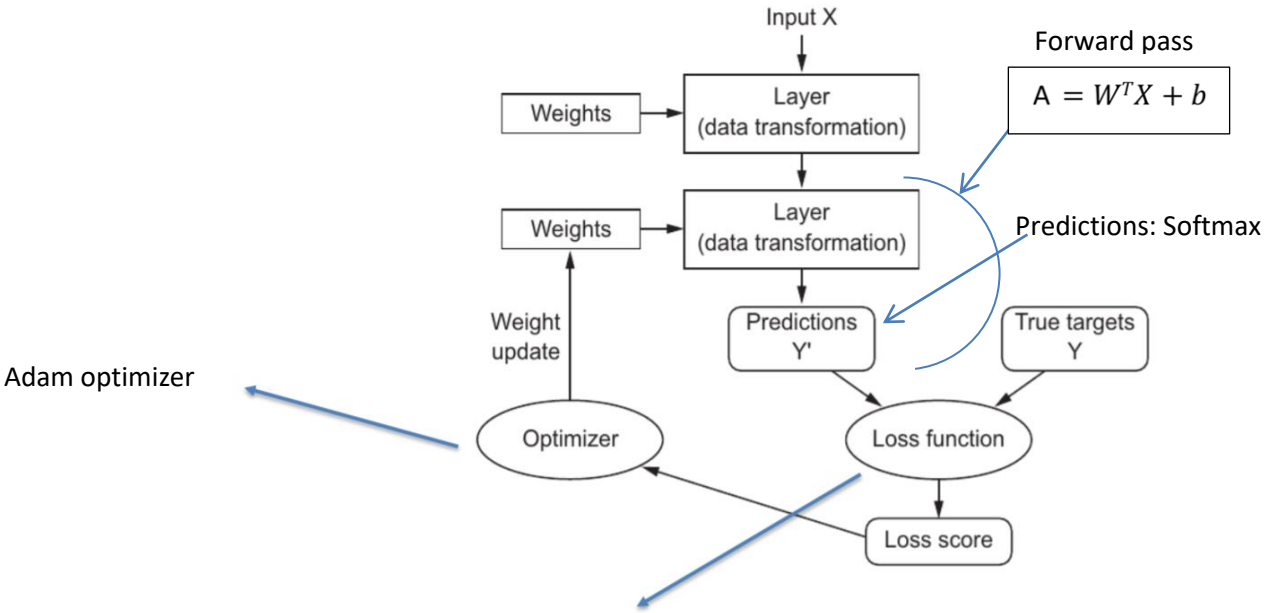
Part A: Tensor Flow's low-level API and directly using automatic differentiation

Question1_1: Build an SoftMax classifier



$$L(p^i, y^i) = \quad -\sum_{j=1}^{C} y_j^i \log(p_j^i)$$

$$C = \frac{1}{m}\sum_{i=1}^{m} L(p^i, y^i)$$

The main components of the question 1_1

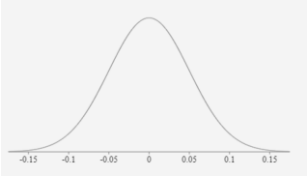
- Loading the data:

```
Shape of training features  (784, 60000)
Shape of test features  (784, 10000)
Shape of training labels  (10, 60000)
Shape of testing labels  (10, 10000)
```

- Initial random weight and bias

```
w = tf.Variable(tf.random.normal(shape=[784,10], mean=0.0, stddev=0.05))
```

  - Weight will be divided into the approximately same count of positive and negative weights as mean is kept as 0
  - Stddev is 0.05, to a majority of numbers between -0.05 to 0.05, and almost all of the values between -0.15 to 0.15
  - The distribution will follow the Gaussian curve
  - We will make a matrix of 784 (features) by 10 (Classes). The shape will be useful during the matrix multiplication later



- The gaussian curve for out random weight matrix

`b = tf.Variable([0.])`

- A single-bias value is good enough, we will keep the value as 0 for now

• Forward pass

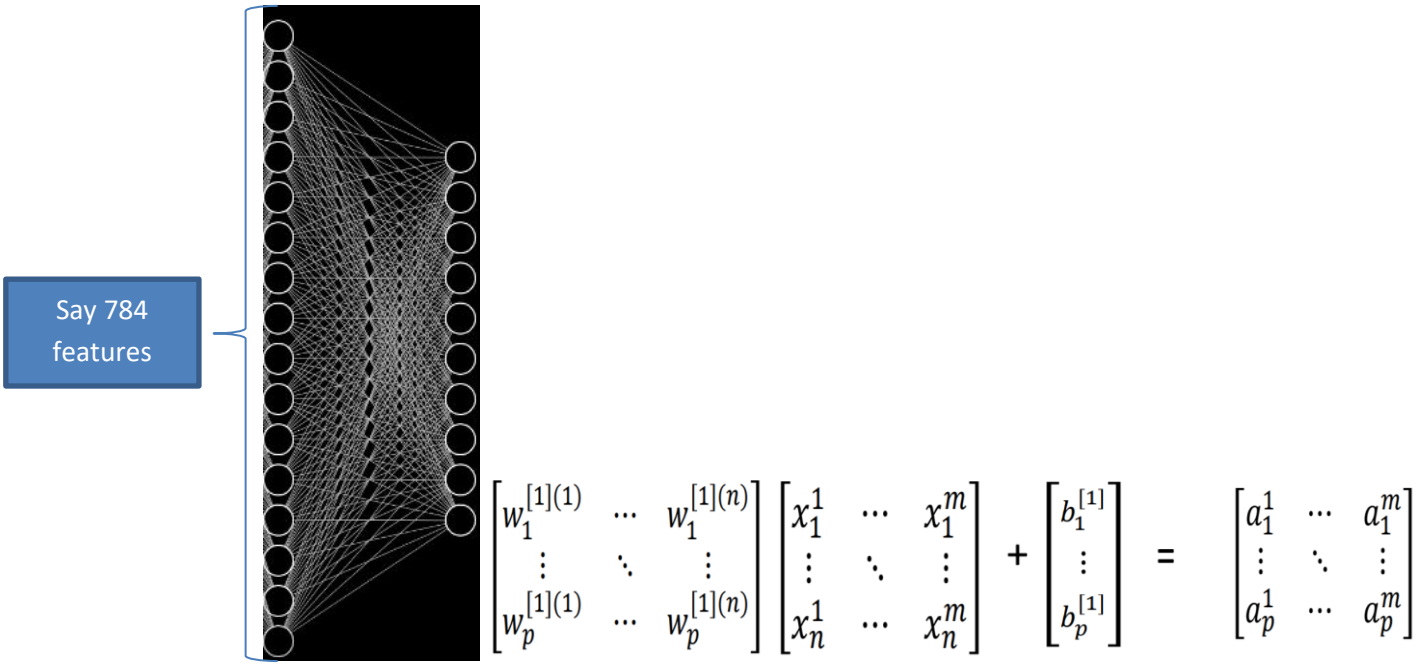
```
def forwardPass(x, w, b):
    w_T = tf.transpose(w)
    y_pred = tf.matmul(w_T, x) + b
    # Pipe the results through the softmax activiation function.
    # pass the values to the softmax layer, the softmax layer to convert the values to probabilites
    return softmax(y_pred)
```

- x is input data, feature values for all the instances, (training data - 60000 images)
- X has dimension as features by number of instances
- weight matrix w is created earlier, matrix of shape 784X10
- b is the bais

The following method will implement the formula given below.

$$A = W^T X + b$$

The **forward pass** is pushing 784 feature data to the 10 neurons.



Take a note of the shape of the matrices, the magic about the calculation is in the shape of the matrix.

The output will the activation values for the 10 classes for all the images, now the activation values will be passed to the softmax layer to converting them as probabilities. We will discuss softmax in next section

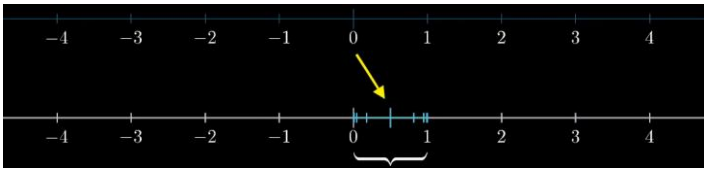
- **Softmax:**

```
# Receive values from a range, converts the range to 0 to 1; sum of all values will be 1
# Returns the probabilites
def softmax(y_pred):
    y_pred_exp = tf.math.exp(y_pred)
    summation = tf.reduce_sum(y_pred_exp, 0, keepdims=True)
    return y_pred_exp / summation
```

  - Receives input from the forwardPass method, the activation values for the classes for every test instance
  - The following method implements the following formula

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

  - The formula converts the activation values range to a new range 0 to 1 and the sum of all values to 1.



  - The softmax method is hugely used In multi-class classification problems (similar to ours). It will give us probabilities for all the 10 classes for the input images features.

- **Cross entropy**

```
def cross_entropy(y_true, y_pred):
    # Any values less than clip_value_min are set to clip_value_min. Any values greater than clip_value_max are set to clip_value_max.
    # if the calue is less than 1e-10, than it will set to 1e-10, and if the value is more than 1.0, it will set to 1.0
    y_pred = tf.clip_by_value(y_pred, 1e-10, 1.0)

    # Following is the formula for calculating the loss, and we will get loss for all the 10 classes
    loss = -tf.reduce_sum(y_true * tf.math.log(y_pred), axis=0)

    # Calculating the loss, by taking the mean of the losses
    return tf.reduce_mean(loss)
```

  - The method will use the following formula (2nd line), and the returns the mean of all the losses for the images

$$L(p^i, y^i) = -\sum_{j=1}^{c} y_j^i \log(p_j^i)$$

  - The cross-entropy is simple to understand, the loss will be low, if the probability of correctly prediction class is high, and loss will be high if the predicted probability of correct class is low.
  - the loss will be return by further applying the formula given below (Last line)
  - $$C = \frac{1}{m}\sum_{i=1}^{m} L(p^i, y^i)$$

- **Calculate accuracy**

```
def calculate_accuracy(x, y, w, b):
  # ForwardPass will internally class softmax, and we will get 10 probabilies for the image for fall under a class
  y_pred_softmax = forwardPass(tf.transpose(tr_x),w, b)

  # Round the predictions by the logistical unit to either 1 or 0
  # If the value will be more than 0.5, we will get a one, else we will set to 0
  predictions = tf.round(y_pred_softmax)

  # tf.equal will return a boolean array: True if prediction correct, False otherwise
  # tf.cast converts the resulting boolean array to a numerical array
  # 1 if True (correct prediction), 0 if False (incorrect prediction)
  predictions_correct = tf.cast(tf.equal(predictions, y), tf.float32)

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

  return accuracy
```

- The following method takes the following parameters
  - X is feature matrix. (We will get predictions from forward pass method)
  - Y is the actual classes
  - Weights
  - Bias
- We will call the forward_pass method to get the predictions from the network
- Predictions will be rounded by tf.round method. If the prediction confidence is more than 0.5, than we will predict 1; else marks 0.
  - `predictions = tf.round(y_pred_softmax)`
- Then we will check the actual class values, with the predicted using the code given below
  - `predictions_correct = tf.cast(tf.equal(predictions, y), tf.float32)`
- Finally, we will send the mean of the correct prediction, and t would be out accuracy.

- **TrainModel method**

```
def trainModel(num_Iterations = 500):
    # Iterate our training loop
    for i in range(num_Iterations):

      # Create an instance of GradientTape to monitor the forward pass
      # and calcualte the gradients for each of the variables m and c
      with tf.GradientTape() as tape:
        y_pred = forwardPass(tf.transpose(tr_x),w, b)
        currentLoss = cross_entropy(tr_y, y_pred)

      gradients = tape.gradient(currentLoss, [w, b])
      accuracy = calculate_accuracy(tr_x, tr_y, w, b)
      print ("Iteration ", i, ": Loss = ",currentLoss.numpy(), "  Acc: ", accuracy.numpy())
      tf.keras.optimizers.Adam().apply_gradients(zip(gradients, [w,b]))
```

  - The following method will train our model and show how the accuracy increase over the period of time
  - Iteration is epoch here, and with the help of Adam optimizer, we will try to improve the model.
  - Forward pass and the cross-entropy calculations are recorded on gradient tape using the following code

```
with tf.GradientTape() as tape:
  y_pred = forwardPass(tf.transpose(tr_x),w, b)
  currentLoss = cross_entropy(tr_y, y_pred)
```

  - Then calculate the gradients, using the tape values using the code given below

```
gradients = tape.gradient(currentLoss, [w, b])
```

  - And apply the Adam optimizer to update the weights and improve the model, code given below

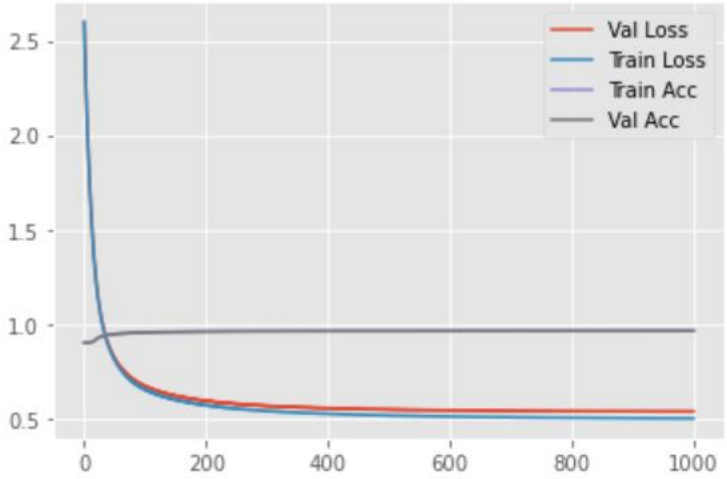
```
tf.keras.optimizers.Adam().apply_gradients(zip(gradients, [w,b]))
```

Tensorflow methods used for the tasks.

- **tf.cast :** Cast the data to a provided data type, the first parameter is data, the second parameter is TensorFlow data type
- **tf.Variable:** Create TensorFlow variable
- **tf.transpose:** Transpose matrix
- **tf.matmul:** Multiple matrices, please note the matrix size should be compatible (e.g. 6000 by 786 X 786 by 10)
- **tf.math.exp:** Returns exponent of the provided value
- **tf.reduce_sum:** Sums the values of the matrix, it can sum at 0 axis, 1 axis or the entire matrix sum

- **tf.clip_by_value:** Any values less than clip_value_min are set to clip_value_min. Any values greater than clip_value_max are set to clip_value_max. Takes 3 inputs: data, clip_value_min, clip_value_max
- **tf.reduce_mean:** Calculate means the values of the matrix, it can calculate mean at 0 axis, 1 axis or the entire matrix sum
- **tf.GradientTape():** Returns a gradient tape
- **numpy():** Given just the value from the tensorflow variable
- **tf.keras.optimizers.Adam:** Returns Adam optimizer object

Performance evaluation of the model (Sofmax)




```
************************************************************
Iteration  999 : train_loss =  0.49722296   train_acc:  0.96511835
Iteration  999 : test_loss =  0.53537333   test_acc:  0.96197
************************************************************
```

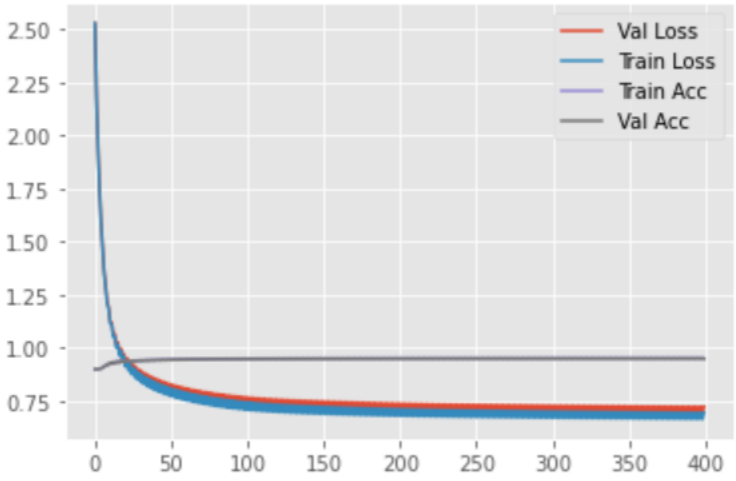
- The Accuracy of the model starts from 89 percent and reaches just above 96 percent and then becomes stagnant.
- After 450 epochs, we cannot see any further improvement inaccuracy. I am not sure if the model is over fitting of not. There is a very slight noise in the loss reading with the default learning rate of 0.001

```
************************************************************
Iteration  456 : train_loss =  0.51877195   train_acc:  0.96405
Iteration  456 : test_loss =  0.54757506   test_acc:  0.96162
************************************************************
```

- The accuracy for train and validation data is almost the same, we can't even see two lines in the graph

Learning rate 0.003

- **If we increase the learning rate to 0.003, the accuracy doesn't reach more than 95.2 percent after 400 epochs**

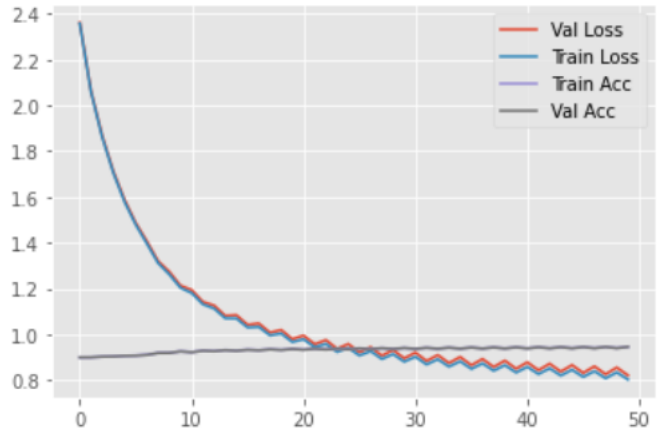



```
**********************************************************
Iteration  399 : train_loss =  0.6835369   train_acc:  0.9521767
Iteration  399 : test_loss =  0.71256906   test_acc:  0.95037
**********************************************************
```

The accuracies and loss become stagnant post 160 epochs

```
**********************************************************
Iteration  158 : train_loss =  0.6793891   train_acc:  0.95197
Iteration  158 : test_loss =  0.7007778   test_acc:  0.9507
**********************************************************
```

With the updated learning rate, there is a noise in the graph. The noise will be more evident if we reduce the epoch.




```
**********************************************************
Iteration  49 : train_loss =  0.8033278   train_acc:  0.9462733
Iteration  49 : test_loss =  0.8213385   test_acc:  0.94585
**********************************************************
```

By reviewing the effect of the learning rate, we can conclude that if we increase the rate of learning in Adam optimizer, we will see some noise in the graph.

Question1_2_1

Network architecture A **[Notebook should be called Question1_2_1]** :

a. Layer 1: 300 neurons (ReLu activation functions).
b. Layer 2: Softmax Layer (from Q1 (i))

- For implementing a hidden layer in our network we will update our forwardPass method and we will need 2 weight matrices. There will be a minor update in our train method as well.
- The following are the snippet of updated code.

```
w1 = tf.Variable(tf.random.normal(shape=[784,300], mean=0.0, stddev=0.05))
w2 = tf.Variable(tf.random.normal(shape=[300,10], mean=0.0, stddev=0.05))
b = tf.Variable([0.])
```

```
def forwardPass(x, w1,w2, b):
    w1 = tf.transpose(w1)
    w2 = tf.transpose(w2)

    y_pred = tf.matmul(w1, x) + b
    a1 = tf.maximum(y_pred, 1)

    y_pred = tf.matmul(w2, a1) + b

    return softmax(y_pred)
```

```
def trainModel(num_Iterations = 50):

    trainingLosses= []
    testLosses= []
    trainingAccuracies = []
    testAccuracies = []

    for i in range(num_Iterations):

      with tf.GradientTape() as tape:
        y_pred = forwardPass(tr_x,w1,w2, b)
        currentLoss = cross_entropy(tr_y, y_pred)

      gradients = tape.gradient(currentLoss, [w1,w2, b])

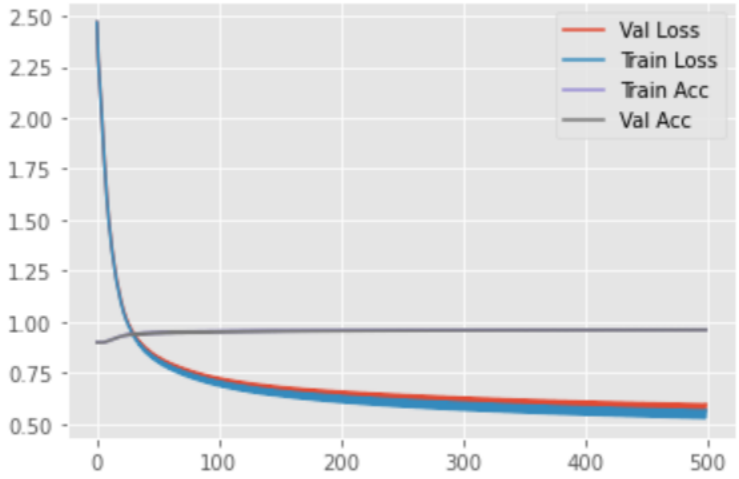

      tr_accuracy, y_pred_softmax = calculate_accuracy(tr_x, tr_y, w1,w2, b)
      te_accuracy, y_pred_softmax = calculate_accuracy(te_x, te_y, w1,w2, b)
      te_currentLoss = cross_entropy(te_y, y_pred_softmax)

      # Appending and print the information for training instances
      trainingAccuracies.append(tr_accuracy.numpy())
      trainingLosses.append(currentLoss.numpy())
      print ("Iteration ", i, ": train_loss = ",currentLoss.numpy(), "  train_acc: ", tr_accuracy.numpy())

      # Appending and print the information for validation instances
      testLosses.append(te_currentLoss.numpy())
      testAccuracies.append(te_accuracy.numpy())
      print ("Iteration ", i, ": test_loss = ",te_currentLoss.numpy(), "  test_acc: ", te_accuracy.numpy())
      print("*"*60)

      #Calling Adam optimizer the for updating the weights and trainfing the data
      tf.keras.optimizers.Adam(learning_rate=0.001).apply_gradients(zip(gradients, [w1,w2,b]))
```

- When we run the network with a hidden layer of 300 neurons, the following graph will be generated from the loss and accuracies overtime.
- Note that, the learning rate is kept at the default value of 0.001

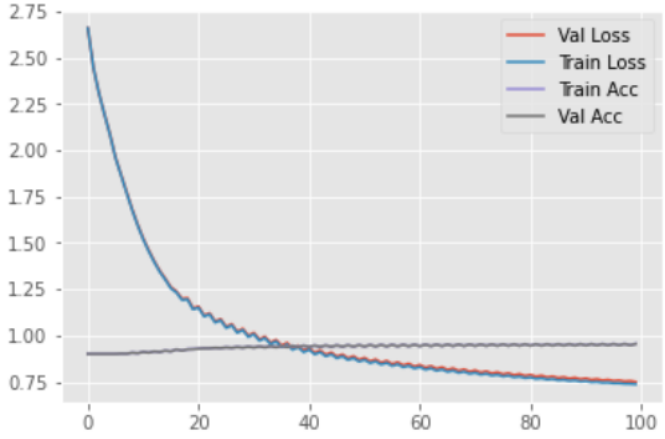



```
**********************************************************
Iteration  499 : train_loss =  0.5556481   train_acc:  0.96174
Iteration  499 : test_loss =  0.5879186   test_acc:  0.95938
**********************************************************
```

- After 500 epochs, we reach the accuracy of approx. 96 percent
- The model becomes stagnant post 305 epochs

```
**********************************************************
Iteration  302 : train_loss =  0.6095765   train_acc:  0.96005166
Iteration  302 : test_loss =  0.6308215   test_acc:  0.95848
**********************************************************
```

- If we observe the graph properly, we will find the noise in the graph. If we reduce the epochs to 100, the noise will be more evident. The graph with 100 epochs given below.



Question1_2_2

Network architecture B **[Notebook should be called Question1_2_2]**:

a. Layer 1: 300 neurons (ReLu activation functions).
b. Layer 2: 100 neurons (ReLu activation function)
c. Layer 3: Softmax Layer (from Q1 (i))

- For implementing 2 hidden layers in our network we will update our forwardPass method and we will need 3 weight matrices. There will be a minor update in our train method as well.
- The following are the snippet of updated code.

```
w1 = tf.Variable(tf.random.normal(shape=[784,300], mean=0.0, stddev=0.05))
w2 = tf.Variable(tf.random.normal(shape=[300,100], mean=0.0, stddev=0.05))
w3 = tf.Variable(tf.random.normal(shape=[100,10], mean=0.0, stddev=0.05))
b = tf.Variable([0.])
```

```
def forwardPass(x, w1,w2, w3, b):
    w1 = tf.transpose(w1)
    w2 = tf.transpose(w2)
    w3 = tf.transpose(w3)

    y_pred = tf.matmul(w1, x) + b
    a1 = tf.maximum(y_pred, 1)

    y_pred = tf.matmul(w2, a1) + b
    a2 = tf.maximum(y_pred, 1)

    y_pred = tf.matmul(w3, a2) + b
    return softmax(y_pred)
```

```
def trainModel(num_Iterations = 50):

    trainingLosses= []
    testLosses= []
    trainingAccuracies = []
    testAccuracies = []

    for i in range(num_Iterations):

      with tf.GradientTape() as tape:
        y_pred = forwardPass(tr_x,w1,w2,w3, b)
        currentLoss = cross_entropy(tr_y, y_pred)

      gradients = tape.gradient(currentLoss, [w1,w2,w3, b])

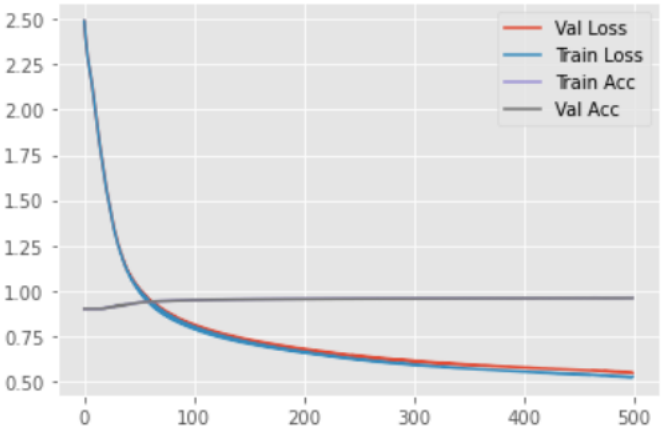

      tr_accuracy, y_pred_softmax = calculate_accuracy(tr_x, tr_y, w1,w2,w3, b)
      te_accuracy, y_pred_softmax = calculate_accuracy(te_x, te_y, w1,w2,w3, b)
      te_currentLoss = cross_entropy(te_y, y_pred_softmax)

      # Appending and print the information for training instances
      trainingAccuracies.append(tr_accuracy.numpy())
      trainingLosses.append(currentLoss.numpy())
      print ("Iteration ", i, ": train_loss = ",currentLoss.numpy(), "  train_acc: ", tr_accuracy.numpy())

      # Appending and print the information for validation instances
      testLosses.append(te_currentLoss.numpy())
      testAccuracies.append(te_accuracy.numpy())
      print ("Iteration ", i, ": test_loss = ",te_currentLoss.numpy(), "  test_acc: ", te_accuracy.numpy())
      print("*"*60)

      #Calling Adam optimizer the for updating the weights and trainfing the data
      tf.keras.optimizers.Adam(learning_rate=0.001).apply_gradients(zip(gradients, [w1,w2,w3,b]))
```

- When we run the network with 2 hidden layers with 300 and 100 neurons respectively, the following graph will be generated from the loss and accuracies overtime.
- Note that, the learning rate is kept at the default value of 0.001

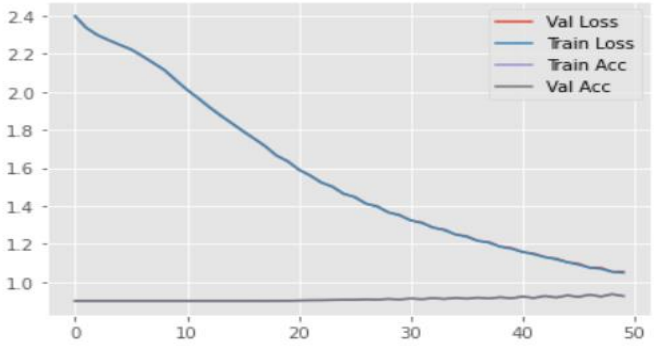



```
************************************************************
Iteration  499 : train_loss =  0.5248033   train_acc:  0.96170336
Iteration  499 : test_loss =  0.5464563   test_acc:  0.96008
************************************************************
```

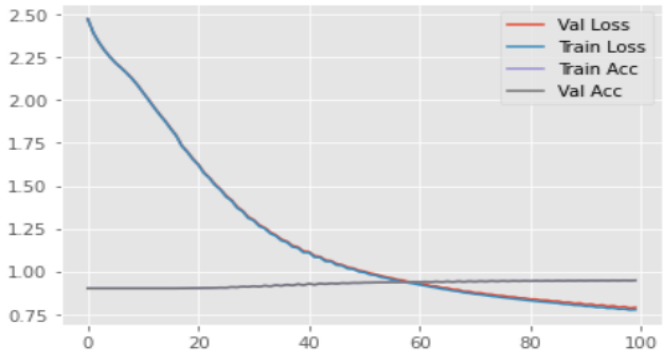
- After 500 epochs, we reach the accuracy of approx. 96 percent
- The model doesn't become stagnant and there is some learning.
- Let's run the network from 1000 epochs

```
************************************************************
Iteration  999 : train_loss =  0.43657142   train_acc:  0.96906835
Iteration  999 : test_loss =  0.4734895   test_acc:  0.96661
************************************************************
```

- We got some improvement in the performance of the network, the accuracy reaches more than 96.5 percent in our current network.
- If we check the graphs with fewer epochs, there is some noise. The graphs are given below.




```
************************************************************
Iteration  49 : train_loss =  1.0475686   train_acc:  0.925895
Iteration  49 : test_loss =  1.0540028   test_acc:  0.92565
************************************************************
```

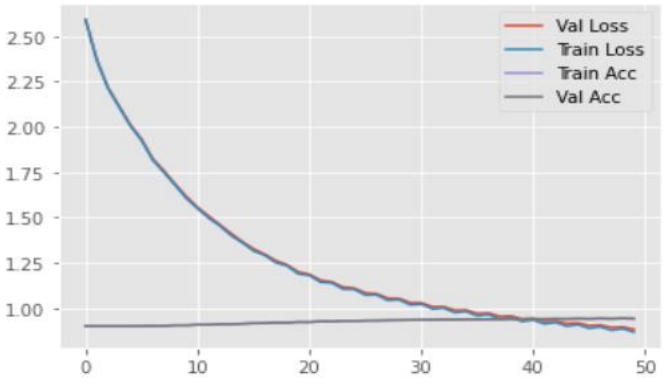



```
************************************************************
Iteration  99 : train_loss =  0.77574694   train_acc:  0.94824
Iteration  99 : test_loss =  0.78964365   test_acc:  0.94757
************************************************************
```

Evaluation of both of the above network architectures (Network A & Network B)

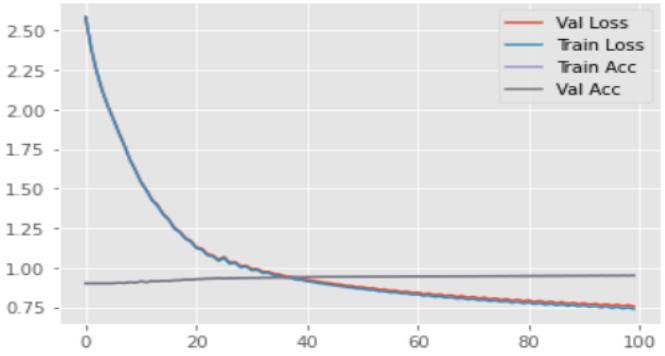
Network A:

- Accuracy after 50 epochs



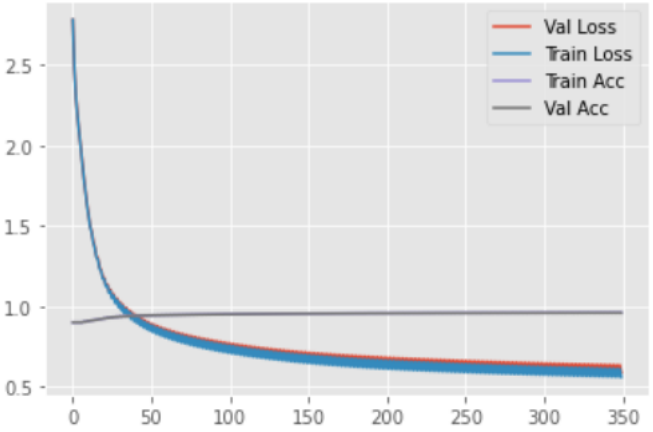

```
**********************************************************
Iteration  49 : train_loss =  0.8673137   train_acc:  0.94179165
Iteration  49 : test_loss =  0.88256544   test_acc:  0.94104
**********************************************************
```

- Accuracy after 100 epochs



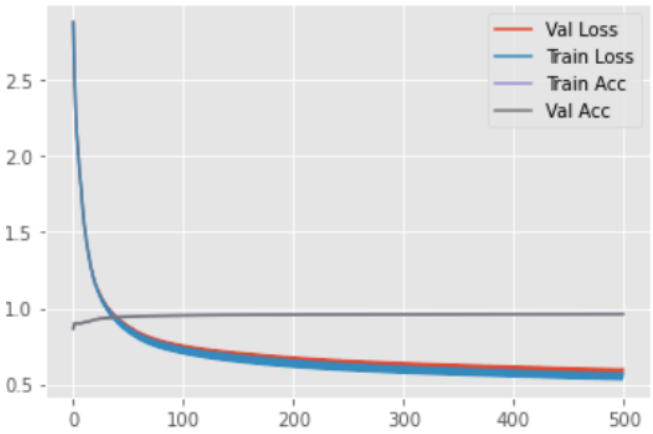

```
**********************************************************
Iteration  99 : train_loss =  0.7376588   train_acc:  0.94963664
Iteration  99 : test_loss =  0.75407785   test_acc:  0.94858
**********************************************************
```

- Accuracy after 350 epochs



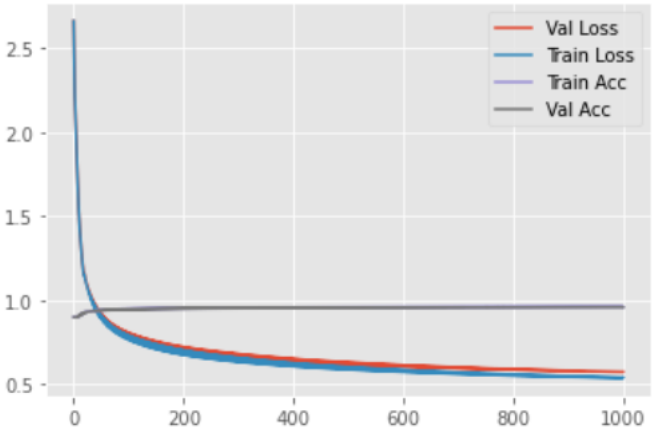

```
**********************************************************
Iteration  349 : train_loss =  0.5607551   train_acc:  0.96094334
Iteration  349 : test_loss =  0.58908087   test_acc:  0.95878
**********************************************************
```

- Accuracy after 500 epochs




```
************************************************************
Iteration  499 : train_loss =  0.5706172   train_acc:  0.963735
Iteration  499 : test_loss =  0.5985661   test_acc:  0.96212
************************************************************
```

- Accuracy after 1000 epochs



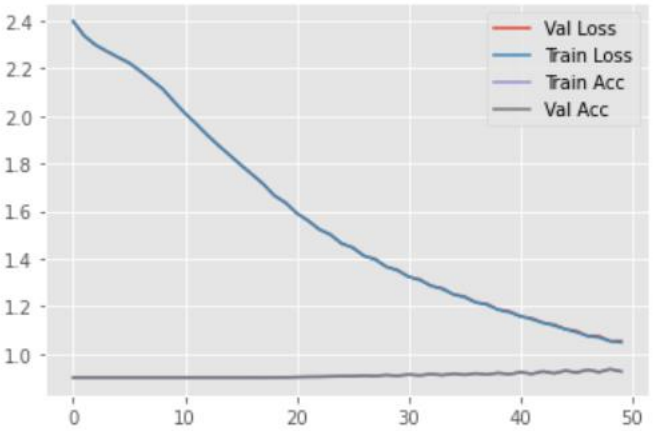

```
************************************************************
Iteration  999 : train_loss =  0.54166424   train_acc:  0.96078336
Iteration  999 : test_loss =  0.57501847   test_acc:  0.95906
************************************************************
```

- Stagnant value (Accuracy flatten out) value: Almost flatten post 500 epochs (Note, every run given slight different results)
- **Note: The network do improve over time, very slightly post 300 epochs**
- **Peak performance is given below**

```
************************************************************
Iteration  994 : train_loss =  0.5381937   train_acc:  0.963255
Iteration  994 : test_loss =  0.57459366   test_acc:  0.96053
************************************************************
```

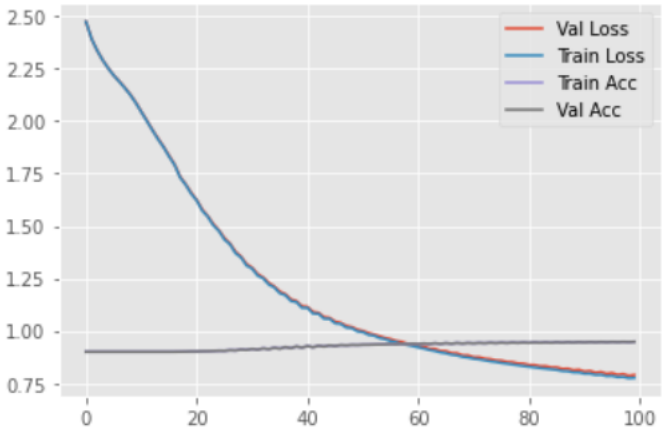
Network B:

- Accuracy after 50 epochs



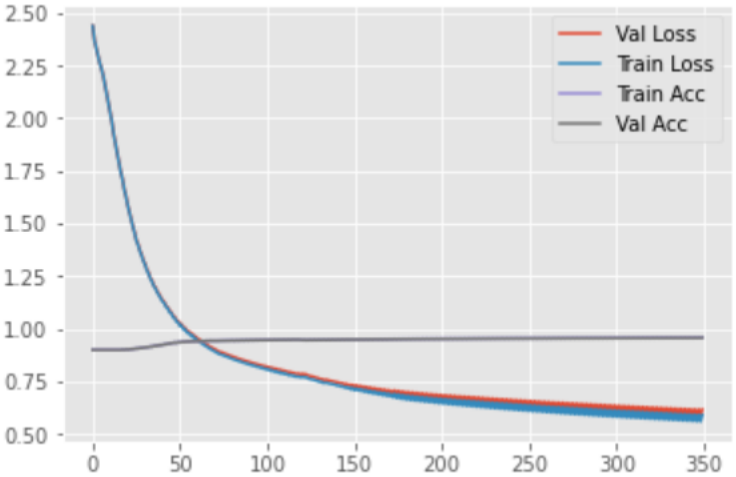

```
************************************************************
Iteration  49 : train_loss =  1.0475686   train_acc:  0.925895
Iteration  49 : test_loss =  1.0540028   test_acc:  0.92565
************************************************************
```

- Accuracy after 100 epochs



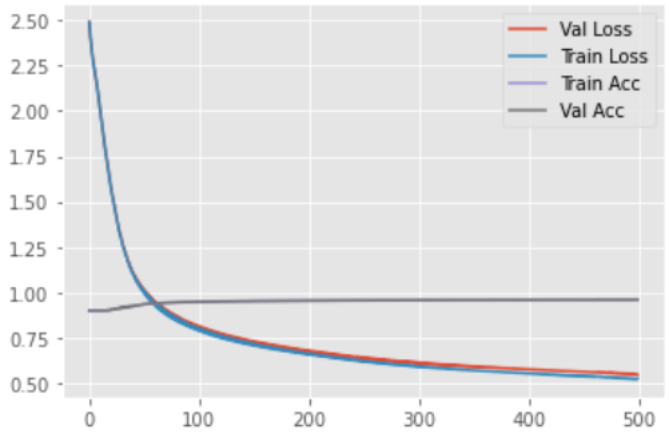

```
************************************************************
Iteration  99 : train_loss =  0.77574694   train_acc:  0.94824
Iteration  99 : test_loss =  0.78964365   test_acc:  0.94757
************************************************************
```

- Accuracy after 350 epochs



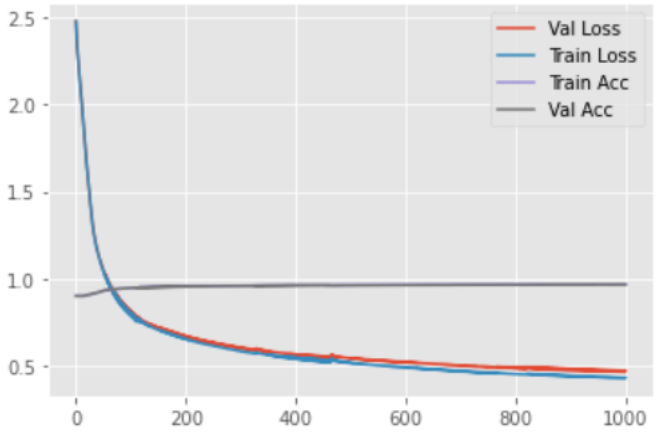

```
************************************************************
Iteration  349 : train_loss =  0.591542   train_acc:  0.95915836
Iteration  349 : test_loss =  0.6167223   test_acc:  0.95804
************************************************************
```

- Accuracy after 500 epochs




```
************************************************************
Iteration  499 : train_loss =  0.5248033   train_acc:  0.96170336
Iteration  499 : test_loss =  0.5464563   test_acc:  0.96008
************************************************************
```

- Accuracy after 1000 epochs




```
************************************************************
Iteration  999 : train_loss =  0.43001518   train_acc:  0.9682933
Iteration  999 : test_loss =  0.4712207   test_acc:  0.96539
************************************************************
```

- Stagnant value (Accuracy flatten out) value: The learning went very slow post 350 epochs
- Peak performance

```
************************************************************
Iteration  996 : train_loss =  0.4284375   train_acc:  0.96851164
Iteration  996 : test_loss =  0.4641331   test_acc:  0.96634
************************************************************
```

Differences in network A and Network

- Network 1 provides more noise compared with network B
- For reaching 96 percent accuracy from test and validations instances, network A takes 893 epochs, whereas network B takes 337 epochs

```
************************************************************
Iteration  892 : train_loss =  0.5451379   train_acc:  0.96276164
Iteration  892 : test_loss =  0.58059365   test_acc:  0.96007
************************************************************
```

Network A

```
**********************************************************
Iteration  336 : train_loss =  0.56881404   train_acc:  0.96098834
Iteration  336 : test_loss =  0.5853393   test_acc:  0.96004
**********************************************************
```

Network B

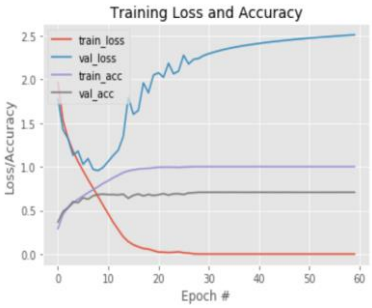
- Network B requires more time to train but overall gives better results

Are these architectures sensitive to the number of neurons in each layer?

- With the introduction with layers, there is very little difference in final accuracy, the networks do improve with the introduction with hidden layers. But the improvement is tiny

Is there evidence of these networks over-fitting?

- No substantial evidence of overfitting.
- There is a very slight divergence in loss reading



The graph from the model doesn't diverge clearly as the example given above

Question1_3: L1 and L2 regularization techniques

L1 Regularization: Cross entropy method

```
def cross_entropyL1(y_true, y_pred, factor = 0.0001):
    m = 3
    y_pred = tf.clip_by_value(y_pred, 1e-10, 1.0)
    regularlization = 0
    loss_per_classes = -tf.reduce_sum(y_true * tf.math.log(y_pred), axis=0)
    regularlization = (factor/2) * (  tf.reduce_sum(tf.abs(w1))+  tf.reduce_sum(tf.abs(w2)) + tf.reduce_sum(tf.abs(w2)))
    return tf.reduce_mean(loss_per_classes+regularlization)
```

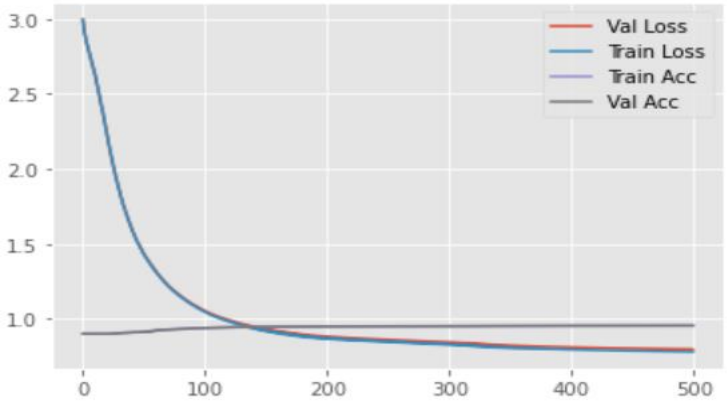
L2 Regularization: Cross entropy method

```
def cross_entropyL2(y_true, y_pred, factor=0.001):
    y_pred = tf.clip_by_value(y_pred, 1e-10, 1.0)
    loss_per_classes = -tf.reduce_sum(y_true * tf.math.log(y_pred), axis=0)
    regularlization = (factor/2 ) * (  tf.reduce_sum(tf.square(w1))+  tf.reduce_sum(tf.square(w2)) + tf.reduce_sum(tf.square(w3)))
    return tf.reduce_mean(loss_per_classes+regularlization)
```

Results from L1 regularization

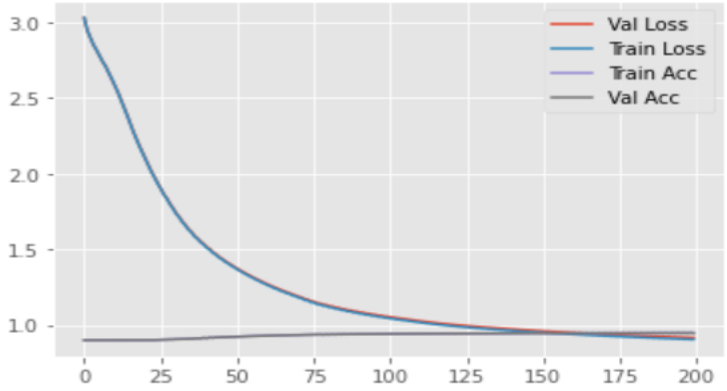
- With Regularization factor: 0.0001

Epochs 500



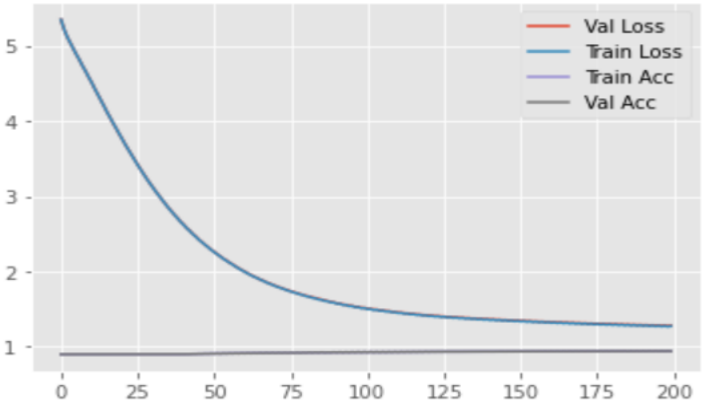

```
**********************************************************
Iteration  499 : train_loss =  0.7801579   train_acc:  0.9546367
Iteration  499 : test_loss =  0.7943437   test_acc:  0.95388
**********************************************************
```

Epochs 200



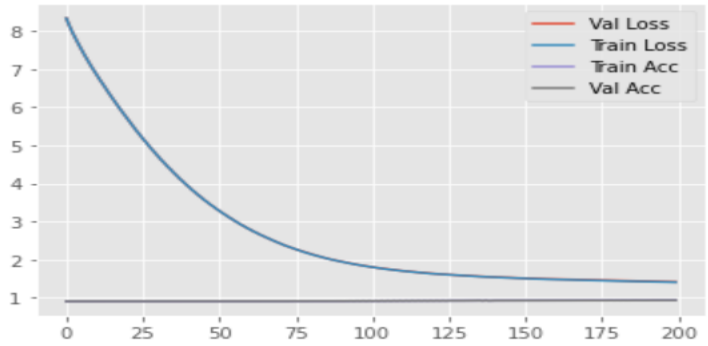

```
************************************************************
Iteration  199 : train_loss =  0.9057781   train_acc:  0.94764835
Iteration  199 : test_loss =  0.9163254   test_acc:  0.94663
************************************************************
```

With factor 0.0005




```
************************************************************
Iteration  199 : train_loss =  1.2756306   train_acc:  0.943135
Iteration  199 : test_loss =  1.2821356   test_acc:  0.94319
************************************************************
```

With factor 0.001

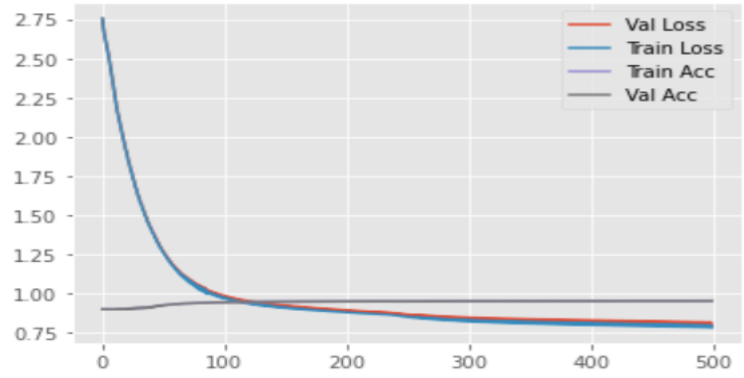



```
************************************************************
Iteration  199 : train_loss =  1.4016005   train_acc:  0.92866164
Iteration  199 : test_loss =  1.4097329   test_acc:  0.92849
************************************************************
```

Results from L2 regularization

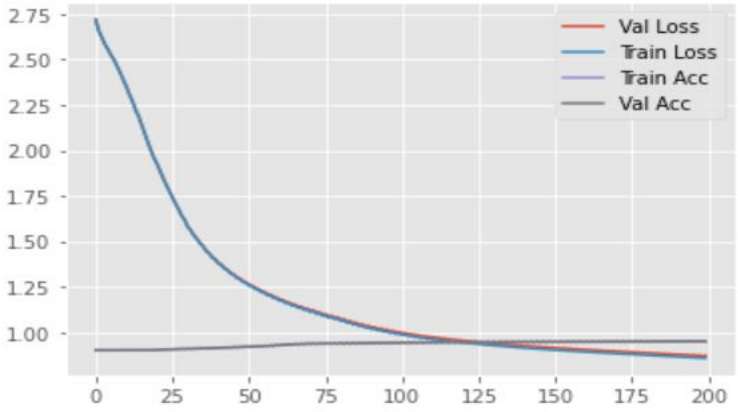
With regularization factor: 0.001

500 epochs



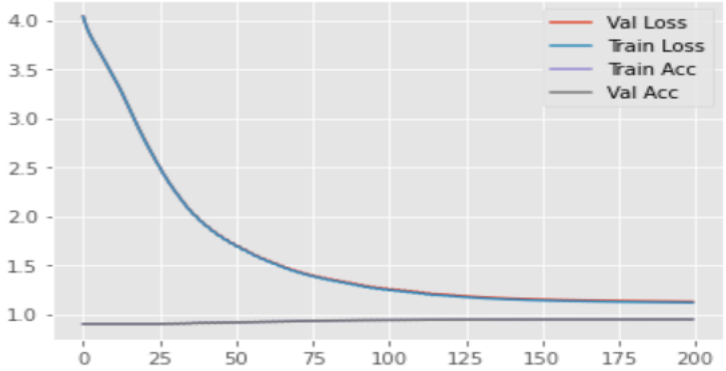

```
************************************************************
Iteration  499 : train_loss =  0.78377676   train_acc:  0.95522666
Iteration  499 : test_loss =  0.7971078   test_acc:  0.95442
************************************************************
```

200 epochs



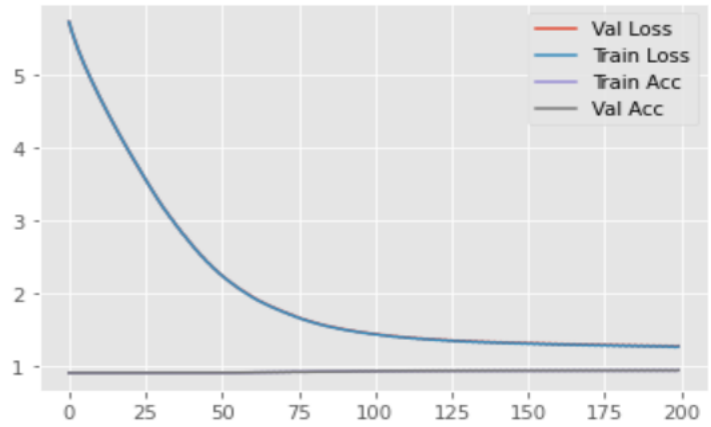
With regularization factor: 0.005

200 epochs




```
************************************************************
Iteration  199 : train_loss =  1.116907   train_acc:  0.9472067
Iteration  199 : test_loss =  1.1280153   test_acc:  0.94695
************************************************************
```

With factor 0.01




```
************************************************************
Iteration  199 : train_loss =  1.2642611   train_acc:  0.93743
Iteration  199 : test_loss =  1.270668   test_acc:  0.93767
************************************************************
```

Impact of both L1 and L2 on the model

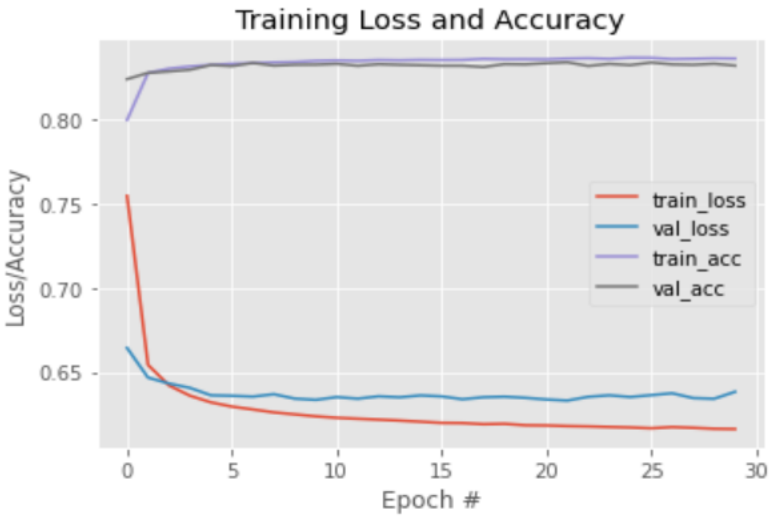
- Decreases accuracy a little, as regularization penalize the high weight
- Smoothen the graph, decreases the noise in the readings
- L2 performs slightly better
- L1 factor 0.0001 and L2 factor 0.001 performs better
- With higher regularization factor, we get higher weight values
- As there is no substantial evidence of over-fitting, the impact of regularizations is not evident that much

PART B - Keras – High Level API

Note: Accuracy priority is validation accuracy

SoftMax classifier

```
model=tf.keras.models.Sequential([layers.Dense(10, activation=tf.nn.softmax, input_shape= (784,))])
```



- **Accuracy after 30 epochs: 0.83**

```
Epoch 30/30
170000/170000 [==============================] - 1s 8us/sample - loss: 0.6162 - accuracy: 0.8361 - val_loss: 0.6383 - val_accuracy: 0.8319
```

- **Peak accuracy: 0.8339**

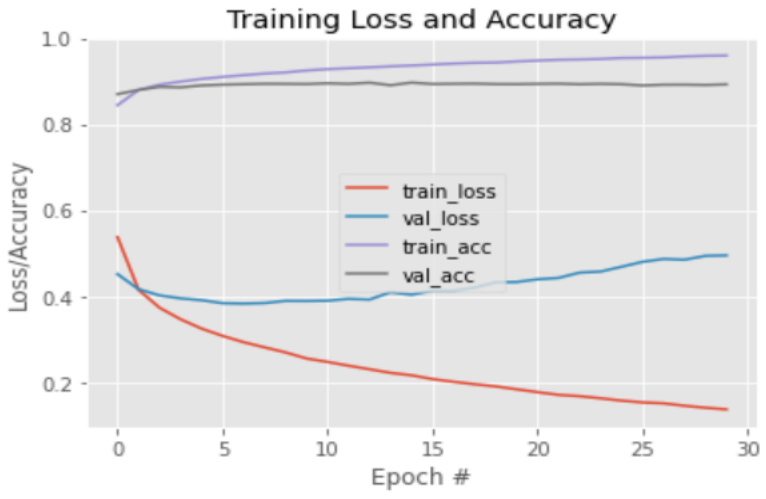
```
Epoch 22/30
170000/170000 [==============================] - 2s 9us/sample - loss: 0.6180 - accuracy: 0.8361 - val_loss: 0.6331 - val_accuracy: 0.8339
```

- **Over-fitting: Mild over fitting**
- **Approximate Over-fitting epoch: 9**
- **Accuracy when Over-fitting epoch: 0.8324**

```
Epoch 9/30
170000/170000 [==============================] - 1s 9us/sample - loss: 0.6249 - accuracy: 0.8340 - val_loss: 0.6343 - val_accuracy: 0.8324
```

2 layers: L1 200 Neurons L2 Softmax:

```
model=tf.keras.models.Sequential([layers.Dense(200, activation=tf.nn.relu, input_shape= (784,)),
                                  layers.Dense(10, activation=tf.nn.softmax)])
```



- **Accuracy after 30 epochs: 0.8926**

```
Epoch 30/30
170000/170000 [==============================] - 2s 15us/sample - loss: 0.1368 - accuracy: 0.9597 - val_loss: 0.4948 - val_accuracy: 0.8926
```

- **Peak accuracy: 0.8942**

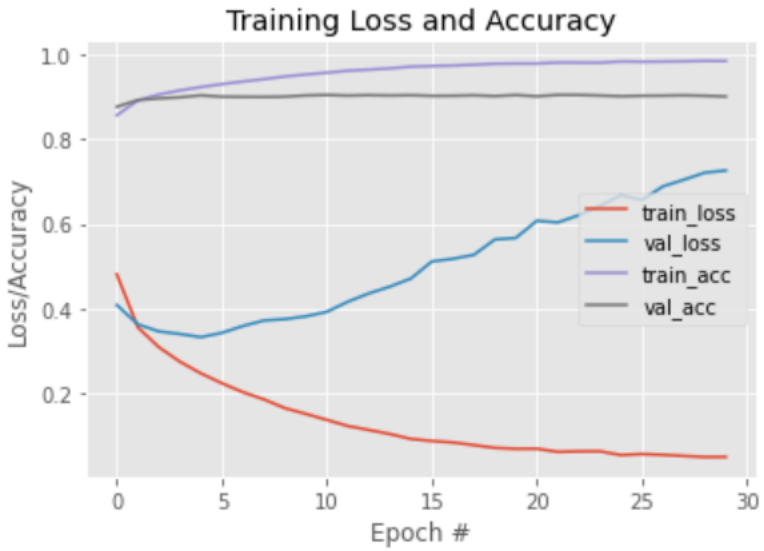
```
Epoch 22/30
170000/170000 [==============================] - 3s 15us/sample - loss: 0.1706 - accuracy: 0.9498 - val_loss: 0.4425 - val_accuracy: 0.8942
```

- **Over-fitting: Yes**
- **Approximate Over-fitting epoch: 5**
- **Accuracy when Over-fitting epoch: 0.89**

```
Epoch 5/30
170000/170000 [==============================] - 2s 14us/sample - loss: 0.3247 - accuracy: 0.9052 - val_loss: 0.3907 - val_accuracy: 0.8900
```

3 Layers: L1 400 Neurons L2 200 Neurons L3 Softmax

```
model=tf.keras.models.Sequential([layers.Dense(400, activation=tf.nn.relu, input_shape= (784,)),
                                  layers.Dense(200, activation=tf.nn.relu),
                                  layers.Dense(10, activation=tf.nn.softmax)])
```



- **Accuracy after 30 epochs: 0.90**

```
Epoch 30/30
170000/170000 [==============================] - 4s 25us/sample - loss: 0.0493 - accuracy: 0.9846 - val_loss: 0.7259 - val_accuracy: 0.9007
```

- **Peak accuracy:0.9046**

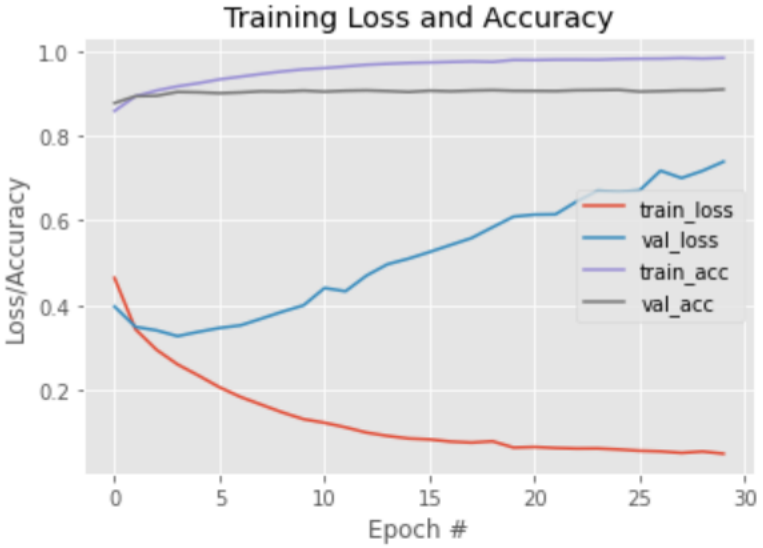
```
Epoch 22/30
170000/170000 [==============================] - 5s 28us/sample - loss: 0.0612 - accuracy: 0.9810 - val_loss: 0.6032 - val_accuracy: 0.9046
```

- **Over-fitting: Yes**
- **Approximate Over-fitting epoch: 4**
- **Accuracy when Over-fitting epoch: 0.89**

```
Epoch 4/30
170000/170000 [==============================] - 4s 26us/sample - loss: 0.2744 - accuracy: 0.9151 - val_loss: 0.3397 - val_accuracy: 0.8987
```

4 Layers: L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 Softmax

```
model=tf.keras.models.Sequential([layers.Dense(600, activation=tf.nn.relu, input_shape= (784,)),
                                  layers.Dense(400, activation=tf.nn.relu),
                                  layers.Dense(200, activation=tf.nn.relu),
                                  layers.Dense(10, activation=tf.nn.softmax)])
```



- **Accuracy after 30 epochs: 0.91**

```
Epoch 30/30
170000/170000 [==============================] - 8s 45us/sample - loss: 0.0484 - accuracy: 0.9848 - val_loss: 0.7394 - val_accuracy: 0.9103
```

- **Peak accuracy:0.913**

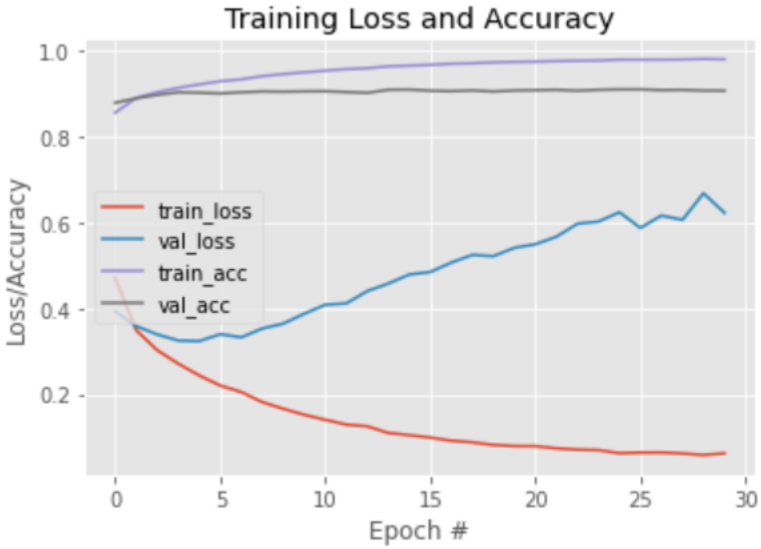
```
Epoch 30/30
170000/170000 [==============================] - 8s 45us/sample - loss: 0.0484 - accuracy: 0.9848 - val_loss: 0.7394 - val_accuracy: 0.9103
```

- **Over-fitting: Yes**
- **Approximate Over-fitting epoch:3**
- **Accuracy when Over-fitting epoch: 0.89**

```
Epoch 3/30
170000/170000 [==============================] - 7s 42us/sample - loss: 0.2943 - accuracy: 0.9076 - val_loss: 0.3402 - val_accuracy: 0.8952
```

5 Layers: L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 100 Neurons L5 Softmax

```
model=tf.keras.models.Sequential([layers.Dense(600, activation=tf.nn.relu, input_shape= (784,)),
                                  layers.Dense(400, activation=tf.nn.relu),
                                  layers.Dense(200, activation=tf.nn.relu),
                                  layers.Dense(200, activation=tf.nn.relu),
                                  layers.Dense(10, activation=tf.nn.softmax)])
```



- **Accuracy after 30 epochs: 0.90**

```
Epoch 30/30
170000/170000 [==============================] - 8s 48us/sample - loss: 0.0624 - accuracy: 0.9806 - val_loss: 0.6230 - val_accuracy: 0.9074
```

- **Peak accuracy: 0.916**

```
Epoch 26/30
170000/170000 [==============================] - 8s 46us/sample - loss: 0.0643 - accuracy: 0.9798 - val_loss: 0.5878 - val_accuracy: 0.9106
```

- **Over-fitting: Yes**
- **Approximate Over-fitting epoch:3**
- **Accuracy when Over-fitting epoch:**

```
170000/170000 [==============================] - 8s 49us/sample - loss: 0.3493 - accuracy: 0.8907 - val_loss: 0.3587 - val_accuracy: 0.8897
Epoch 3/30
```

Model training and graph building code

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
history = model.fit(trainX, trainY, validation_split=0.15, epochs=epochs,batch_size=256)
resultsProb = model.predict(testX)
results = np.argmax(resultsProb, axis =1)
print(confusion_matrix(testY, results))

plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, epochs), history.history["loss"], label="train_loss")
plt.plot(np.arange(0, epochs), history.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, epochs), history.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, epochs), history.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
```

Observation	**Network**				
	1 Layer	2 Layer	3 Layer	4 Layer	5 Layer
Over-fitting	Mild	Yes	Yes	Yes	Yes
Accuracy after 30 epochs	0.8319	0.8926	0.9007	0.9103	0.9074
Peak accuracy	0.8339	0.8942	0.9046	0.913	0.916
Over fitting epoch	9	5	4	3	3
Accuracy at over-fitting epoch	0.8324	0.8900	0.8987	0.8952	0.8897

Observations

- With more hidden layers, learning rate increase and model display evidence of over-fitting
- Hidden layers can increase the performance of the network
- For current data, network with 4 layers is performing the best (Optimal)
- Data complexity and optimal hidden count are directly proportional
- We may try to improve the current network with regularization

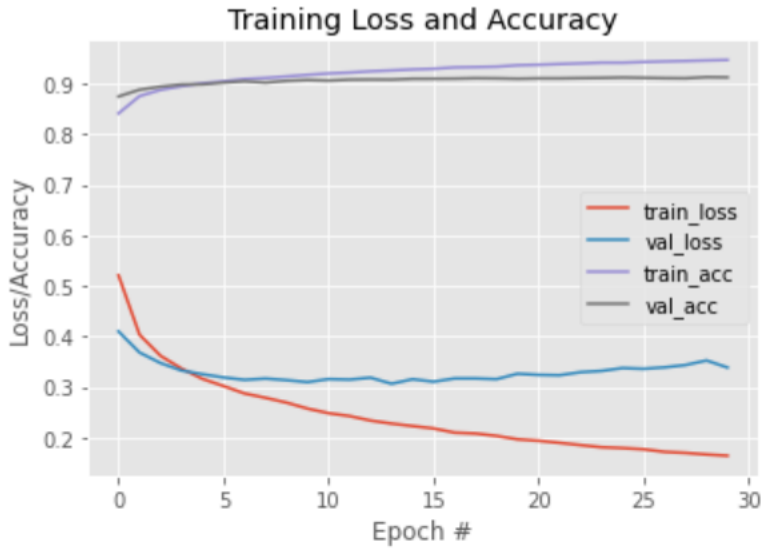
Part B- Regularization technique

Note: Accuracy priority is validation accuracy

4 Layers: L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 Softmax with dropout regularization

Drop-out 0.2 on every layer

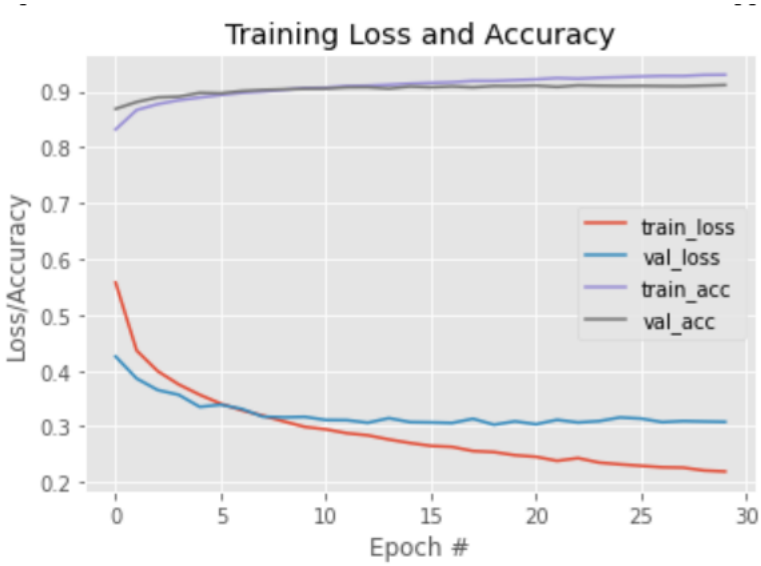
```
<matplotlib.legend.Legend at 0x2a8171fbe10>
```



- Over-fitting epoch: 9
- Accuracy before over-fitting: 0.9058

```
Epoch 9/30
170000/170000 [==============================] - 9s 50us/sample - loss: 0.2690 - accuracy: 0.9143 - val_loss: 0.3135 - val_accuracy: 0.9058
```

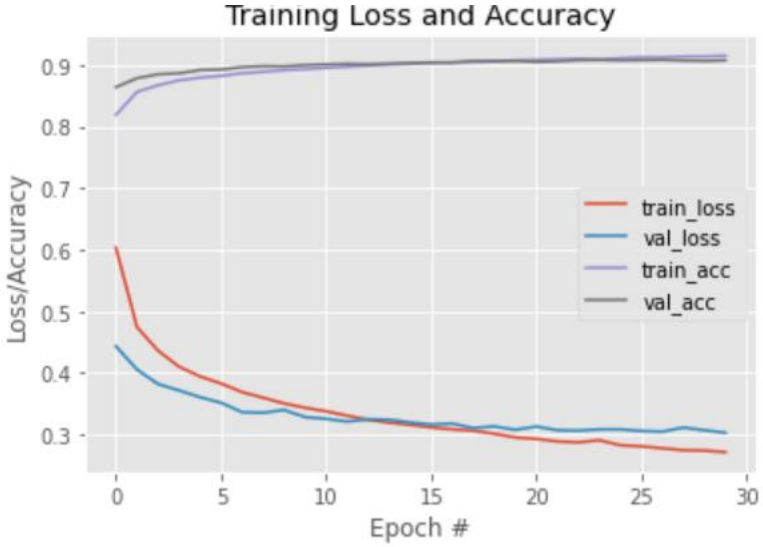
Drop-out 0.3 on every layer



- Over-fitting epoch: 12
- Accuracy before over-fitting: 0.9073

```
Epoch 12/30
170000/170000 [==============================] - 12s 73us/sample - loss: 0.2880 - accuracy: 0.9096 - val_loss: 0.3113 - val_accuracy: 0.9073
```

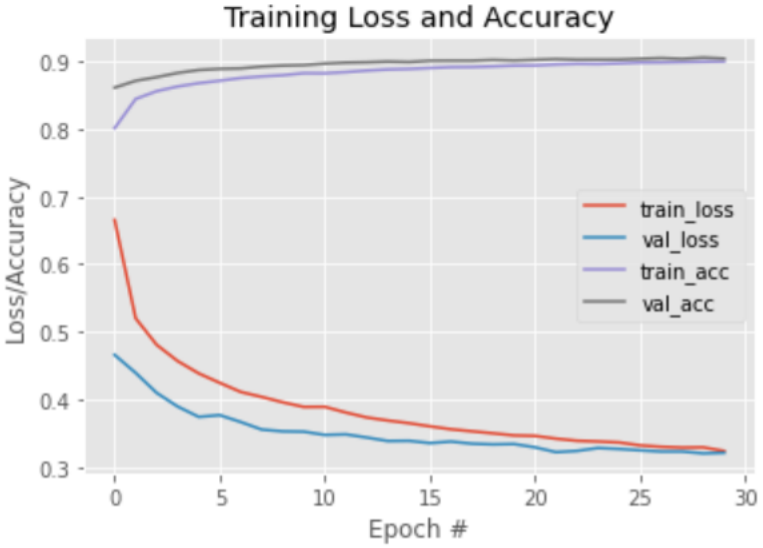
Drop-out 0.4 on every layer



- Over-fitting epoch: No/mild over-fitting: 27
- Accuracy before over-fitting: : 0.9078

```
Epoch 27/30
170000/170000 [==============================] - 9s 50us/sample - loss: 0.2768 - accuracy: 0.9130 - val_loss: 0.3086 - val_accuracy: 0.9078
```

Drop-out 0.5 on every layer

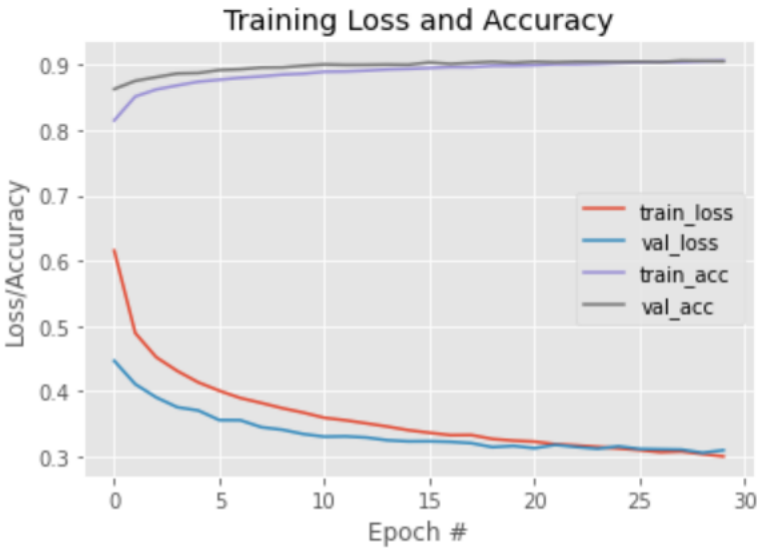


- Over-fitting epoch: NA
- Accuracy before over-fitting: (After 30 epoch): 0.9042

```
Epoch 30/30
170000/170000 [==============================] - 12s 73us/sample - loss: 0.3237 - accuracy: 0.9000 - val_loss: 0.3210 - val_accuracy: 0.9042
```

Drop-out 0.5 to 0.3 (Step -0.1) on every layer

```
model=tf.keras.models.Sequential([layers.Dense(600, activation=tf.nn.relu, input_shape= (784,)),
                                  layers.Dropout(0.5),
                                  layers.Dense(400, activation=tf.nn.relu),
                                  layers.Dropout(0.4),
                                  layers.Dense(200, activation=tf.nn.relu),
                                  layers.Dropout(0.3),
                                  layers.Dense(10, activation=tf.nn.softmax)])
```

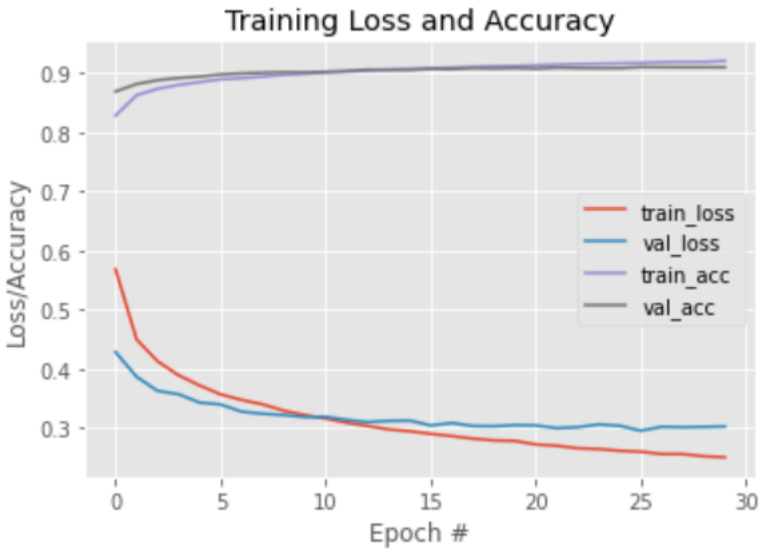


- Over-fitting epoch: 21
- Accuracy before over-fitting: 0.9051

```
Epoch 21/30
170000/170000 [==============================] - 8s 46us/sample - loss: 0.3225 - accuracy: 0.8998 - val_loss: 0.3124 - val_accuracy: 0.9051
```

Drop-out 0.4 to 0.2 (Step -0.1) on every layer

```
model=tf.keras.models.Sequential([layers.Dense(600, activation=tf.nn.relu, input_shape= (784,)),
                                  layers.Dropout(0.4),
                                  layers.Dense(400, activation=tf.nn.relu),
                                  layers.Dropout(0.3),
                                  layers.Dense(200, activation=tf.nn.relu),
                                  layers.Dropout(0.2),
                                  layers.Dense(10, activation=tf.nn.softmax)])
```

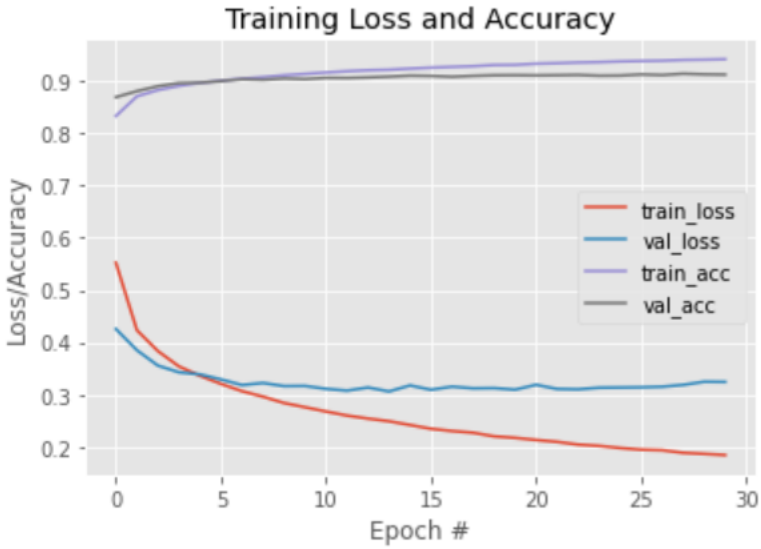


- Over-fitting epoch: 12
- Accuracy before over-fitting: 0.9036

```
Epoch 12/30
170000/170000 [==============================] - 8s 45us/sample - loss: 0.3085 - accuracy: 0.9027 - val_loss: 0.3134 - val_accuracy: 0.9036
```

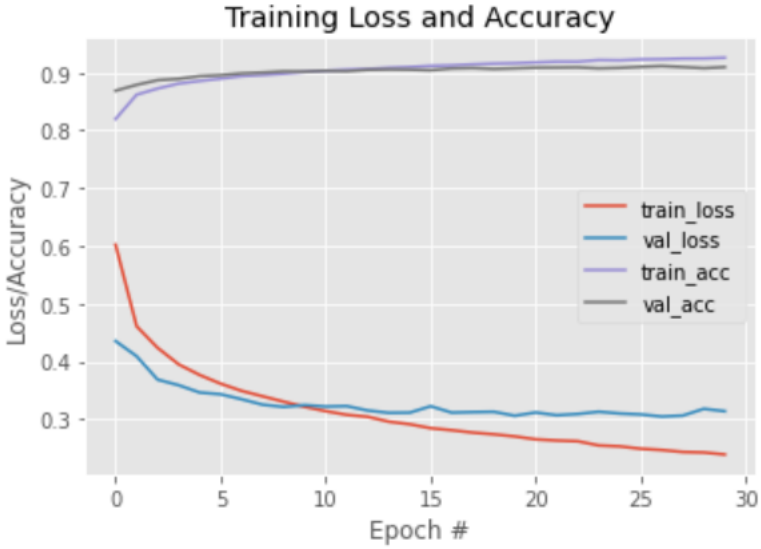
5 Layers: L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 100 Neurons L5 Softmax with dropout regularization

Drop-out 0.2 on every layer



- Over-fitting epoch: 11
- Accuracy before over-fitting: 96.60
- ```
  Epoch 11/30
  170000/170000 [==============================] - 10s 56us/sample - loss: 0.2675 - accuracy: 0.9163 - val_loss: 0.3109 - val_accuracy: 0.9060
  ```

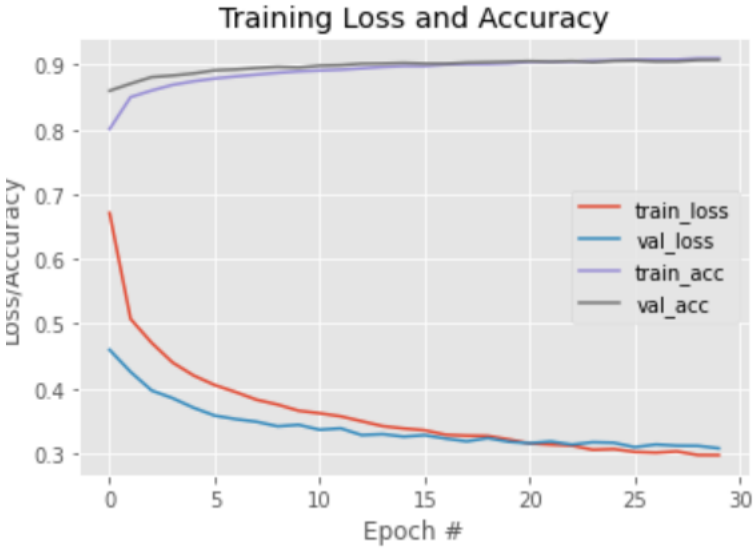
Drop-out 0.3 on every layer



- Over-fitting epoch: 8
- Accuracy before over-fitting: 0.9001

```
Epoch 8/30
170000/170000 [==============================] - 8s 47us/sample - loss: 0.3395 - accuracy: 0.8959 - val_loss: 0.3249 - val_accuracy: 0.9001
```
```

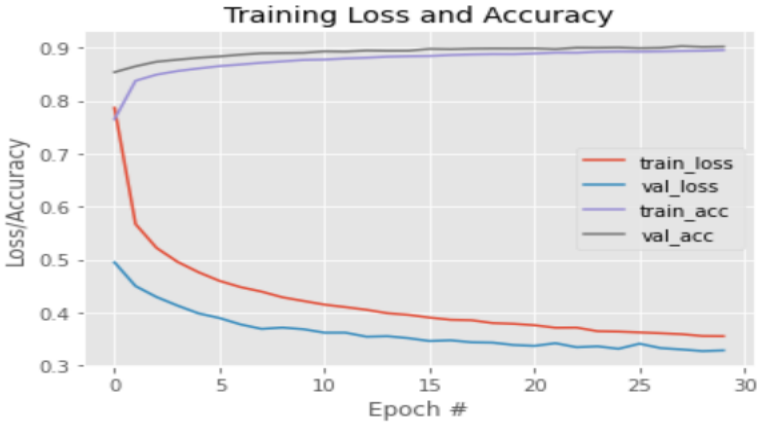
Drop-out 0.4 on every layer



- Over-fitting epoch: 20
- Accuracy before over-fitting: 90.46

```
Epoch 20/30
170000/170000 [==============================] - 8s 48us/sample - loss: 0.3215 - accuracy: 0.9022 - val_loss: 0.3179 - val_accuracy: 0.9046
```

Drop-out 0.5 on every layer

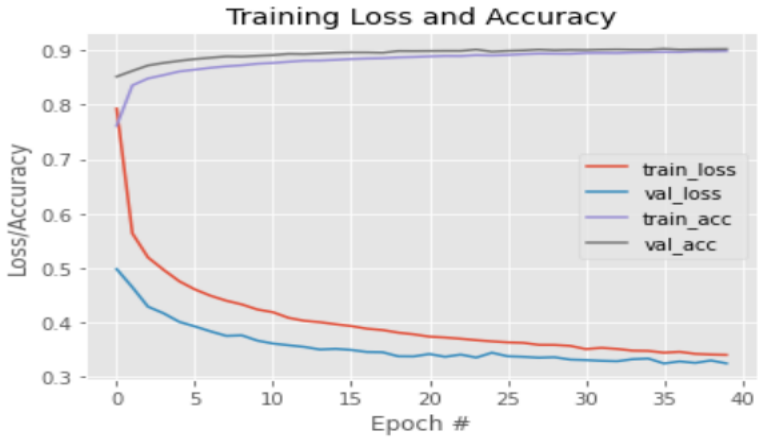


- Over-fitting epoch: No over-fitting
- Accuracy before over-fitting: 0.9018

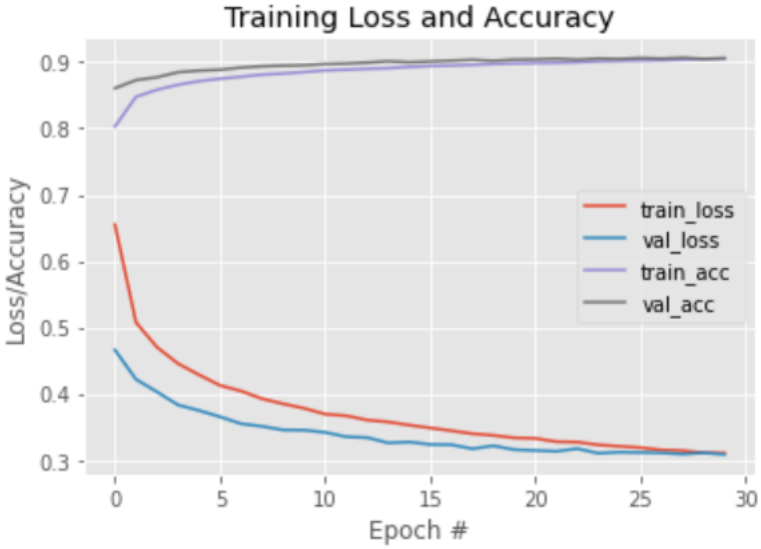
```
Epoch 30/30
170000/170000 [==============================] - 8s 47us/sample - loss: 0.3556 - accuracy: 0.8952 - val_loss: 0.3289 - val_accuracy: 0.9018
```

- Increasing the epochs to 40 (No over-fitting)

```
Epoch 40/40
170000/170000 [==============================] - 9s 51us/sample - loss: 0.3400 - accuracy: 0.8991 - val_loss: 0.3243 - val_accuracy: 0.9028
```



Drop-out 0.5 to 0.2 (Step -0.1) on every layer

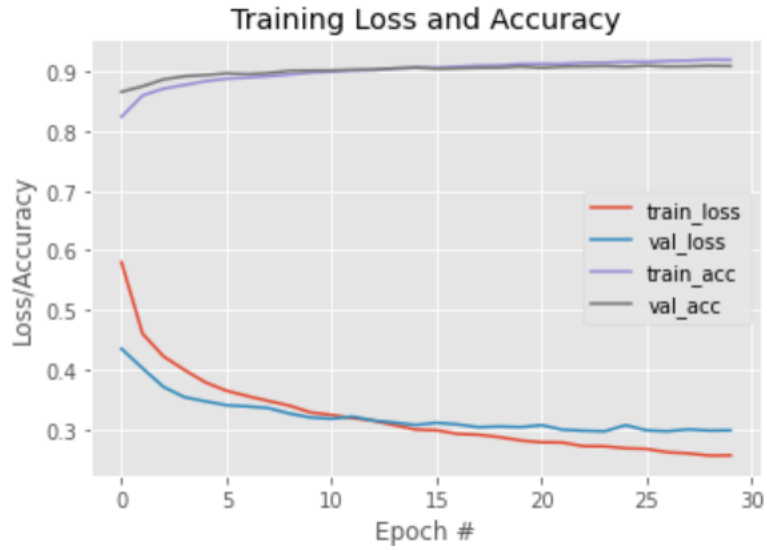



```
model=tf.keras.models.Sequential([layers.Dense(600, activation=tf.nn.relu, input_shape= (784,)),
                                  layers.Dropout(0.5),
                                  layers.Dense(400, activation=tf.nn.relu),
                                  layers.Dropout(0.4),
                                  layers.Dense(200, activation=tf.nn.relu),
                                  layers.Dropout(0.3),
                                  layers.Dense(100, activation=tf.nn.relu),
                                  layers.Dropout(0.2),
                                  layers.Dense(10, activation=tf.nn.softmax)])
```

- Over-fitting epoch: No over-fitting
- Accuracy before over-fitting: .9060

```
Epoch 30/30
170000/170000 [==============================] - 8s 47us/sample - loss: 0.3105 - accuracy: 0.9055 - val_loss: 0.3084 - val_accuracy: 0.9060
```

Drop-out 0.4 to 0.1 (Step -0.1) on every layer



- Over-fitting epoch: 12
- Accuracy before over-fitting: 0.9029

```
Epoch 12/30
170000/170000 [==============================] - 8s 47us/sample - loss: 0.3190 - accuracy: 0.9014 - val_loss: 0.3215 - val_accuracy: 0.9029
```

Network	Over-fitting epoch	Accuracy before over-fitting
4 Layers: Drop-out 0.2 on every layer	9	0.9058
4 Layers: Drop-out 0.3 on every layer	12	0.9073
4 Layers: Drop-out 0.4 on every layer	27	0.9078
4 Layers: Drop-out 0.5 on every layer	NA	0.9042
4 Layers: Drop-out 0.5 to 0.3 (Step -0.1) on every layer	21	0.9051
4 Layers: Drop-out 0.4 to 0.2 (Step -0.1) on every layer	12	0.9036
5 Layers: Drop-out 0.2 on every layer	11	96.60
5 Layers: Drop-out 0.3 on every layer	8	96.001
5 Layers: Drop-out 0.4 on every layer	20	96.46
5 Layers: Drop-out 0.5 on every layer	40	90.28
5 Layers: Drop-out 0.5 to 0.2 (Step -0.1) on every layer	30	0.9060
5 Layers: Drop-out 0.4 to 0.1 (Step -0.1) on every layer	12	90.29

Observations of the impact of Dropout

- Dropout decreases of mitigating over-fitting
- With larger dropout, we can train the network for more epochs
- Network: **L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 Softmax:** Dropout 0.4 on all layers performs best
- Network: **L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 100 Neuron L5 Softmax:** Dropout 0.2 on all layers performs best
- Overall for the given data Network: **L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 100 Neuron L5 Softmax performs better**
- Network: **L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 100 Neuron L5 Softmax:** Dropout 0.2 on all layers is overall the best setting

Dropout is very effective to overcome over-fitting

Part C: Batch Normalization

Sources:

- https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
- https://www.coursera.org/specializations/deep-learning
- https://www.youtube.com/watch?v=nUUqwaxLnWs
- https://www.youtube.com/watch?v=tNIpEZLv_eg
- https://arxiv.org/pdf/1502.03167v3.pdf
- https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/
- http://videolectures.net/icml2015_ioffe_batch_normalization/
- https://www.youtube.com/watch?v=dXB-KQYkzNU

What is batch Normalization?

Batch learning is nothing but mini-batch standardization of the inputs to a layer. For standardizing the data, we calculate the mean and variance of the mini-batch, the formula's given below. The concept of batch normalization was suggested by

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Source: https://arxiv.org/pdf/1502.03167v3.pdf (Original paper)

Why batch Normalization?

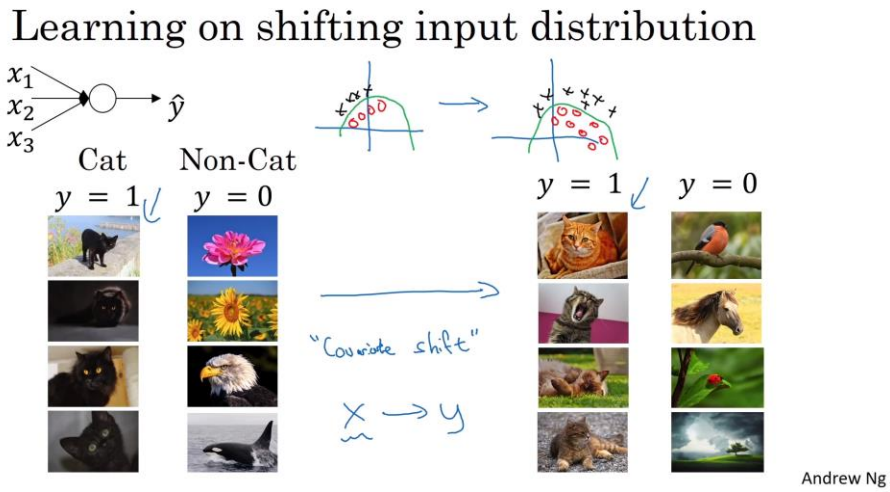
Training very deep networks with 10 to hidden layers can to time-consuming, with every hidden layer; the number of calculations gets increases. The process of batch normalization, standardize the mini-batches and this, in turn, helps the network to get trained in less number of epochs. The normalization has a stabilization effect of the mini-batch, it converts an elliptical contour to a more circular contour. The circular contour helps algorithms to learn faster and indeed helps in reduction in epochs and speeds up the training time.

Norm batch (batch Normalization) as slight regularization

As we standardize the mini-batch, we are using a comparatively mini sample to data for the actual data. This has a slight regularization effect on the model. We will need more regularization in our model, we can use L1, L2 or

Dropout regularization with Norm batch. Take a note, lower the size of mini-batch higher will be the effect of regularization. But when we lower the size of the mini-batch, it will increase the training time.

Covariance shift



Andrew Ng

Source: https://www.youtube.com/watch?v=nUUqwaxLnWs

Covariance shift is an effect where the model is trained on data to say black cats, and now if we try to predict cats of another color, the model will not perform properly. The curve of the function may be the same but the model the over-fit for the specific type of data. The norm batch can help us to generalize the model by standardizing the mini-batches.

The issue with Norm batch while validation data:

As we have discussed that we have to calculate the mean and variance based on the mini-batch while calculating the norm batch. But now imagine that when we will try to use a validation instance for prediction, what would be the outcome. How we can calculate the mean and variance of a mini-batch on a single instance. To tackle this issue, we can use the average of mean and variance through-out all the mini-batches from training data. And use it for calculation. (Formulas are given above)

Batch normalization on pre-training models:

Batch normalization is a great technique but for example, we want to use the VGG16 pre-trained model, the vgg16 model didn't use the batch norm technique. But even if the batch norm is not used for vgg16, we can apply the batch norm method on the layers. The batch norm can be applied to the activation values based on the batch size, this makes batch norm a yet more powerful technique.

Batch normalization in Keras:

In here, the axis in BatchNormalization method will be the features

```
[6]: from keras.models import Sequential
     from keras.layers import Dense, Activation, BatchNormalization

[10]: model = Sequential([
          Dense(16, input_shape=(1,5), activation='relu'),
          Dense(32, activation='relu'),
          BatchNormalization(axis=1),
          Dense(2, activation='softmax')
      ])
```