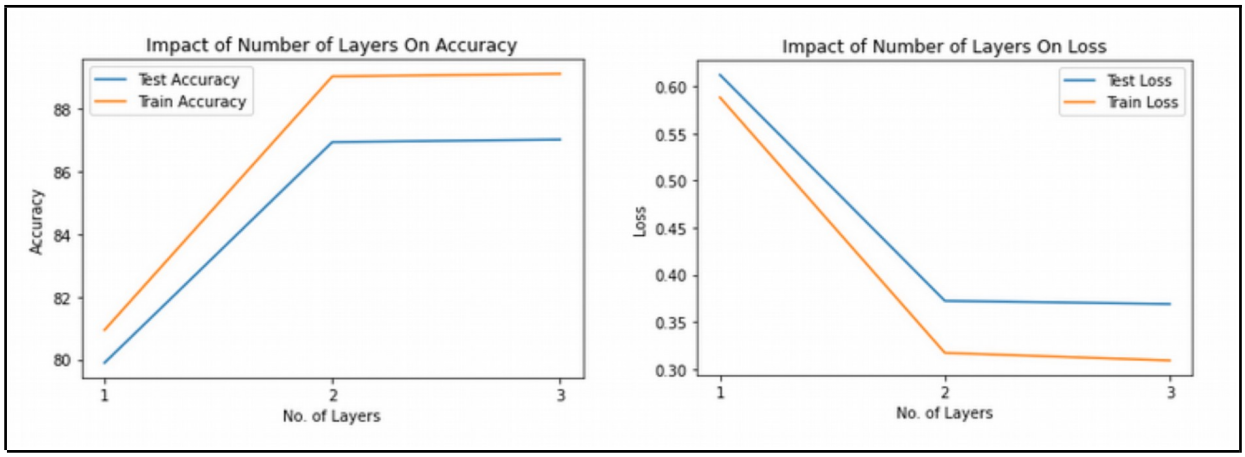


From the above graph, it can be observed that the accuracy and the loss for the train and test data has improved just a little in comparison with the Network Architecture A(Question1_2_1).

3. **Impact of the number of layers on the model**

The models that were created in the Question1_1, Question1_2_1 and Question1_2_2 have increasing layers.
- 1 Layer : Softmax Layer
- 2 Layers : Relu Layer(300 neurons) + Softmax Layer
- 3 Layers : Relu Layer(300 neurons) + Relu Layer(100 neurons) +Softmax Layer

The below graph depicts the performance of the model with increasing number of layers.



From the above graph, it can be observed that both the metrics(accuracy and loss) improve drastically on addition of the first Relu layer. However, an additional Relu layer did not significantly improve the model's performance. It is also evident from the graphs that with increasing layers, the gap between the train and test curve gradually increases. This hints about overfitting problem. These are the situations when a right decision needs to be made about the number of layers in the model. While the negatives(gradual overfitting) is unnoticeable/less significant when compared to the positives(increase in accuracy and dip in loss) , adding a layer would not do much harm as in the 2 layer model. However, adding another layer has not done much good to the accuacy or loss but has only created a further gap between the train and test curves as in the 3 layer model.

In conclusion, choosing the optimum number of neurons in the layers or choosing the right number of layers in the model should be based on the complexity of the data. Lesser neurons, lesser layers leads to poor performance of the model while too many neurons, too many layers leads to overfitting and a not so generalized model. Too much or too less is bad!

Question1_3 : L1 and L2 Regularization

- L1 and L2 regularization are applied to the model having Network Architecture B(Question1_2_2).
- Instead of the cross_entropy function, 2 functions named cross_entropy_l1_regularization() and cross_entropy_l2_regularization() are created.
- Regularization is performed in order to tackle the problem of overfitting. It adds a penalty factor to the loss function thereby decreasing the rate by which the vales of the coefficients increase.

1. **Cross_entropy_l1_regularization function**
   - The L1 regularization(Lasso) function tends to shrink the coefficients to 0 thereby leading to variable selection, that is, elimination of few coefficients(features) from the model training.
   - This function is very similar to the cross_entropy function from Question1_2_2 until the loss calculation part. We just add an extra regularization factor to the loss component.
   - L1 regularization is done by simply computing a summation of all the coefficient values for all layers, neurons and samples and multiplying it with a regularization parameter delta. The formula is:

L1 = delta*Σ(coefficients)

which is implemented using tensorflow functions as:

Reg =tf.multiply(delta,tf.math.add(tf.math.reduce_sum(coefficients),.....))

Here , the summation of the individual coefficients is computed using tf.math.reduce_sum, all coefficients are added using tf.math.add and then multiplied with delta using tf.math.multiply.

- This regularization factor is now added to the loss using tf.math.add. This sum is returned as the output of the function.

2. **Cross_entropy_l2_regularization function**
   - The L2 regularization(Ridge) shrinks the coefficients by the same proportion but ensures that none of them are eliminated.
   - This function is very similar to the cross_entropy function from Question1_2_2 until the loss calculation part. We just add an extra regularization factor to the loss component.
   - L1 regularization is done by computing a summation of all the squares of the coefficient values for all layers, neurons and samples and multiplying it with a regularization parameter delta. The formula is:

$$\textbf{L1 = delta*}\Sigma\textbf{(coefficients)}^2$$

which is implemented using tensorflow functions as:

Reg =tf.multiply(delta,tf.math.add(tf.math.reduce_sum(tf.math.pow(coefficients,2)),.....))

Here , the squares of the coefficients is computed using tf.math.pow(), summation of the individual coefficients is computed using tf.math.reduce_sum, all coefficients are added using tf.math.add and then multiplied with delta using tf.math.multiply.

- This regularization factor is now added to the loss using tf.math.add. This sum is returned as the output of the function.

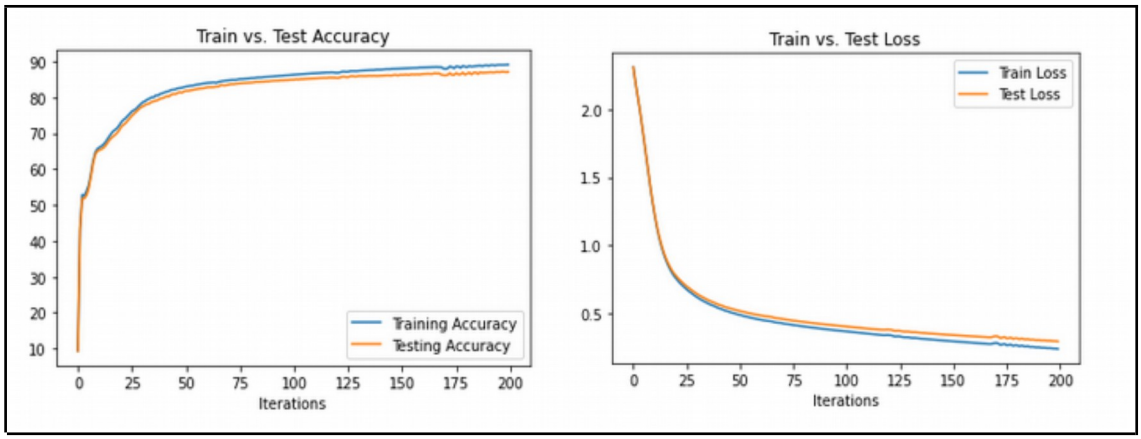
3. **Evaluation and Analysis of L1 and L2 Regularization Models**

On training the model with 200 iterations, the metrics obtained are as follows:

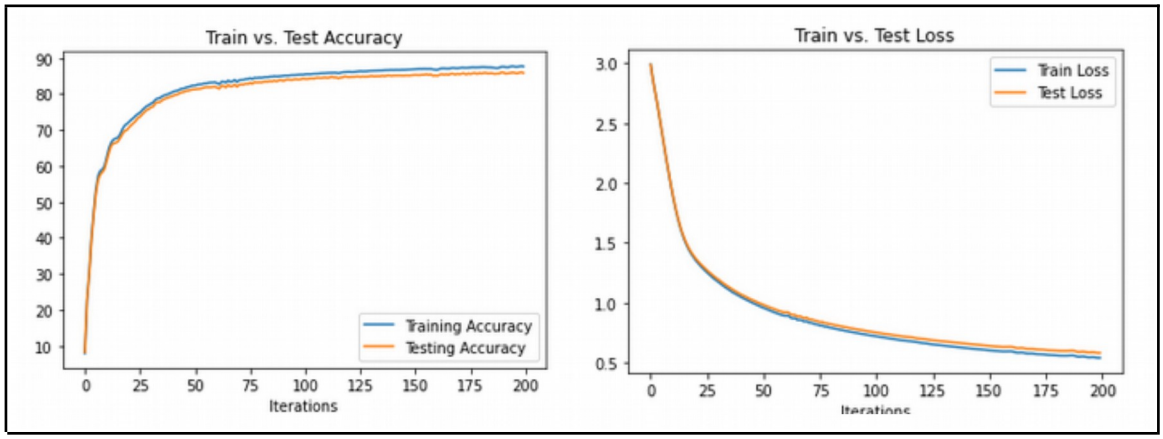
Data/Metrics	Accuracy	Loss
L1 Regularization (delta = 0.00001)		
Train Data (60,000)	89.096664%	0.23582304
Test Data (10,000)	87.07%	0.29142278
L2 Regularization (delta = 0.001)		
Train Data (60,000)	89.028336%	0.37476146
Test Data (10,000)	86.86%	0.43106246

From the above table, it can be inferrred that the L1 regularization model performs well(~87% accuracy) with delta = 0.00001 and the L2 regularization model performs almost with the same(~86.9% accuracy) with delta = 0.001.

Below are the graphs depicting the performance of the L1 Regularization model.

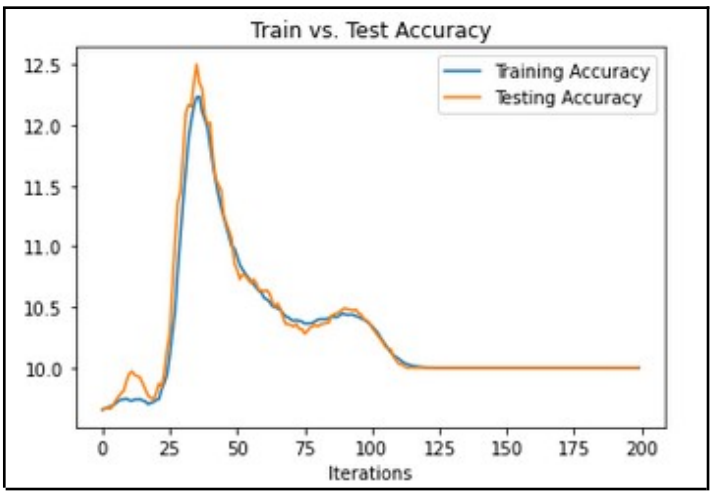


Below are the graphs depicting the performance of the L2 Regularization model.



It can be observed from the above graphs that the gap between the train and test curves are considerably smaller when compared to the previous models in Question1_1, Question1_2. This proves that regularization reduces overfitting of the model.

- If the delta values are made larger(less no. Of 0s after the decimal), the model tends to incur more loss, which means, the value of the coefficients start becoming larger and larger on gradient updates. This would lead to underperformance of the model.
- If the delta values are made smaller(more number of zeros after the decimal), then the model takes many iterations to perform in the same manner as it does with the correct delta value.
- For instance, the below graph depicts the accuracy of the L2 regularization model with delta= 1.0. This leads to severes impact on the accuracy of the model. Therefore, choosing the apt delta value is important.



In conclusion, the no. of layer, no. of neurons in layers, the loss calculation(with/without regularization) impact the performance of a deep learning model. With regularization, overfitting is reduced and the model performs fairly well on any data.

Part B : Keras – High Level API

- A valiation split of 0.2 is applied as it helps visualize the trend in accuracy and loss with increasing epochs.
- Optimizer = ‘adam’
- Loss = ‘sparse categorical crossentropy’
- Pre-activation using Dense layers
- All models are trained for 20 epochs with batch size 256

Question1 : Building Keras models

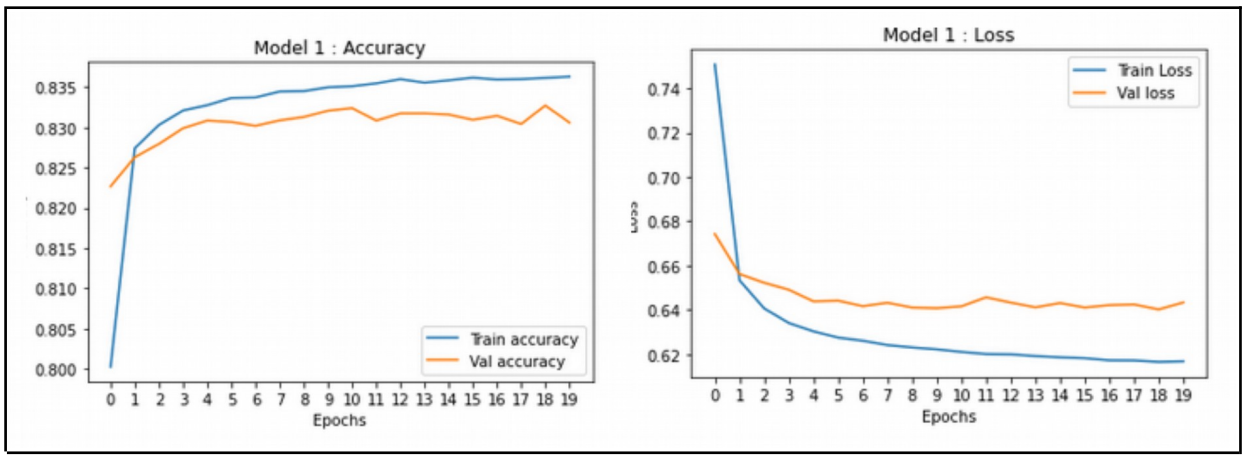
Code Flow

1. **Importing Data and Packages**
   - Data is imported to the notebook using the code provided in the assignment PDF and required packages are imported.

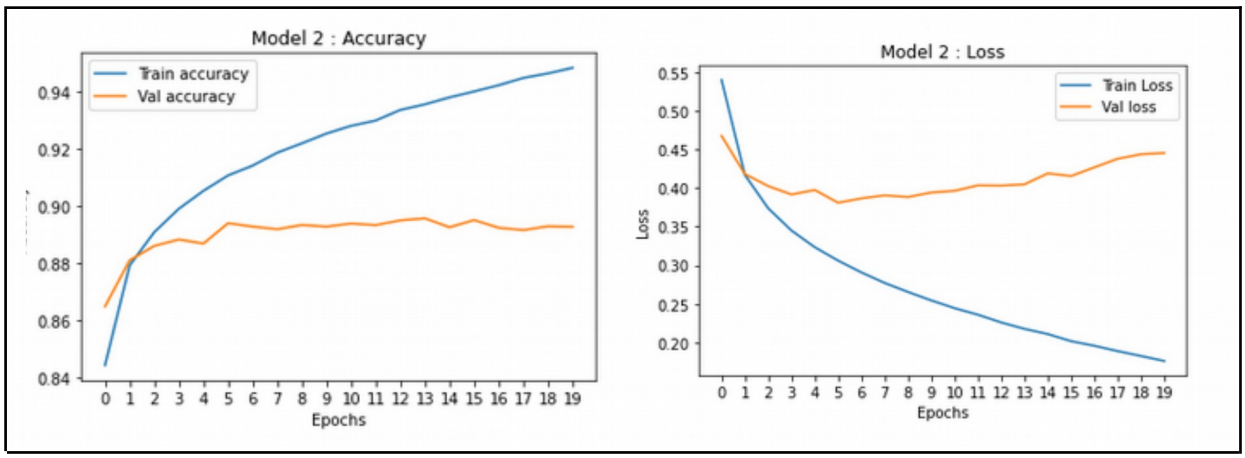
2. **Computing No. Of Classes in the Data**
   - The number of classes in the data needs to be computed as the number of neurons in the Softmax layer depends on the number of classes. It is computed by computing the length of the first row of the to_categorical() function of the labels data.

3. **Model 1 - 1 layer - softmax layer**

   Test Accuracy = 85.5%
   Test Loss = 0.55
   Below are the graphs depicting the accuracy and loss for this model. It can be observed that the validation data underperforms as and the model would begin to overfit after the 20th epoch. This model only does softmax classification and with that in mind, I think it is a fair performance.



4. **Model2 - 2 layers - relu layer, softmax layer**
   Test Accuracy = 91.25%
   Test Loss = 0.35
   The below graphs depict the accuracy and loss of model 2.

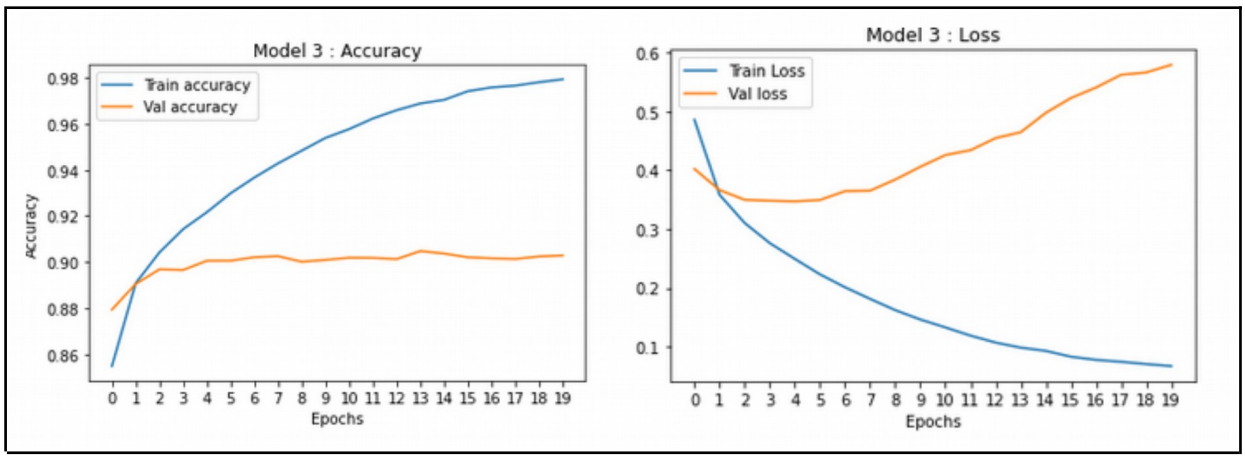


It can be inferred from the graphs that adding the relu layer has improved the test accuracy of the model by 5%. However, the cost paid for the 5% increase in accuracy is a severe case of overfitting. The gap between the train and test curves is increasing rapidly with increasing epochs.

5. **Model3 - 3 layers - relu layer1, relu layer2, softmax layer**

   Test Accuracy = 92.01%
   Test Loss = 0.46

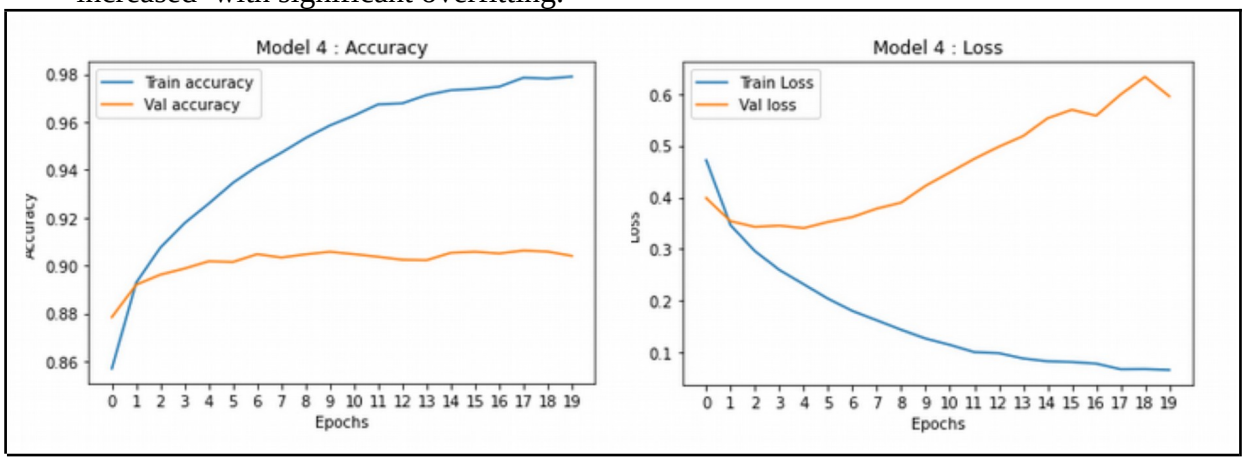
   The below graphs depicts the performance of model3. It can be observed that the accuracy does not improve significantly, the loss increases in comparison with the previous model. And overfitting takes a new dimension in this case. The gap has severely increased in comparison with model2.



6. Model4 - 4 layers - relu layer1, relu layer2, relu layer3, softmax layer

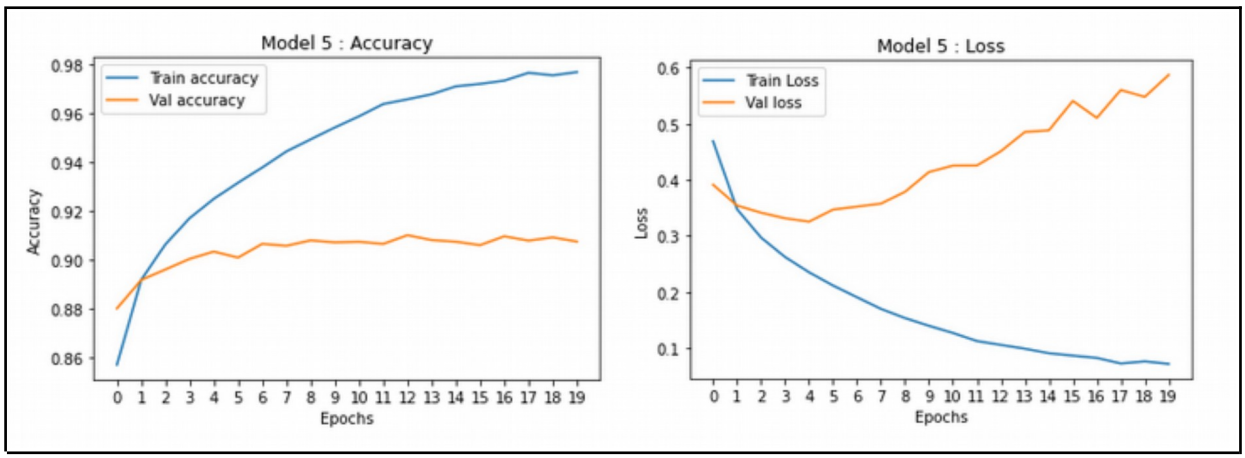
Test Accuracy = 92.3%
Test Loss = 0.47

The abve graphs depict the performance of model4. It can be observed that there is not significant improvement in the accuracy of the validation data and the loss has also increased with significant overfitting.



7. Model5 - 5 layers - relu layer1, relu layer2, relu layer3, relu layer4, softmax layer

Test Accuracy = 92.6%
Test Loss = 0.45

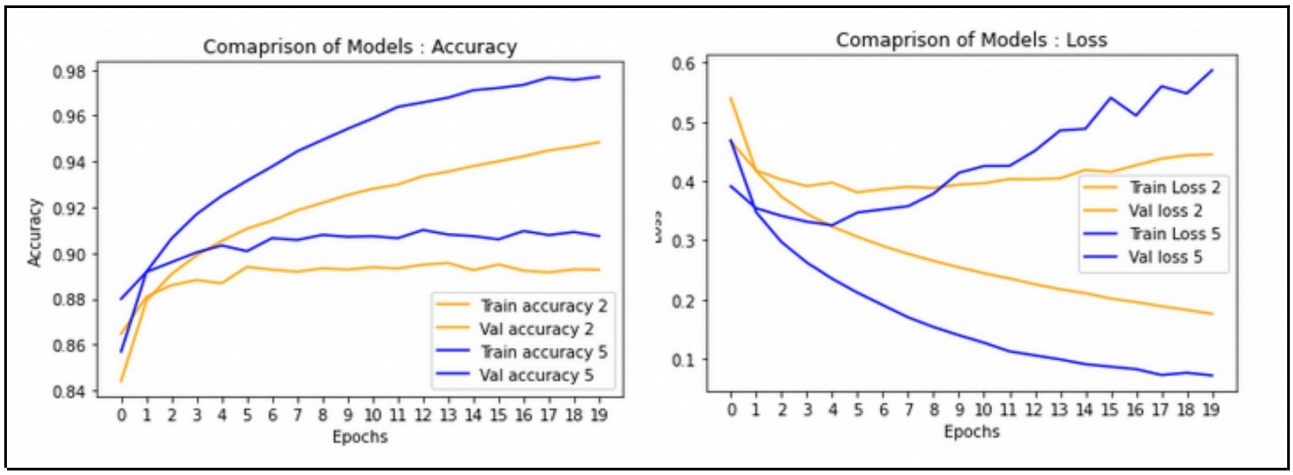


The above graphs depict the performance of model5. There is few less imrpovement in the test accuracy and the validation curves seem to completely diverge from the train curves which again shows that the model is absolutely overfitted and cannot perform well with unknown data.

Comparative Analysis of the 5 models

It can be observed that model 2 (1 relu and 1 softmax layer) performed the best amongst the 5 models with a test accuracy of 90%. Althugh other models gave a betetr accuracy model 2 is chosen to have performed better than them because as the number of Relu layers increased in the models, there was no significant improvement in the test accuracy and overfitting became larger and larger.

The below graphs represent the comparison between two model : model2 and model5.
The blue curves depict the performance of the train and validation data of model5 whereas the orange lines depict the performance of the train and validation data of model2.



It can be observed from the above graphs that although model5 give a slightly better accuracy, the gap between the train and validation curve is larger in the accuracy graph. This phenomenon can be observed more evidently in the loss graph where the two curevs for model5 diverge way most than acceptable measures.

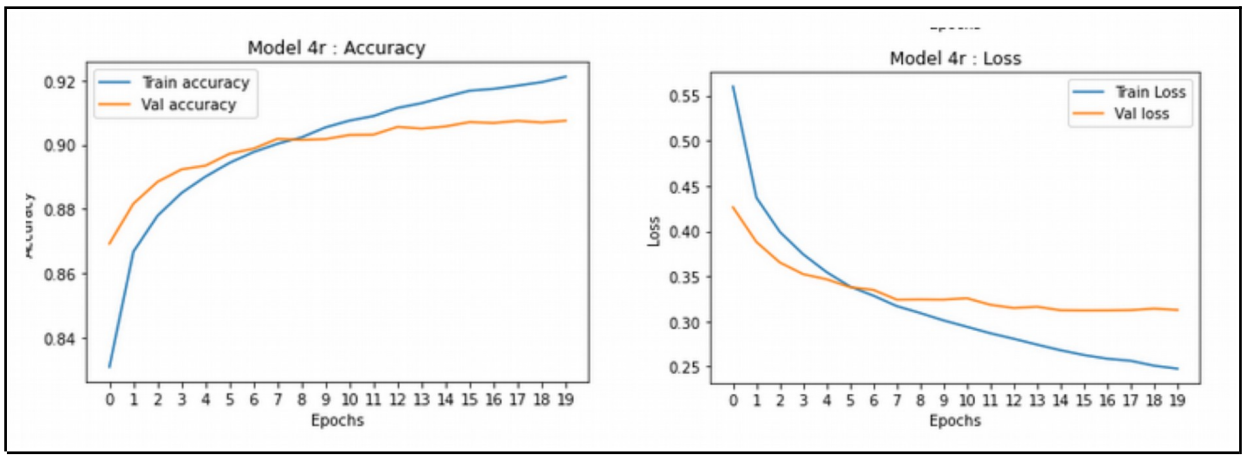
Question2 : Building Keras models

-Dropout layers(regularization) are added after all the Relu layers in order to reduce overfitting.

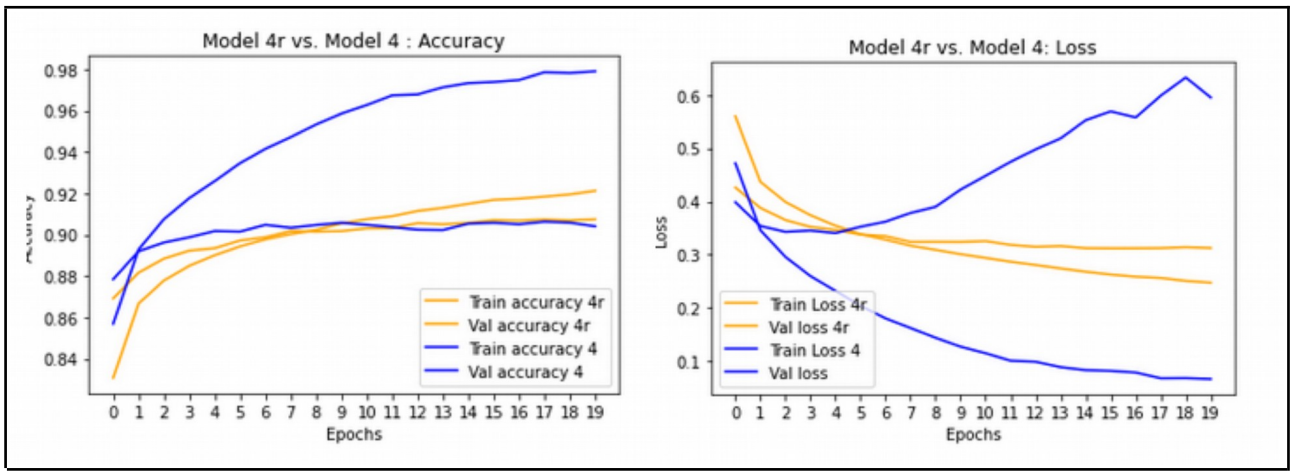
1. **Model4r – Model 4 along with dropout layers**

   Dropout factor = 0.3 after every Relu layer
   Test Accuracy = 92.6%
   Test Loss = 0.24

   Below graphs represent the performance of model4r. It can be observed that the accuracy is very good, loss is acceptable.



Now, a comparison between model4 and model4r will clearly demonstrate how regularization reduces overfitting. The below graphs represent the performance of model4 and model4r. The orange curves depict the performance of the train and validation data of model4r while the blue lines depict the train and validation performances of model4.

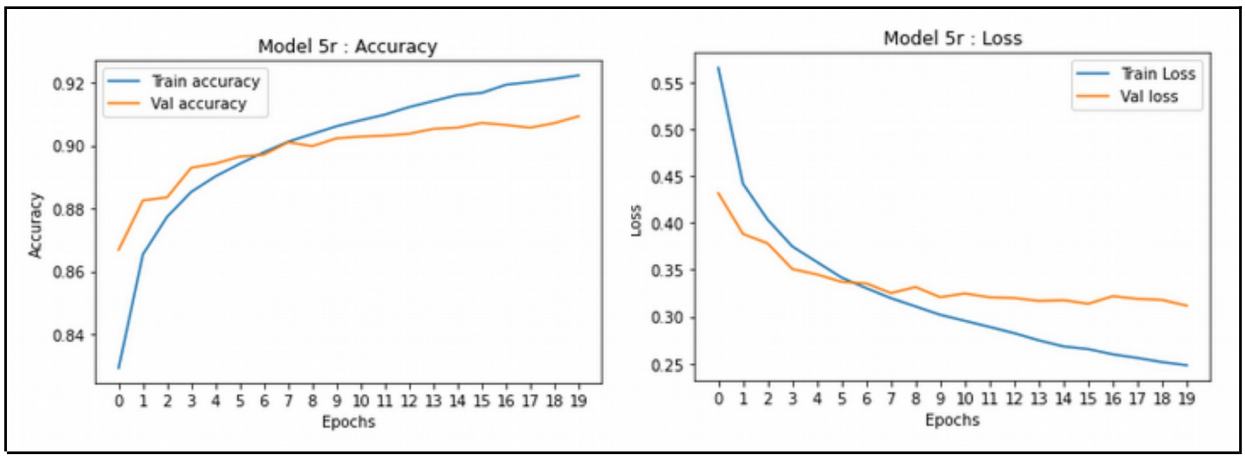


It is so evident from the graphs that the train and validation curves do not diverge as much for model4r as it does for model4 which does not have dropout layers(regularization). This proves that regularization reduces overfitting of the data while giving a good accuracy for the model.

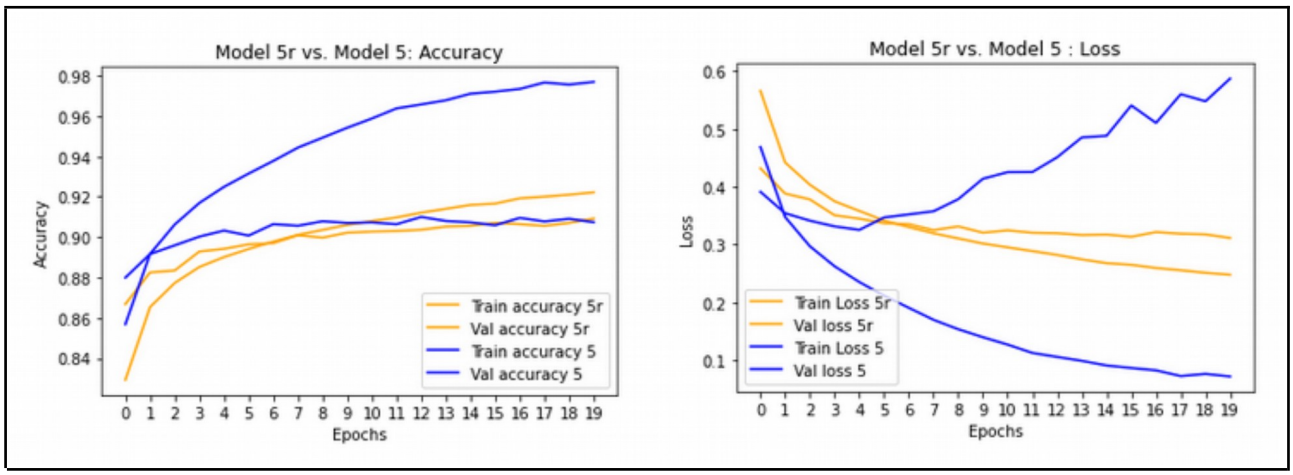
2. **Model5r – Model 5 along with dropout layers**

   Dropout factor = 0.3 after every Relu layer
   Test Accuracy = 92.8%
   Test Loss = 0.24

   Below graphs represent the performance of model4r. It can be observed that the accuracy is very good, loss is acceptable.



Now, a comparison between model5 and model5r will again clearly demonstrate how regularization reduces overfitting. The below graphs represent the performance of model5 and model5r. The orange curves depict the performance of the train and validation data of model5r while the blue lines depict the train and validation performances of model5.



It can be observed from the above graphs that the gap between the curves has further reduced indicating a good progress in tackling overfitting.

In conclusion, overfitting can be reduced by including regularization mechanisms in the model.

Part C : Research on Batch Normalization

What problem does it address?

The basic operation of deep learning neural networks involves computing weights and biases for all the neurons in the network for the data sample in the forward direction, and then updating these weights and biases in a backward fashion layer-by-layer based on the loss incurred by the model. The updation of all the weights in the model typically happens simutaneously which means that once the loss is calculated, the entire model's weights are updated at one go. However, this updation poses a problem. Since all layers are connected sequentially to each other, when a layer A is updated with the new weights, and then the model goes on to update the weights of the layer previous to the current layer (layer A-1) during the backward propagation, the updation in current layer A is no more relevant to the previous layer's update. Instead it would be more in synchronization with the previous values of weights of the layer A-1. This is because layer A was trained to expect/receive a

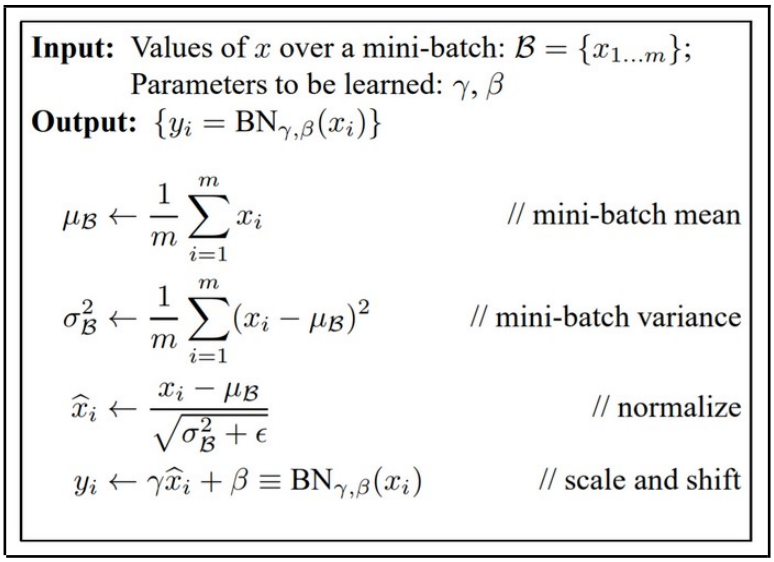
certain distribution of values from Layer A-1 and now this new update would not necessarily deliver the same distribution of values to layer A after the new updation of weights. This phenomenon is known as Internal Covariate Shift.

The aforementioned problem may occur due to poor initialization of weights or due to extreme variations in mean and standard deviations of features data. In order to tackle this problem, batch normalization was introduced to deep learning models.

How does it work and how does it overcome the problem??

Batch Normalization is used to ensure co-ordination during the simultaneous updation of weights in the network . It is like a pre-processig step in every layer of the model. It is done by standardizing the inputs to not only the first layer of the network, but also to all the subsequent layers in between. It means that for every mini batch, the output after the activation of every layer in a neural network will be standardized with mean = 0 and standard deviation = 1(guassian distribution) and this normalized activations will be sent as input to the subsequent layer during the training process. This ensures that the assumptions that layer A makes about the spread and distribution of data from layer A-1 during the weights update does not significantly change. If the input to a layer is normalized, even on updation of weights it will more or less receive similar input data[5].

Algorithm for Batch Normalization:

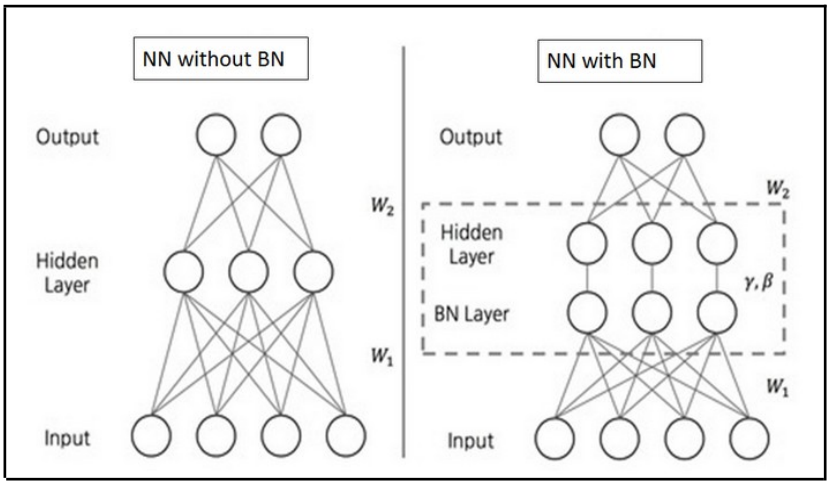


The above image depicts the algorithm of the Batch normalization problem from the original paper[2].

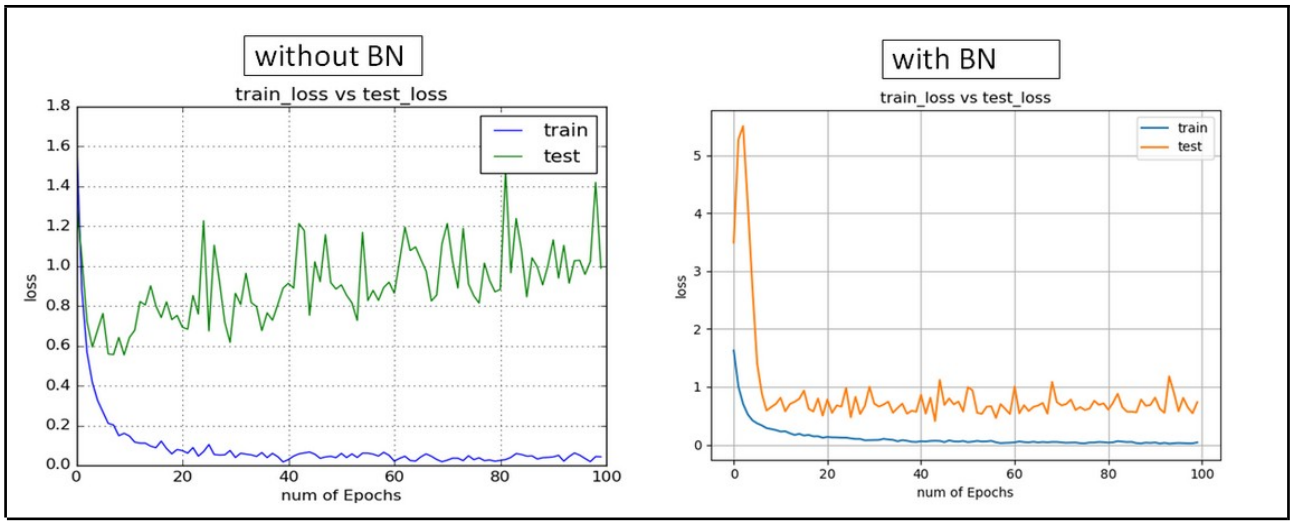
- Since batch normalization happens in every layer of the network, the input to the batch_normalization layer would be the activation outputs of the previous layer, that is, the values of x over a mini-batch.
- The parameters to be learned by the model during batch normalization are beta(mean) and gamma(standard deviation) of x (values in the mini-batch).
- The output of the batch normalization layer would be y , the normalized values of x with mean = beta and standard deviation as gamma.
- The mean of the x activation outputs is calculated as beta.
- The standard deviation of the x activation outputs is calculated as gamma.
- The x activations are normalized by subtracting beta from each term in x and dividing the term by gamma to obtain y. The epsilon values are added to avoid divide by zero error.
- Y is sent as the input to the next layer for further training.

By applying this algorithm to the training data between every layer, the next layer can thrive on the assumption of the spread and distribution of what activation outputs it would get from the previous layer even after updation of weights.

The below image shows a network with and without batch normalization.



The below graphs represent the loss incurred by a deep learning network with and without Batch Normalization. It can be observed from the graphs that, with Batch Normalization, the loss during testing is reduced significantly. Also this happens during the initial epochs itself. Therefore, batch normalization does have a positive impact on the performance of the model.



Advantages of Batch Normalization

Batch normalization does not only address the above mentioned problem, but it also improves the training of the model by reducing the number of epochs as all the data is normalized and eases the work of the optimizer as convergence happens at a faster rate. It also enables usage of larger learning rates as the data is normalized which makes the model more stable during training. In turn, the model will learn faster.

References :

1. Lecture notes and lab codes by Dr. Ted Scully
2. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift from https://arxiv.org
3. https://towardsdatascience.com/implementing-batch-normalization-in-python-a044b0369567
4. https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
5. https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/
6. https://medium.com/@anuj_shah/breaking-the-ice-with-batch-normalization-bbc41dab8403