R00182510 – Deep Learning Assignment Report

PART A - Tensorflow and Low-Level API

The dataset used for this part is the Fashion MNIST dataset containing 60000 training instances and 10000 test instances. The total number of classes is 10.

Task 1

The Objective is to build a neural network with a SoftMax layer for the Fashion MNIST dataset. The SoftMax layer will contain 10 neurons indicating the number of classes present in the input dataset. The neurons in the SoftMax layer will output the probability of the input image belonging to a specific class. The three main functions related to building the model is described below.

1) **Forward Pass**

   The code for this function is given below

```
def forward_pass(x, w_T, b):

    # Multiply each training example by the weights and add the bias
    y_pred = tf.matmul(w_T, x) + b

    # Pipe the results through the Softmax activation in the following
      steps
    t = tf.math.exp(y_pred)
    t_sum = tf.reduce_sum(t, 0)
    # Reshape the data to apply the normalization to t
    t_sum = tf.reshape(t_sum,[1, -1])

    #output of the Softmax is t normalized by dividing by the sum of t.
    y_pred_softmax = tf.divide(t, t_sum)

    return y_pred_softmax
```

   This function is used to push all the training instances through the SoftMax layer by multiplying each training example by the weights and the bias associated with each neuron. The resulting tensor y_pred is then passed through the Softmax activation function which involves calculating the exponential of y_pred and then normalizing its values by dividing by the column sum of the tensor y_pred. Each column in y_pred indicate the output of the 10 activation units/neurons for each training instance. The resulting tensor y_pred_softmax is similar in shape of y_pred and will now contain the probabilities of the input image/training instance belonging to the 10 classes.

2) **Cross Entropy Loss**

   The code for this function is given below

```
def cross_entropy(y, y_pred):
    # Calculate the cross entropy error for all training data
    cross_entropy_loss = -(tf.reduce_sum(tf.multiply(y,
                                         tf.math.log(y_pred)), 0))

    # Calculate cost which is the mean cross entropy error/ average loss
```

```
      across all training instances
    cost = tf.reduce_mean(cross_entropy_loss)

    # cost = tf.reduce_mean(cross_entropy_loss)

    return cost
```

This function is used to calculate the loss from the SoftMax function. The loss for each training instance is calculated as the negative log of the outputted SoftMax value multiplied by the true class labels. Since the class labels are one hot encoded, only one value within the column vector of the true class labels will have a value of 1. Thus, cross_entropy_loss is a tensor that will contain a vector of values indicating the loss corresponding to each training instance. The final cost function is calculated as the average loss across all training instances.

3) Calculate Accuracy

The code for this function is given below

```
def calculate_accuracy(y, y_pred_softmax):
    # Calculate the predictions in the form of a boolean array, by
      considering only the class with the highest probability
    # 1 if True (correct prediction), 0 if False (incorrect prediction)
    predictions_bool = tf.equal(tf.argmax(y_pred_softmax, 0), tf.argmax(y,
                                                                        0))
    predictions_correct = tf.cast(predictions_bool, tf.float32)

    # Finally, we just determine the mean value of the correct predictions
    accuracy = tf.reduce_mean(predictions_correct)

    return accuracy
```

The predicted SoftMax output contains the probabilities of an instance belonging to the 10 classes. In order to calculate the prediction accuracy, the class with the highest probability for each instance is compared with the true class labels. The boolean array predictions_bool will contain a value 1 if the instance has been predicted correctly and 0 if it is an incorrect prediction. The final accuracy is calculated as the mean of the correct predictions.

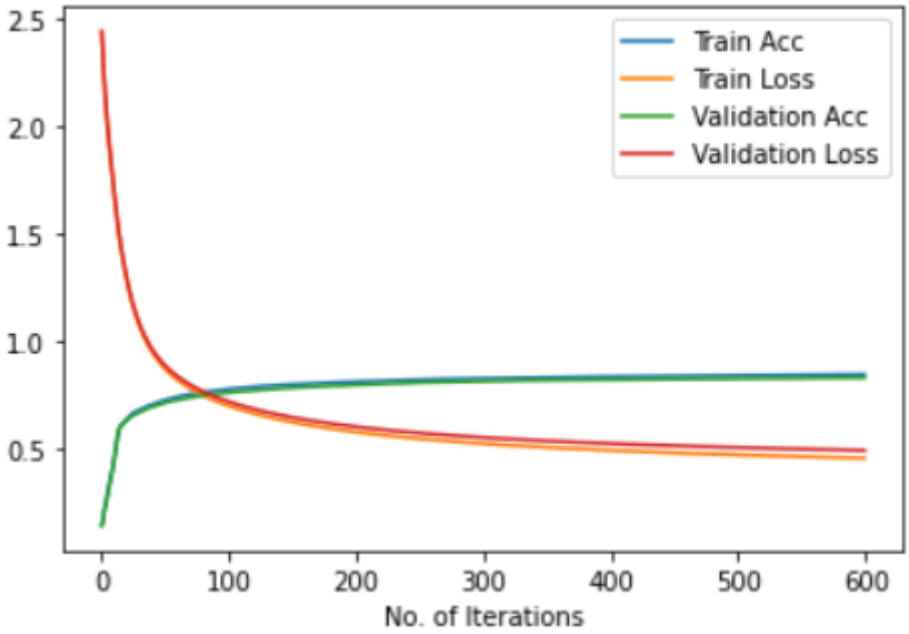
Results and Evaluation

The Training and validation accuracy and loss is given below.

Train Acc: 0.8471, Train Loss = 0.4621571
Validation Acc: 0.83, Validation Loss = 0.49859193

The plotted figure below shows how the loss and accuracy values vary with respect to the number of Iterations. Both the accuracies almost plateaus after 200 iterations with over 80% accuracy. After which it very slowly increases for the number of Iterations.

The train and validation loss steadily decrease until 150 Iterations and follows a very similar pattern, after which there is a minute gap that follows through the rest of the iterations. However, the gap is not large enough to see an overfitting model. The gap can be considered as the generalization gap. However, there is room for improvement with respect to increasing the accuracy.



Overall, the model built with just a single SoftMax layer has shown pretty good results for the Fashion MNIST dataset. An attempt to improve the model is studied in the rest of the document.

Task 2

In this task, we introduce additional ReLu based neurons before the SoftMax layer to study a more complex neural network model. The models studied are discussed below.

Network Architecture A

a. Layer 1: 300 neurons (ReLu activation functions).
b. Layer 2: SoftMax layer (from Task 1)

With respect to the code, the only updates are done in Forward Pass function. The code is shown below.

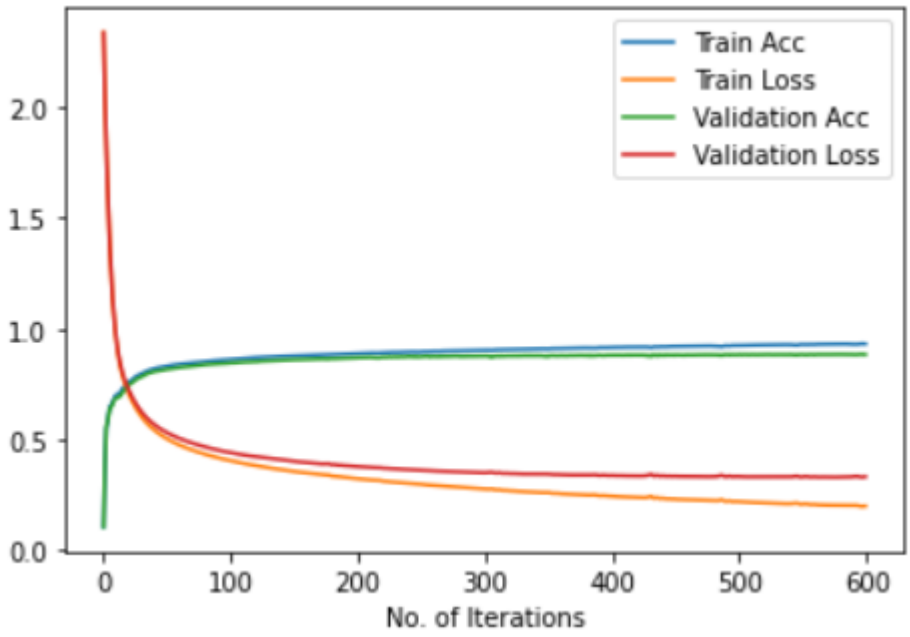
```
def forward_pass(x, w_T1, b1, w_T2, b2):
    # Multiply each training example by the weights and add the bias for the 1st
      hidden layer
    A1 = tf.matmul(w_T1, x) + b1
    # ReLu activation
    H1 = tf.nn.relu(A1)

    # Multiply each training example by the weights and add the bias for the
      SoftMax layer
    A2 = tf.matmul(w_T2, H1) + b2
    # SoftMax activation
    t = tf.math.exp(A2)
    t_sum = tf.reduce_sum(t, 0)
```

```
    t_sum = tf.reshape(t_sum, [1, -1])

    y_pred_softmax = tf.divide(t, t_sum)

    return y_pred_softmax
```

As shown above, just before the SoftMax layer, the pre-activation and ReLu activation (A1, H1 respectively) for the hidden layer is implemented.

The results are shown below

Train Acc: 0.9320667, Train Loss = 0.19903933
Validation Acc: 0.884, Validation Loss = 0.33143488



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 Task 1)

The forward pass function for this architecture is given below.

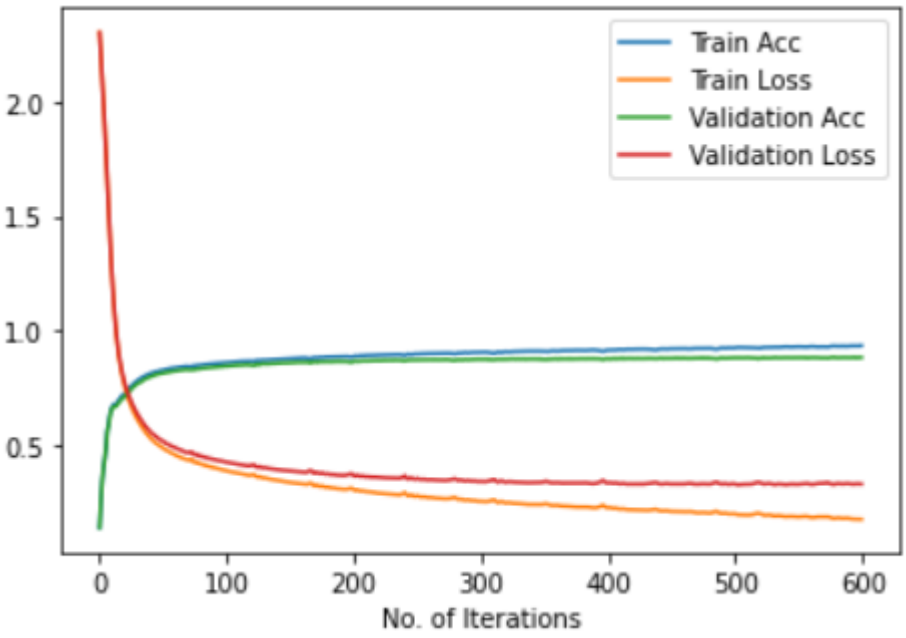
```
def forward_pass(x, w_T1, b1, w_T2, b2, w_T3, b3):
    # Multiply each training example by the weights and add the bias for the 1st
hidden layer
    A1 = tf.matmul(w_T1, x) + b1
    H1 = tf.nn.relu(A1)

    # Multiply each training example by the weights and add the bias for the 2nd
hidden layer
```

```
    A2 = tf.matmul(w_T2, H1) + b2
    H2 = tf.nn.relu(A2)

    # SoftMax activation
    A3 = tf.matmul(w_T3, H2) + b3
    t = tf.math.exp(A3)
    t_sum = tf.reduce_sum(t, 0)
    t_sum = tf.reshape(t_sum, [1, -1])

    y_pred_softmax = tf.divide(t, t_sum)

    return y_pred_softmax
```

As shown above, just before the SoftMax layer, the pre-activation and ReLu activations (A1, H1, A2, H2 respectively) for the hidden layers is implemented.

The results are shown below

Train Acc: 0.9374833, Train Loss = 0.18114457
Validation Acc: 0.8853, Validation Loss = 0.33479336



Evaluation from the above Results

The accuracies of both the architectures, follows a very similar pattern with a reported training accuracy of 93% and a validation accuracy of 88% approximately. Although the reported accuracies are higher than what was observed on Task 1, the interesting observation is more towards the reported loss.

In both the architectures the model tends to overfit after 150 epochs. The addition of hidden layers might be causing the models to overfit. The loss observed from the architecture B is slightly noisier than the architecture A, which shows the model might be sensitive to the neurons in each layer.

Another interesting observation when compared to the Task 1 and the Task 2 models are with respect to the training time. The training time for Task1 was a lot more that the training time for the Task2 architectures. This might be due to the addition of hidden layers that enables the model to learn faster.

Although the models in Task 2 report a higher accuracy, overfitting is observed which can be corrected by regularization techniques that is studied discussed in the following task.

Task 3

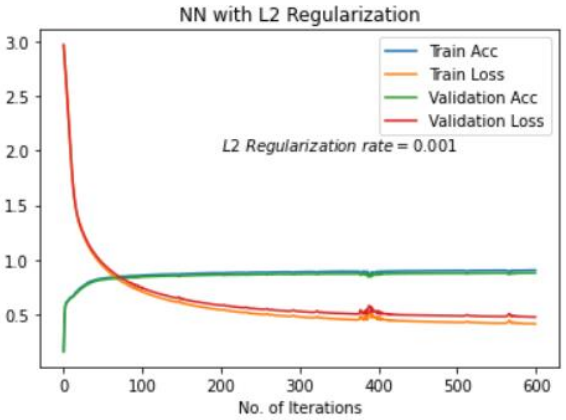
The L1 and L2 regularization techniques were applied to the Network architecture B from Task 2.

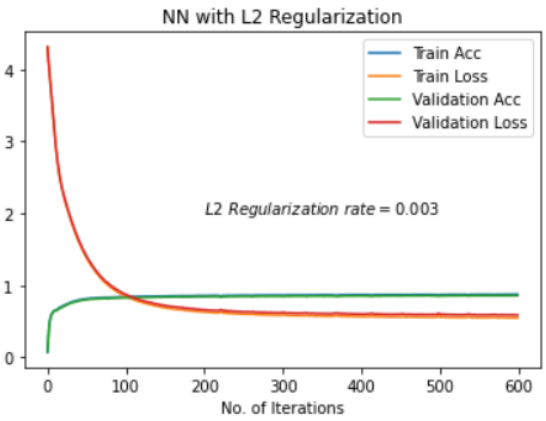
L2 Regularization

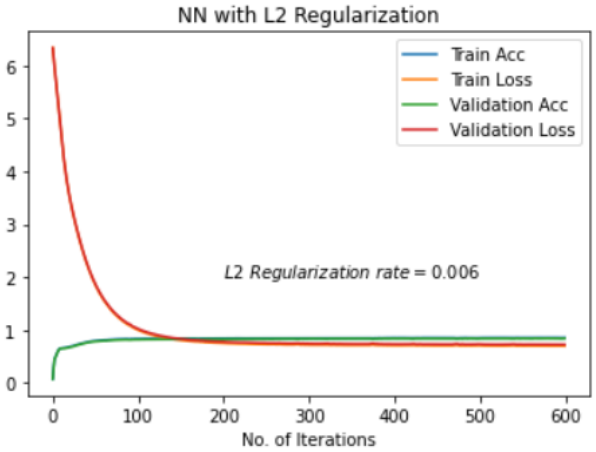
The L2 regularization cost was calculated by summing the squares of all the weights multiplied by the regularization rate. The code changes are given below.

```
def cross_entropy_L2Reg(y, y_pred, w1, w2, w3, alpha):
    # Calculate the cross entropy error for all training data
    cross_entropy = -(tf.reduce_sum(tf.multiply(y, tf.math.log(y_pred)), 0))

    # Calculate the L2 regularization cost
    regularization_cost_L2 = alpha * ((tf.reduce_sum(tf.math.square(w1))) +
                                      (tf.reduce_sum(tf.math.square(w2))) +
                                     (tf.reduce_sum(tf.math.square(w3))))

    # Calculate cost which is the mean cross entropy error/ average loss across
      all training instances along with the L2 regularization cost
    loss = tf.reduce_mean(cross_entropy) + regularization_cost_L2

    return loss
```

As we know, the more we increase the regularization rate, the more the shrinkage of the weights. Three different experiments with regularization rates at 0.001, 0.003, 0.006 was studied. The plots below show the results from these experiments







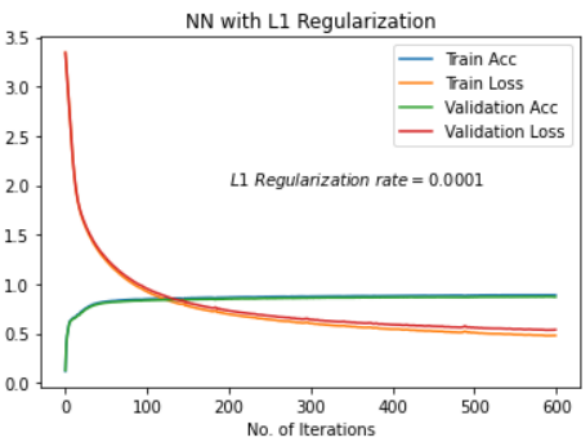
Although the accuracy is high, a lower regularization rate of 0.001 is actually causing the model to slightly overfit when compared to the other two models. On the other hand, a higher rate of 0.006 produces a good model, however the validation accuracy plateaus at approximately 83%. The ideal model with a best fit and good accuracy was determined by setting the regularization rat t 0.003. The reported training and validation accuracy were 87% and 86% respectively.

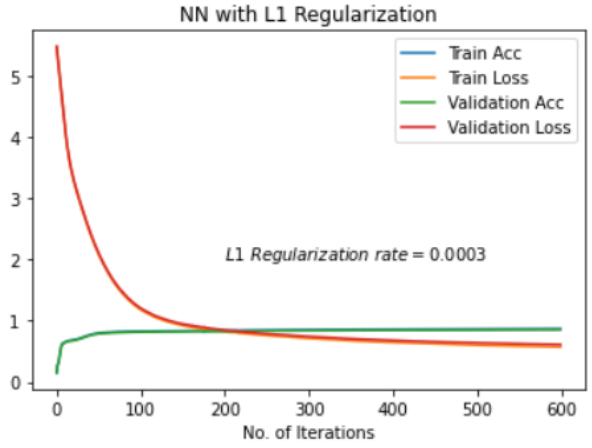
L1 Regularization

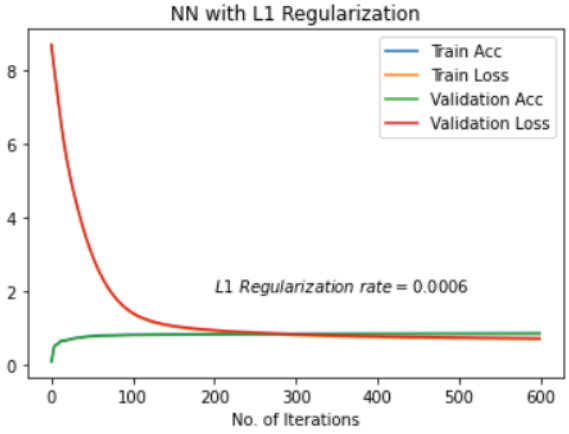
The L1 regularization cost was calculated by summing the absolute value of all the weights multiplied by the L1 regularization rate. The code changes are given below.

```
def cross_entropy_L1Reg(y, y_pred, w1, w2, w3, alpha):
    # Calculate the cross entropy error for all training data
    cross_entropy = -(tf.reduce_sum(tf.multiply(y, tf.math.log(y_pred)), 0))

    # Calculate the L1 regularization cost
    regularization_cost_L1 = alpha * (tf.reduce_sum(tf.abs(w1)) +
                                      tf.reduce_sum(tf.abs(w2)) +
                                      tf.reduce_sum(tf.abs(w3)))

    # Calculate cost which is the mean cross entropy error/ average loss across
      all training instances along with the L1 regularization cost
    loss = tf.reduce_mean(cross_entropy) + regularization_cost_L1

    return loss
```

Three different experiments with regularization rates at 0.0001, 0.0003, 0.0006 was studied. The plots below show the results from these experiments.







Like the pattern of results obtained for the L2 regularization, the ideal fit was observed at a regularization rate of 0.0003 with reported training and test accuracy of 87% and 85% respectively. The model with a L1 regularization rate of 0.0001 slightly overfits but it is not very significant and the model with L1 regularization rate of 0.0006 produces a good model, however the accuracy is better in the other model.

Overall, both L1 an L2 regularization techniques has added value to the model developed in Task 2. The above results very clearly show that regularization reduces overfitting in a complex model with the appropriate regularization rate.

PART B - Keras High-Level API

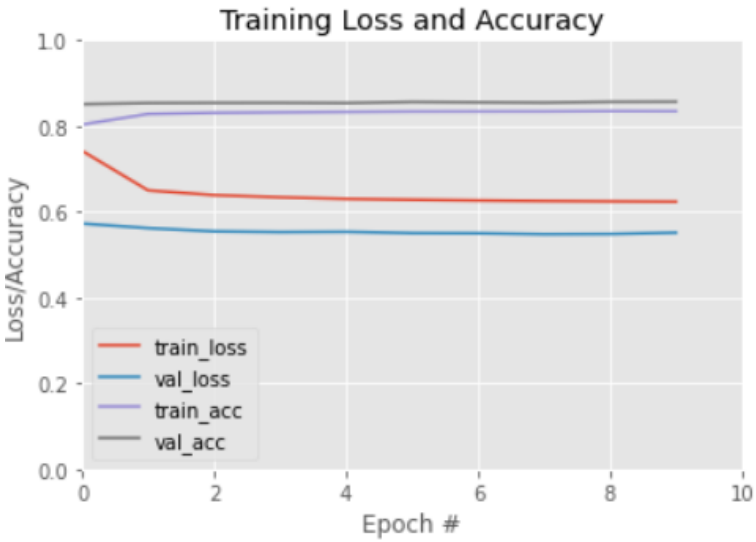
The dataset used for this task consists of images of letters from A-J. In total there are 200000 training images and 17000 test images. It has a total of 10 classes.

Task 1

Model 1:

Using Keras, an initial model was built using just the SoftMax layer. The results from this model after 10 epochs is shown below

Train loss: 0.6235 Train accuracy: 0.8346
Validation loss: 0.5514 Validation accuracy: 0.8569

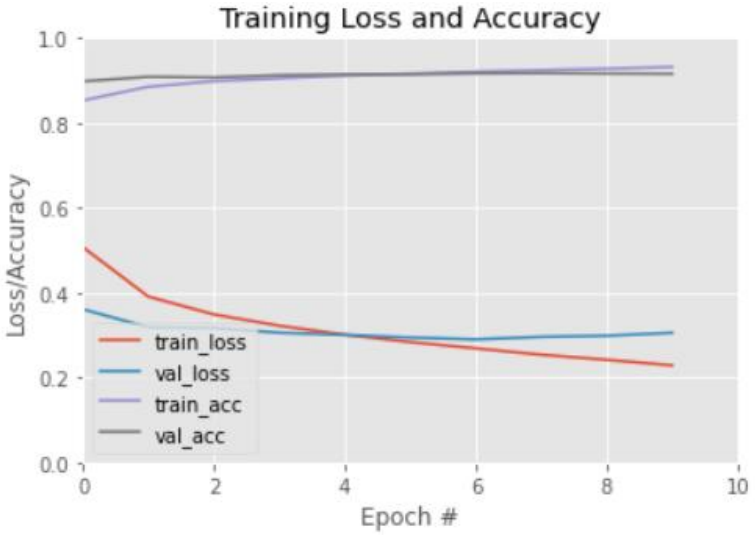


The model is significantly overfitting and both the accuracy and loss values have plateaued 1 epoch and the loss values are higher than expected. This model will possibly never converge. In an attempt to improve over this, the below three models will be implemented and analysed.

Model2:

We update the above model to contain one hidden layer containing 300 neurons just before the SoftMax layer. The results from this model after 10 epochs is shown below.

Train loss: 0.2288 Train accuracy: 0.9321
Validation loss: 0.3055 Validation accuracy: 0.9162

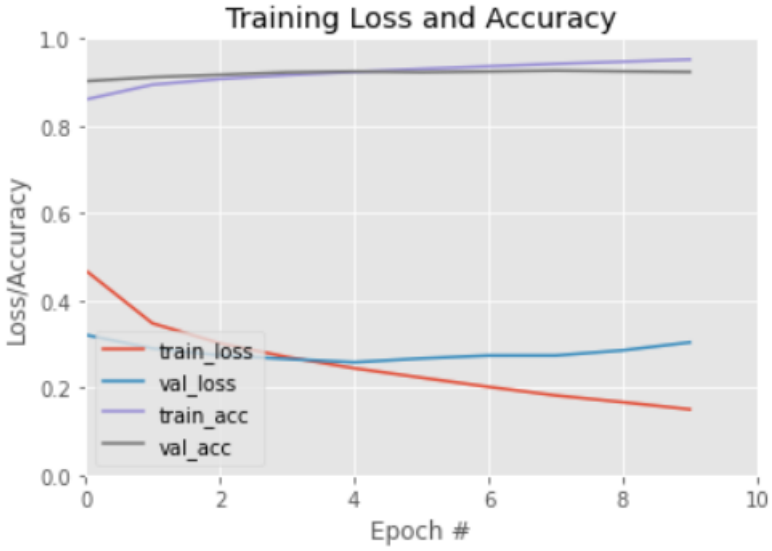


This model has shown an improvement in terms of accuracy and the loss converges after 4 epochs. However, after 6 epochs we can see that model is starting to overfit.

Model 3:

We update the above model to contain 2 hidden layers containing 400 and 200 neurons respectively just before the SoftMax layer. The results from this model after 10 epochs is shown below.

Train loss: 0.1504 Train accuracy: 0.9523
Validation loss: 0.3039 Validation accuracy: 0.9238



Model 4:

We update the above model to contain 3 hidden layers containing 600, 400 and 200 neurons respectively before the SoftMax layer. The results from this model after 10 epochs is shown below.

Train loss: 0.1331 **Train accuracy: 0.9566**
Validation loss: 0.3154 **Validation accuracy: 0.9303**

Comparative study of Models 2, 3 and 4:

Model2 has reported an accuracy of 91% and the loss converges after 4 epochs. However, after 6 epochs we can see that model is starting to overfit.

In Model3 we can see a very small improvement in accuracy which is at 92%. It is also observed that model started to overfit earlier than the previous models at 4 epochs.

In Model4, the reported accuracy is 93% and the overfitting effect of overfitting is even more predominant and starts earlier at around 3 epochs.

As we keep increasing the number hidden layers, we can see that the model tends to overfit more and more. Although the test accuracies are high with minute differences, their generalization capability is very poor. In order to overcome this, regularization techniques such as L1, L2 or dropouts can be applied. In the next task, the dropout technique combined with Models 3 and 4 will be studied.

Task 2

In this task, we will study about Dropout regularization and its impacts over a couple of overfitting models.

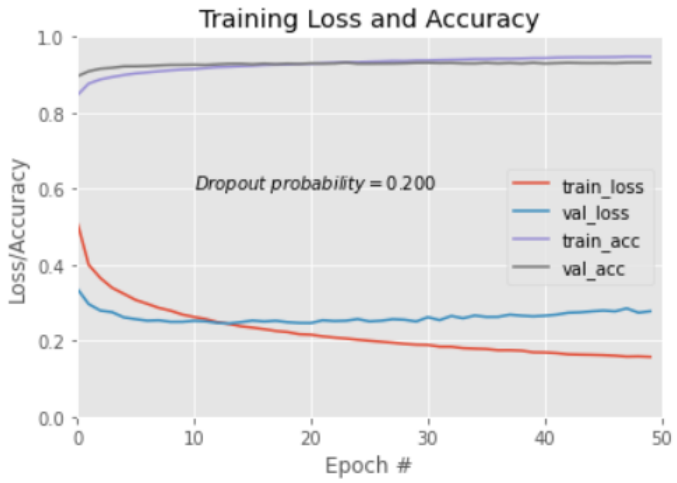
This technique associates a dropout probability for each node in each layer of the network. For a single training instance, a random probability is generated for each node in each layer. If a node's probability is lesser than a threshold value otherwise known as the dropout probability, then that node is removed from the network and the network is trained with the remaining nodes. This is repeated for each instance in the dataset. This technique enables the weights to be evenly distributed across the layer and eliminates the dependency over an incoming connection.

The Dropout technique was applied to the deepest models 3 and 4 from Task1.

Dropout for Model 3 from Task 1

The initial dropout probability was set to 0.2 for both the hidden layers. The results are shown below.

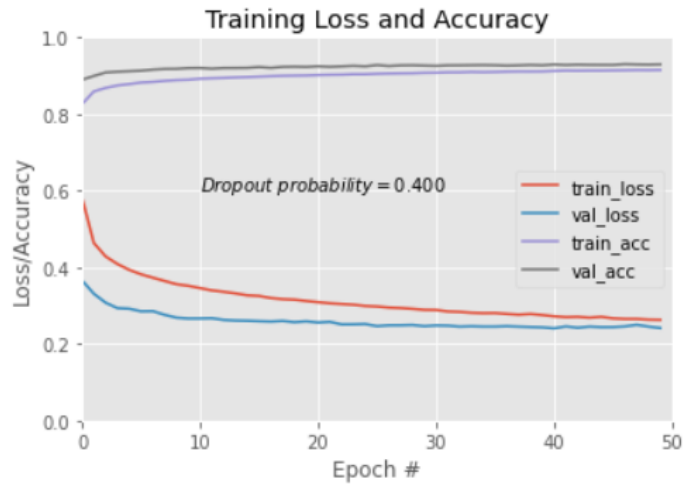
Train loss: 0.1573 Train accuracy: 0.9477
Validation loss: 0.2775 Validation accuracy: 0.9321



It can be observed that now the model starts to overfit only after 13 epochs. This allows the model to learn better and not depend on a single incoming connection.

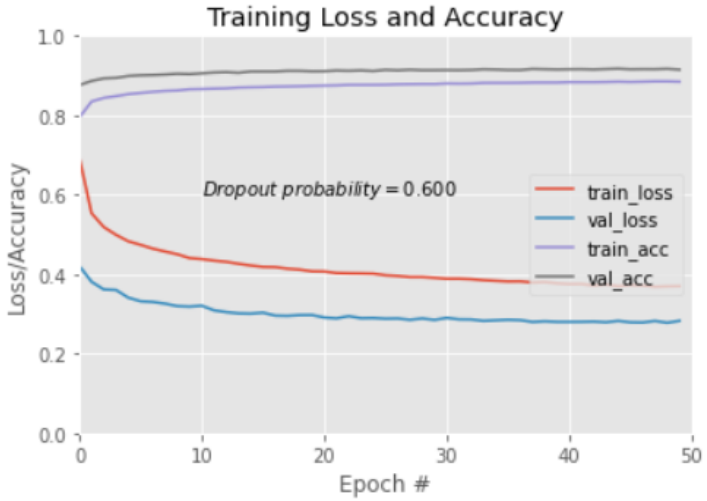
The Dropout probability set to 0.4 produces the below results

Train loss: 0.2628 Train accuracy: 0.9153
Validation loss: 0.2414 Validation accuracy: 0.9298



The dropout probability was then set to 0.6 for both the layers

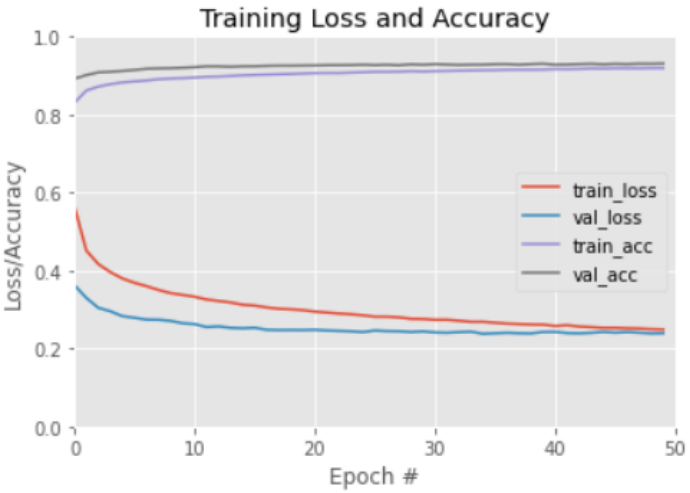
Train loss: 0.3700 Train accuracy: 0.8847
Validation loss: 0.2826 – Validation accuracy: 0.9151



From the above three experiments for different dropout probabilities, it is observed that as we increase the probability, the accuracy tends to decrease, and loss starts to increase thus causing the model to overfit. The model takes a longer time to converge.

After a few trials the ideal value for the dropout probability was found to be 0.4 for the 1st hidden layer and 0.2 for the second hidden layer. This produces a model that does not overfit and starts to converge around 50 epochs with the highest possible accuracy.

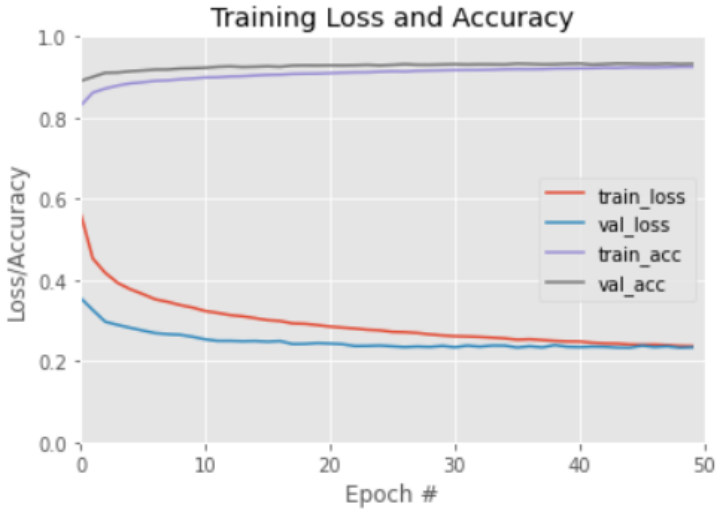
Train loss:0.2488 Train accuracy: 0.9195
Validation loss: 0.2393 Validation accuracy: 0.9312



Dropout for Model 4 from Task 1

A similar pattern of results was observed with the Model 4 as the Dropout probabilities were increased. However, the overfitting from Task1 was corrected using the Dropout technique with a dropout probability of 0.4 for all the hidden layers. The results are as shown below.

Train loss: 0.2367 Train accuracy: 0.9255
Validation loss: 0.2338 Validation accuracy: 0.9330



PART C – Research on Batch Normalization

Introduction

Batch normalization, on a high level is a technique that improves the training of a neural network by stabilizing the input distribution between the hidden layers to which it is applied to. The implementation of the technique involves adding additional network layers that controls the mean and variance of these input distributions.

Why Batch Normalization?

In general, Normalization is applied to the input feature data in order to represent all the feature values across a single scale from 0 to 1. This is done to eliminate the impact or domination of one feature over another in the prediction process. Another reason with respect to neural networks is that, the forward propagation involves the calculation of dot products of the weights and features values, which might result in very high output values, high computational times and increased memory usage, with unnormalized data. A similar issue occurs with back propagation, during gradient descent process. Hence, the input data is normalized to train the network faster and accurately.

The above approach works, if we have a single activation layer. However, when a neural network has additional hidden layers, normalizing the input data alone might still not be enough, as the distribution of the data passing through these hidden layers keeps changing during the training process, as the weights and the bias of the previous layers change. This phenomenon is referred as Internal covariant shift. This in turn increases the training time and makes it very difficult to train the model. Batch normalization addresses this problem, by normalizing the input data between the hidden layers in the network.

How Batch Normalization works?

Considering a neural network as shown below in Figure 1, the Batch normalization layers can be added where the red arrows are pointing, to handle the data passed from layer2, layer3 and layer4. The Internal Covariant shift more specifically refers to the change in the distribution of the activation values of each neuron associated with the layers, due to the varying weights and biases during training. As a result of this, all the hidden activation units passed from layer2 to layer3 and layer3 to layer4 will be normalized. Implementing this transformation for the entire training set at a time might not be practical for a very large dataset and will slow down the training process. Hence batch normalization is applied in mini batches of the training set and the estimated mean and variance of this mini batch will be used for the normalizing the data.

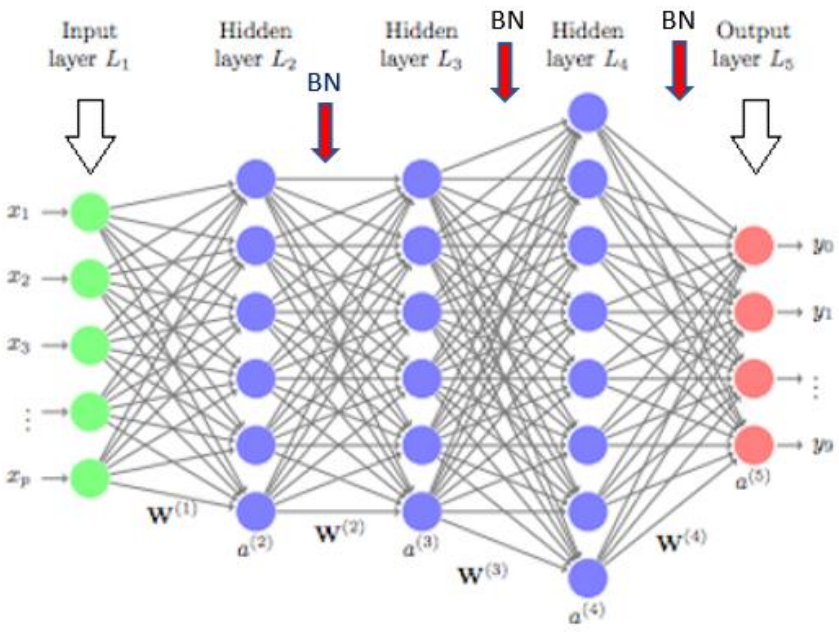


Figure 1

Although normalization means transforming the mean and variance to 0 and 1 respectively for the data, just doing that can reduce the expressive power of each neuron in the layer and can change what each activation unit represents. In order to make the transformation meaningful and better than the original distribution, two additional parameters **gamma** and **beta** are introduced for each activation unit. These two values will be used to scale and the shift the normalized value and allows the data to have a new mean and variance so that the representation of the original activations is restored. The mean, variance, gamma and beta are trainable parameters and will be learnt during the training process. As a result, these parameters will also be involved in the gradient calculation during backpropagation. The figure below explains the transformation that will be applied to each activation unit in the layer during the forward propagation.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 2

Benefits of Batch Normalization

As a result of applying batch normalization, the network will have normalized data coming in and normalized data within the model thus reducing the training time and eliminating the imbalance with the input distributions. The back propagation using the trainable parameters ensures that the model continues to learn the input distributions exhibiting internal covariant shift and thereby stabilizes the training process. Batch normalization also allows us to use higher learning rates thus accelerating the training process for a faster convergence. In some cases, it also acts as a regularizer and enables us to create a better generalized model even without a Dropout.

References

1) Lecture Notes
2) Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.
3) https://stackoverflow.com/questions/4674623/why-do-we-have-to-normalize-the-input-for-an-artificial-neural-network
4) https://orbograph.com/deep-learning-how-will-it-change-healthcare/
5) Bjorck, N., Gomes, C. P., Selman, B., & Weinberger, K. Q. (2018). Understanding batch normalization. In Advances in Neural Information Processing Systems (pp. 7694-7705).