Full Name:
Peter Sunny Shanthveer Markappa

Student Number:
R00208303

Subject:
Deep Learning

Assignment:
01

Data:
03-April-2022

University:
Munster Technology University

Department:
Computer Science

Course:
MSc in Artificial Intelligence

Question 1_1_1

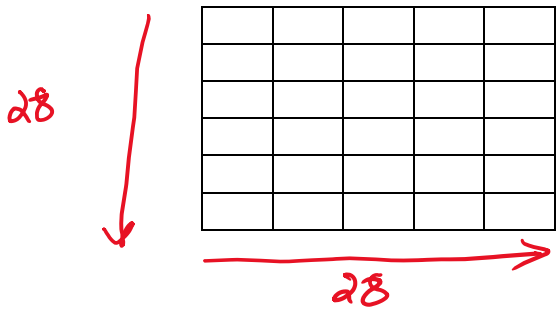
Implementation of Fashion-MNIST dataset using TensorFlow and Low Level API

1. Pre-processing and explanation of Data

Give Data: The shape of the given data is
Training Features: 60000, 784
Training Labels: 10, 60000

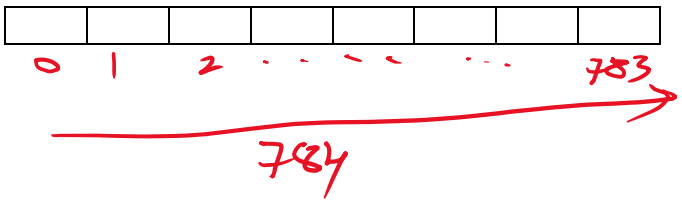
Test Features: 10000, 784
Test Labels: 10, 10000

It Means that totally there are 70,000 images amongst them 60 thousand are given for training and 10 for testing, where the test data is considered as real time data after training the model to check the accuracy that is how much percentage accurately does this model can predict the image. Each image pixel is 28x28 that tends to 784 pixels.



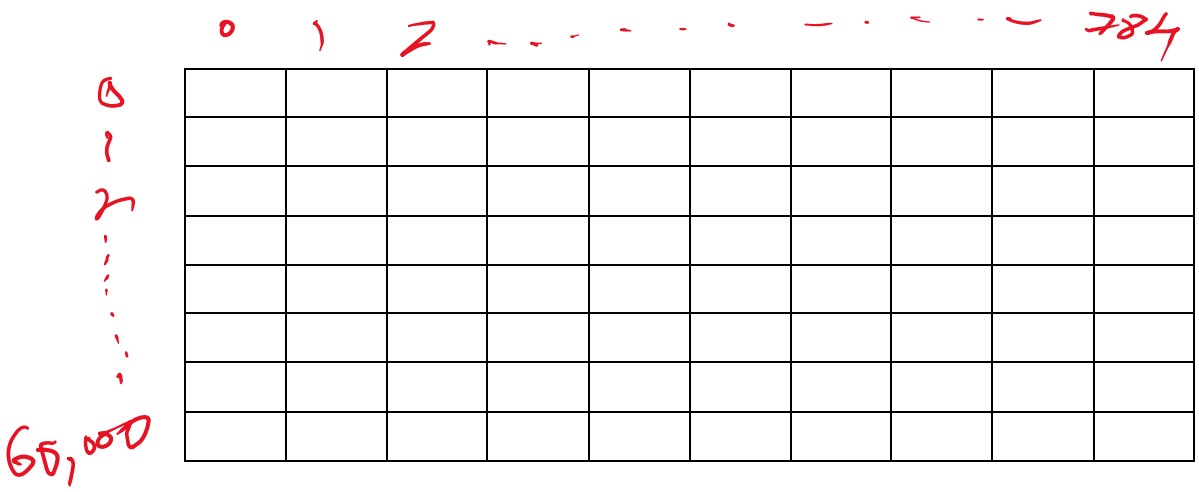
So now the data is in the matrix form which is not appropriate for training the model so we will vectorise this data into single row. i.e., all the 784 pixels will be in the single row or column

```
# reshape the feature data
tr_x = tr_x.reshape(tr_x.shape[0], 784)
te_x = te_x.reshape(te_x.shape[0], 784)
```



Or

So now we have 60000 rows and 784 columns that is each row consists of 784 pixels of 1 image



Normalisation:

```
# noramlise feature data
tr_x = tr_x / 255.0
te_x = te_x / 255.0
```

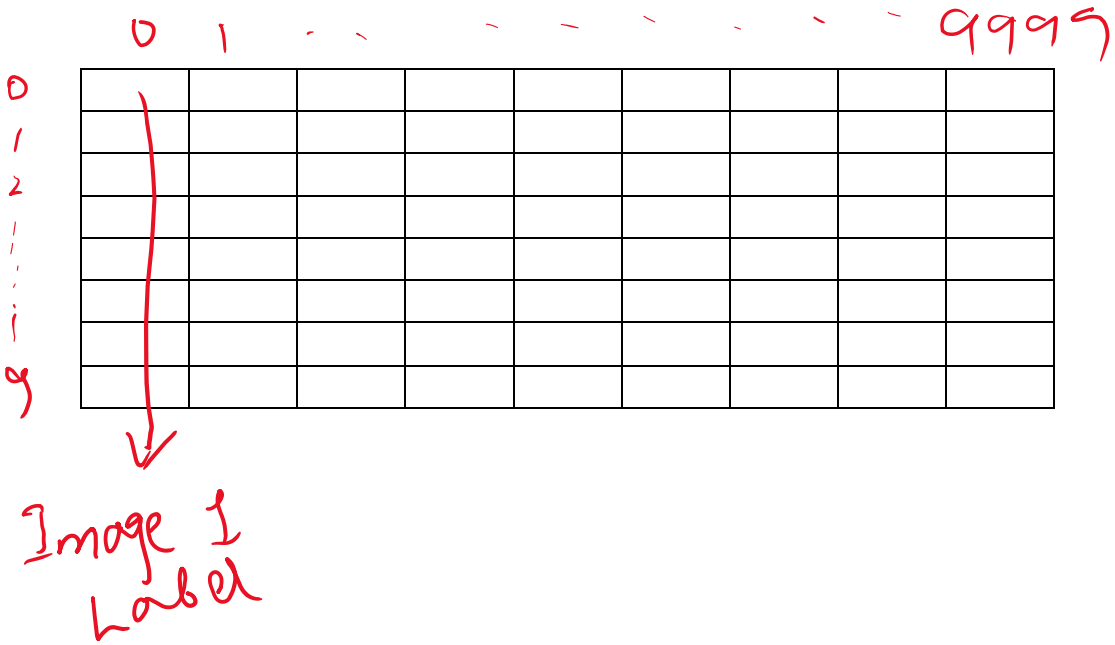
here each pixel of the image will be normalised whose value will be in between 0 to 255 and after normalising its value will be 0 to 1

Shape of Labels:
This is multi class classification that is there are 10 class from 0 to 9 so the training labels and test labels shape is
Training Labels: 10, 60000
Is has 10 rows and 60 thousand columns means here each column will have labels of 1 images

Total there are 60 thousand class label and the value will be 1 hot encoded which ever label is image is there then it will be 1 and rest will be 0

Example: 000100000 means here this column is having the label of 3rd class image

Test Labels: 10, 10000
Is has 10 rows and 10 thousand columns means here each column will have labels of 1 images

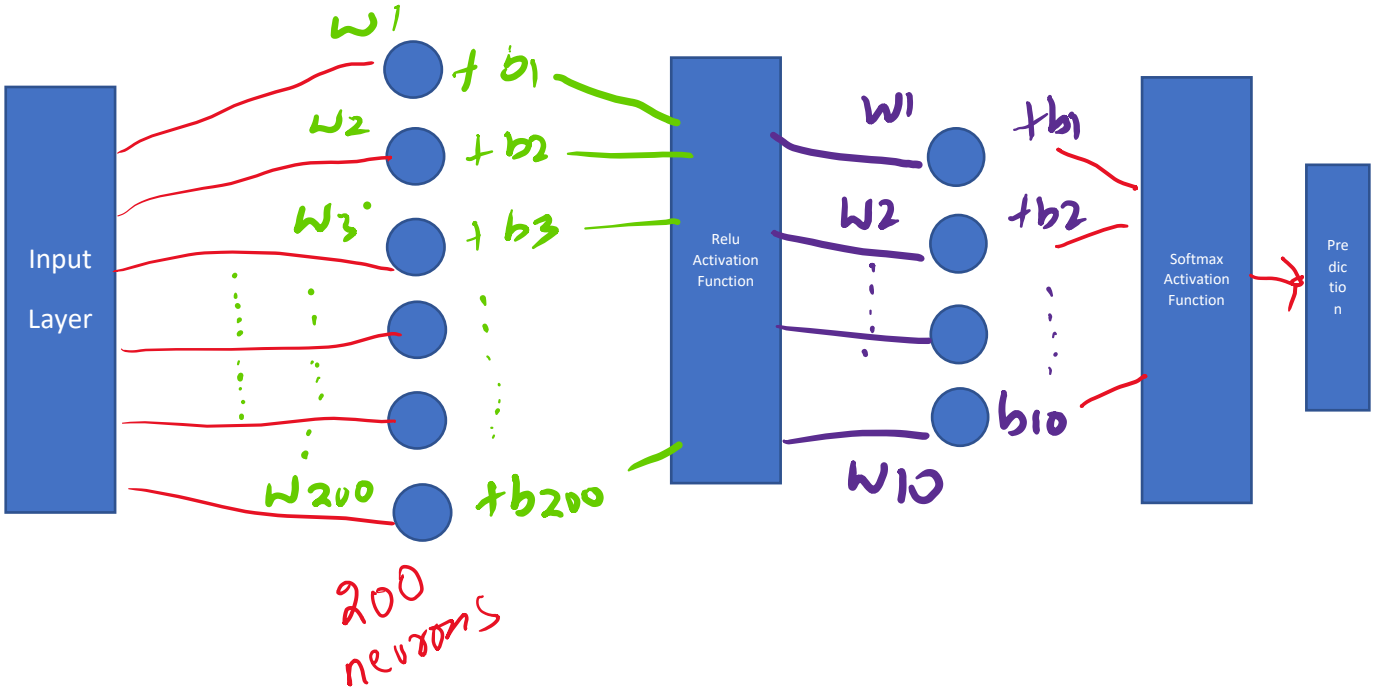


Total there are 60 thousand class label and the value will be 1 hot encoded which ever label is image is there then it will be 1 and rest will be 0

Example: 000100000 means here this column is having the label of 3rd class image

2. Explanation of Architecture for with code

Below is the architecture diagram



Creating the weights for 200 neurons which will be in the form of 200 x 784 and bias will be vector of 200 values for hidden layer and 10 weights with the shape of 10 x 784 for the output layer before sending it to the softmax, there reason for taking **10 weights and bias is** we have 10 class classification

```
    w1 = tf.Variable(tf.random.normal( [200, tr_x.shape[1]] , mean=0.0, stddev=0.05,
dtype=tf.dtypes.float64))
    b1 = tf.Variable(tf.random.normal([200], dtype=tf.dtypes.float64))

    w2 = tf.Variable(tf.random.normal([10, w1.shape[0]] , mean=0.1, stddev=0.08,dtyp
e=tf.dtypes.float64))
    b2 = tf.Variable(tf.random.normal([10], dtype=tf.dtypes.float64))
```

Splitting the Training Data into train and Validation

```
    tr_x_Training = tr_x[ : 48000, :]
    tr_x_Validation = tr_x[ 48000:, :]

    tr_y_Training = tr_y[ : , : 48000]
    tr_y_Validation = tr_y[ : , 48000 :]
```

The given 60 thousand data is divided into 80:20 ratio where 80% of the data will be used for the Training the model and 20% for the validation.

Forward Pass and updating the weights and bias

```
# Iterate our training loop
    for i in range(1000):

        # 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:

            # Training
            y_pred_training = forward_pass(tr_x_Training, w1, b1, w2, b2)

            # print("before cross entropy tr_y_Training",tr_y_Training.shape )
            # print("before cross entropy y_pred_training",y_pred_training.shape )

            training_Loss = cross_entropy(tf.transpose(tr_y_Training), y_pred_traini
ng)
            trainingLoss.append(training_Loss)

        gradients = tape.gradient(training_Loss, [w1, b1, w2, b2])

        training_accuracy = calculate_accuracy(tf.transpose(y_pred_training), tr_y_T
raining)
```

```
        trainingAccuracies.append(training_accuracy)

        adam_optimizer.apply_gradients(zip(gradients, [w1, b1, w2, b2]))
```

this model was repeatedly ran for 1000 iteration, for each time it enter the loop, it performs the forward pass, then the predicted data is sent to the cross entropy to find the loss and later the predicted value will be compared with labels to find the accuracy, below are the 3 function explanation

Forward Pass

```
def forward_pass(x, w1, b1, w2, b2):
    # We need to mutliply each training example by the weights and add bias
    y_pred = tf.matmul(x, tf.transpose(w1)) + b1

    relu_res = tf.keras.activations.relu(y_pred)

    y_pred1 = tf.matmul(relu_res,tf.transpose(w2)) + b2
    # print("Y_pred_1 before entering softmax", y_pred1.shape)
    act = softmax(y_pred1)

    return act
```

This function will receive the feature data with weights and bias, in this case we have 1 hidden layer and output layer,

1) It will multiply all the feature data 60000 x 784 with 784 x 200 weights, the reason is matrix in this shape is **matrix multiplication** must have col of first matrix equals to rows of second matrix.
2) First step it will multiply all the 60000 features with weights of 200 neurons and add each neuron result with its bias value, the resultant prediction will be sent to the relu activation function.
3) The resultant of relu activation will be 60000 x 10 so the weights for the output layer weights shape are 10x784, so the output layer has 10 neurons where all the 60000 result from the relu activation function will now be multiplied with the weights of relu result and add to bias of each neuron, this resultant will be sent to the softmax activation function.
4) The resultant from the softmax is considered as the predictions.

Softmax Activation:

```
def softmax(vector):
    e = tf.exp(vector, name='exp')
    return e / tf.math.reduce_sum(e, keepdims=1, name=None, axis=0)
```

resultant from the output layer is taken and softmax activation is applied

here it calculates the probability of each class

probability = exp(1) / exp(1) + exp(2) + exp(3) + ……. + exp(10)

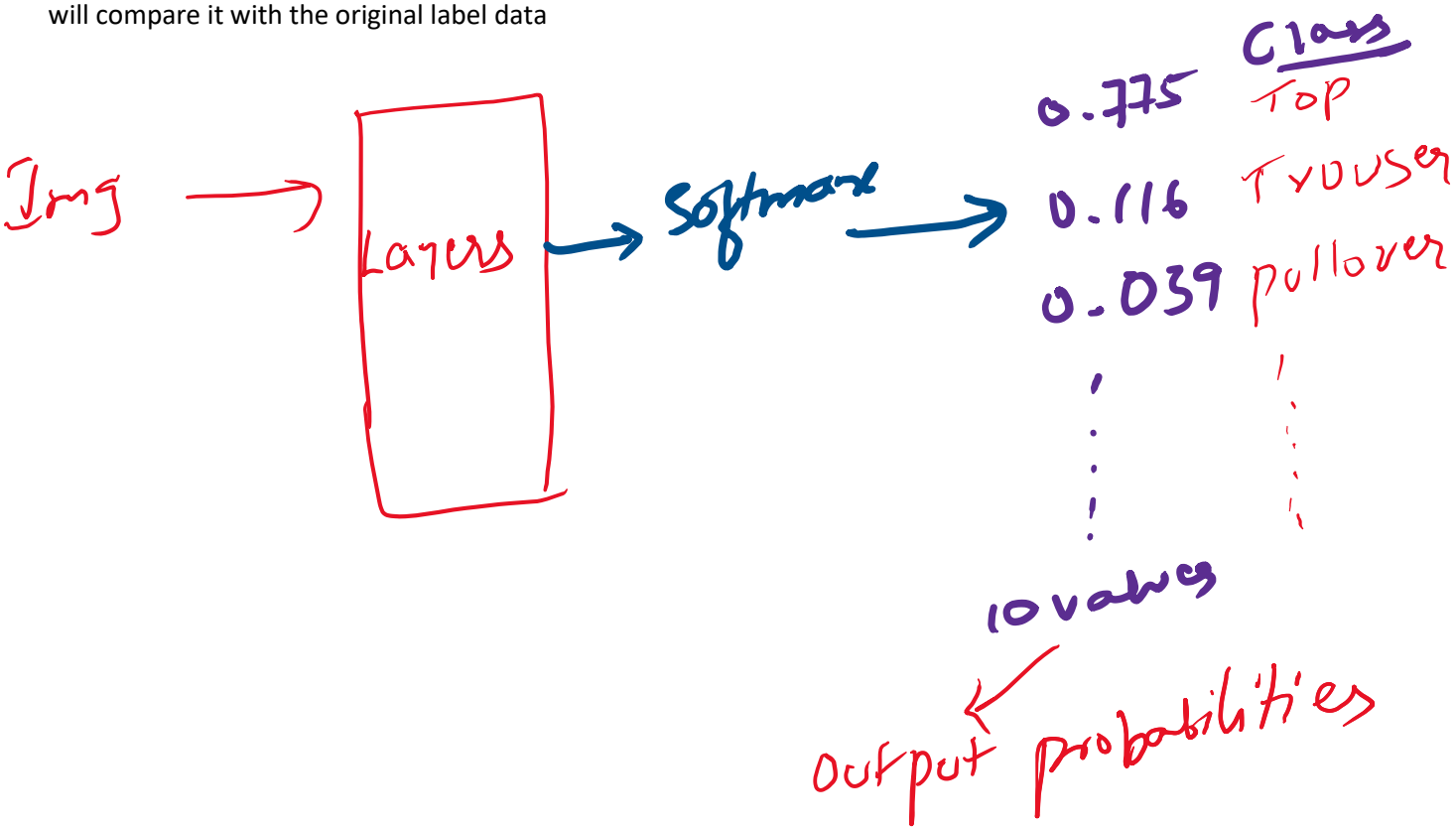
i.e., class1 / class 1 + class 2 + …… + class 10

this will generate the probability of the class like 0.7, 0.6,…

Cross_Entropy:

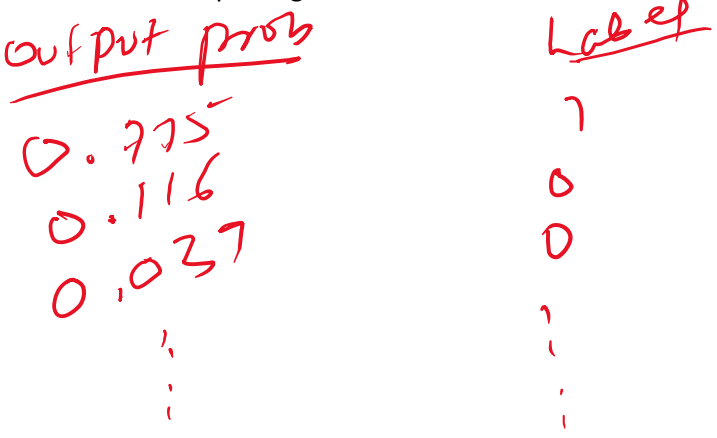
```
def cross_entropy(tr_y, y_pred):
    return (-
1/tr_y.shape[0]) * (tf.math.reduce_sum(tr_y * tf.math.log(y_pred) + (1 - tr_y) * (tf
.math.log(1 - y_pred))))
```

here this function takes the label of training or test data with the prediction made from the forward pass, and then it will compare it with the original label data



Now we have the output probabilities as **0.775, 0.116, 0.039,**... so on 10 values for each feature data.

The main objective of cross entropy is to take the output probabilities from the table and measure the distance from the true value which we are passing it as label data.



From the above explanation it states that the output for [1,0,0] is for class TOP and the model predicted 0.775, 0.116,...

Update of Weights and bias:

```
adam_optimizer.apply_gradients(zip(gradients, [w1, b1, w2, b2]))
```

using Adam optimiser this model is updating the weights and bias of both the layer weights and bias.

Gradient Tape:

```
 with tf.GradientTape() as tape:
```

```
gradients = tape.gradient(training_Loss, [w1, b1, w2, b2])
```

Gradient tape is used to backtrack the process, once we pass the loss and weights then this gradient tape will starts to find the partial derivative with respect to weights and bias

Validation DATA:

```
# # Validation
y_pred_validation = forward_pass(tr_x_Validation, w1, b1, w2, b2)
validation_Loss = cross_entropy(tf.transpose(tr_y_Validation), y_pred_validation)
validation_accuracy = calculate_accuracy(tf.transpose(y_pred_validation), tr_y_Valid
ation)
validationLoss.append(validation_Loss)
validationAccuracies.append(validation_accuracy)
```

the 20% of data that was split from the training will be used for validation, that is to check how the model is trained for each iteration and this is used after updating the weights and bias

```
if i % 50 == 0:
print ("Iteration ", i, "\n")
print(": Training Loss = ",training_Loss.numpy(), ": Validation Loss = ",validation_
Loss.numpy())
print( "Training Accuracies: ", training_accuracy.numpy(), "Validation Accuracies: "
, validation_accuracy.numpy())
```

Training loss Validation loss and training accuracy validation accuracy are printed after every 50 iteration

Test Data:

```
    # Test Data
    y_pred = forward_pass(te_x, w1, b1, w2, b2)
    currentLoss = cross_entropy(tf.transpose(te_y), y_pred)
    test_accuracy = calculate_accuracy(tf.transpose(y_pred), te_y)
    print ("Test Accuracy : ", test_accuracy)
```

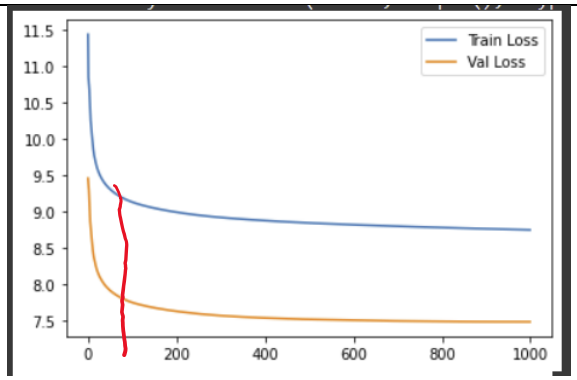
After completion of all the iteration, the model is evaluated with the test data which is considered as real time data, in this case there are 10000 images for testing the data.

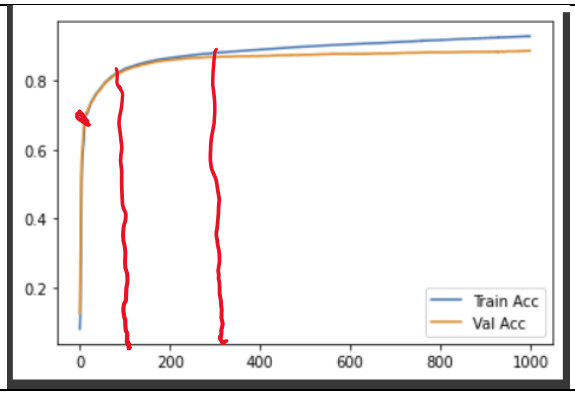
Evaluation report

For 80: 20 Split and 1000 iteration

```
: Training Loss =  8.764592025956068 : Validation Loss =  7.47884690313002
Training Accuracies:  0.9216041666666667 Validation Accuracies:  0.8836666666666667
----------------
Iteration  900

: Training Loss =  8.756490338185202 : Validation Loss =  7.477398157692122
Training Accuracies:  0.9245625 Validation Accuracies:  0.8844166666666666
----------------
Iteration  950

: Training Loss =  8.748814138761011 : Validation Loss =  7.476661652388859
Training Accuracies:  0.92725 Validation Accuracies:  0.8858333333333334
----------------
Test Accuracy :  tf.Tensor(0.8779, shape=(), dtype=float64)
```

The final accuracy of this model is 87 % after 1000 iteration





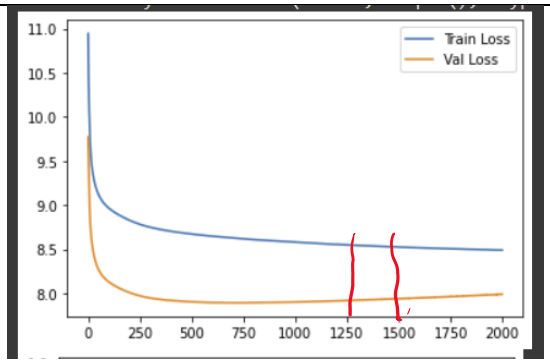
From above graph it can said that after 50 iteration the gap between the train and validation data is constant where as from accuracy grapy it can be said that after 100 iteration the accuracy of both train and validation starts to flatten and increasly slowly and after 300 iteration the gap started to built up.

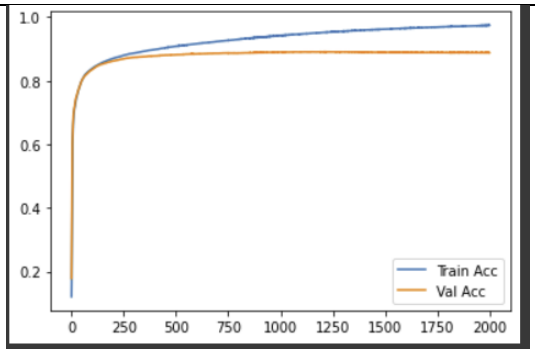
From the first graph we can say that that model is overfitted, by changing the weights and bias it can be reduced.

For 70: 30 Split and 2000 iteration

```
# Splitting the Training data into two parts
# 1) traning 2) Validation

tr_x_Training = tr_x[ : 42000, :]
tr_x_Validation = tr_x[ 42000:, :]

tr_y_Training = tr_y[ : , : 42000]
tr_y_Validation = tr_y[ : , 42000 :]
```

```
Iteration  1900

: Training Loss =  8.495600813596143 : Validation Loss =  7.980157159061262
Training Accuracies:  0.9719761904761904 Validation Accuracies:  0.8875555555555555
---------------
Iteration  1950

: Training Loss =  8.493263532372374 : Validation Loss =  7.9854037429552625
Training Accuracies:  0.9726428571428571 Validation Accuracies:  0.8879444444444444
---------------
Test Accuracy :  tf.Tensor(0.8786, shape=(), dtype=float64)
```

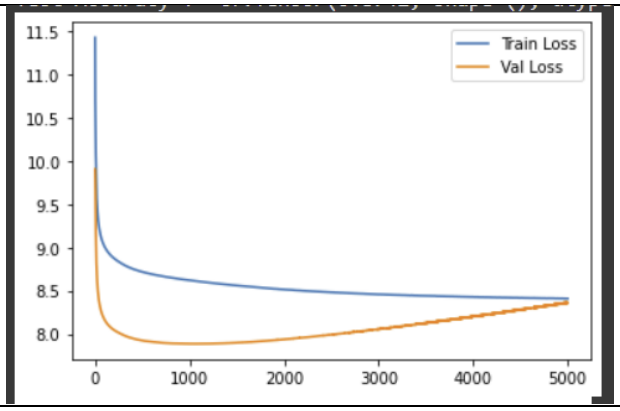


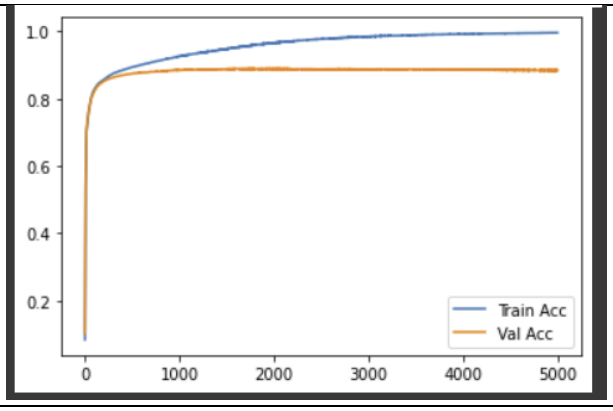


By increasing the iteration and change in the ratio of training and validation data after 1300 iteration the training loss and validation loss started to converge where as the accuracy started to diverge.

For 70: 30 Split and 5000 iterations

```
Iteration  4900

: Training Loss =  8.410506829007758 : Validation Loss =  8.347337718843677
Training Accuracies:  0.9954761904761905 Validation Accuracies:  0.8843333333333333
---------------
Iteration  4950

: Training Loss =  8.409363311022725 : Validation Loss =  8.349715781628571
Training Accuracies:  0.995452380952381 Validation Accuracies:  0.8838888888888888
---------------
Test Accuracy :  tf.Tensor(0.8742, shape=(), dtype=float64)
```





With 5000 iteration the data started to train properly with the less gap between the train loss and validation loss and the accuracy started be constant rate.

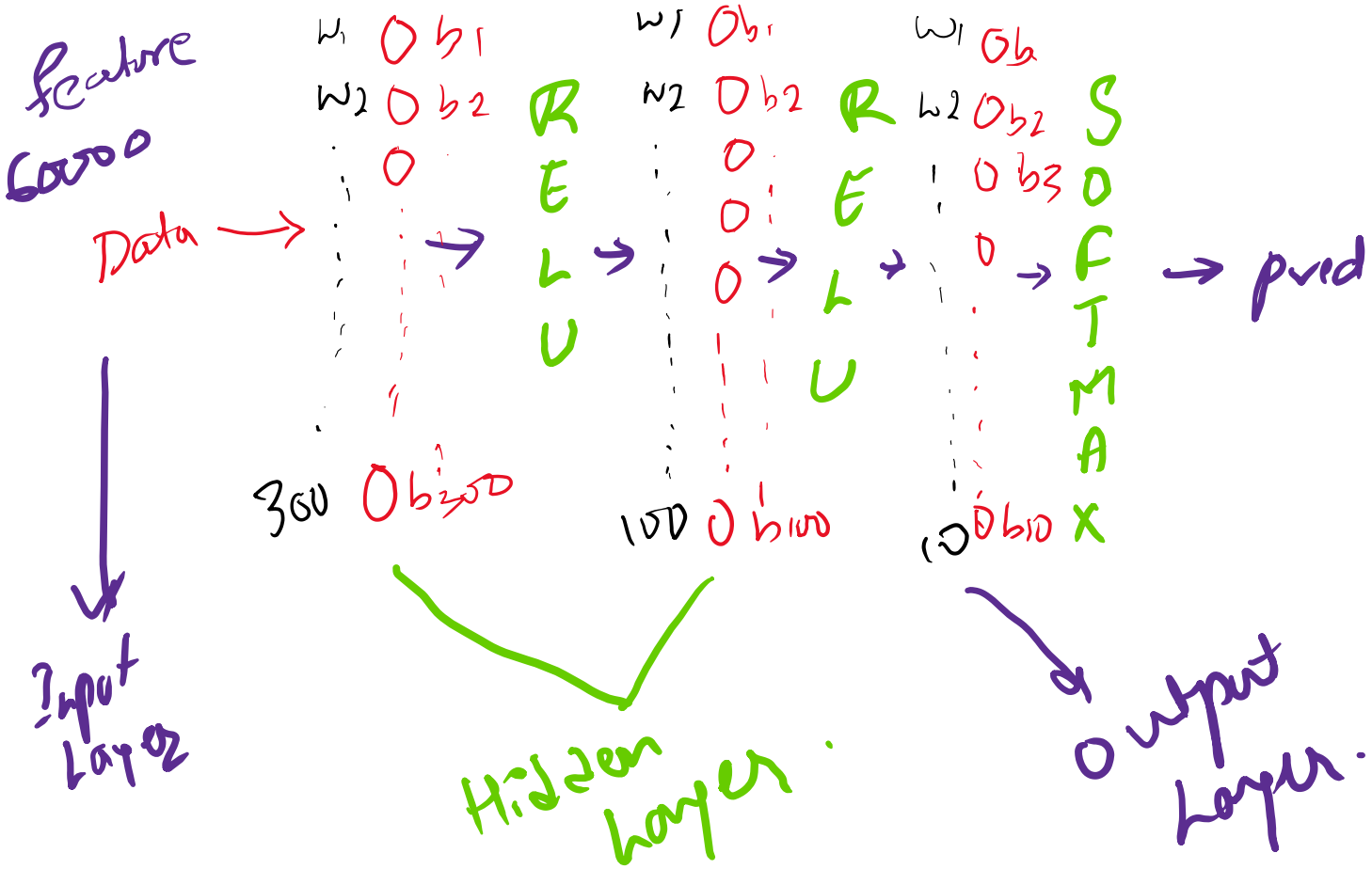
Question 1_1_2

Implementation of Fashion-MNIST dataset using TensorFlow and Low Level API

1. Pre-processing and explanation of Data

Note: all the pre-processing of data will be same as in question 1_1_1 and the working of each function is same is same as in question 1_1_1

Layer Architecture:



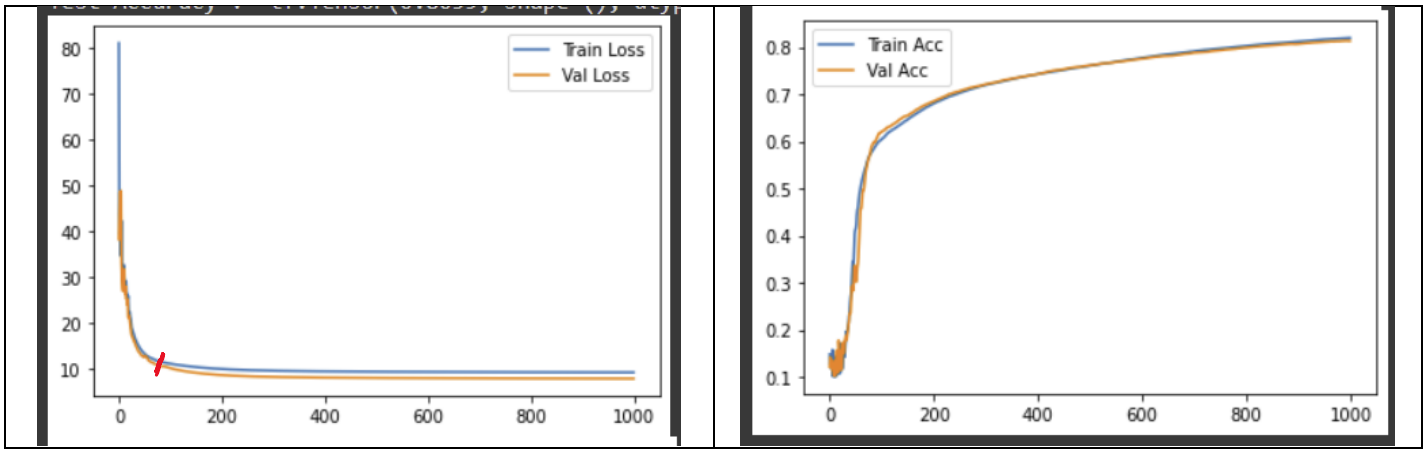
1) Feature data will be multiplied with 300 weights matrix of 300 x 784 and bias of 300 and the resultant will be sent to the relu activation function.
2) The result of first relu activation will be multiplied with the second hidden layer with 100 neurons with weights 100 x 300 and bias of 100 and this resultant will again be sent to relu activation function
3) The resultant from the second relu will be multiplied with the output layer with weights 10x100 and bias of 10 and the resultant will be sent to the softmax activation function.

After this all the process of updating the weights and bias are explained in the question 1_1_1.

Evaluation report

For 80: 20 Split and 1000 iterations and 2 hidden layers

```
Iteration  850

: Training Loss =  9.190561253289749 : Validation Loss =  7.828172207287447
Training Accuracies:  0.8076875 Validation Accuracies:  0.8035833333333333
---------------
Iteration  900

: Training Loss =  9.179598528512795 : Validation Loss =  7.817720890814004
Training Accuracies:  0.811875 Validation Accuracies:  0.807
---------------
Iteration  950

: Training Loss =  9.16962908426049 : Validation Loss =  7.8082115901634515
Training Accuracies:  0.8164166666666667 Validation Accuracies:  0.8113333333333334
---------------
Test Accuracy :  tf.Tensor(0.8059, shape=(), dtype=float64)
```



From the above graph it can be said that model is not overfitting as the training loss and validation loss is very minimum and it can be observed that the accuracy is going constant.

After the 30 iteration the gap between the loss started to built and it started to progress in constant manner, means the model is trained properly.

Note:

From above question 1_1_1 and Question 1_1_2 it can be said that the performance model is not only dependent on the number of neurons and weight but the performance of the model can also be increased by increasing the layer and iterations.

■ Next Page

Question 1_2_1

Drop out Layer

Neural network in the deep learning quickly overfit on the training data if the data is very small, Ensemble's methods in neural network can be used to reduce the overfitting of the data but it will increase the cost of computation.

A single model can be used to build the large neural network architecture by randomly dropping the neurons during training, this method is called **Drop Out**. Applying this method will also reduce the computation cost and also effective regularisation method to reduce the overfitting of training data along with that it will also improve generalisation error.

Data Pre-processing: this is explained in the question1_1_1

Creating the layers of network

```
# We need a coefficient for each of the features and a single bias value
# we create a column vector of weights all initialized to small random values
w1 = tf.Variable(tf.random.normal( [300, 784] , mean=0.0, stddev=0.05,dtype=tf.dtypes.float64))
b1 = tf.Variable(tf.random.normal([300], dtype=tf.dtypes.float64))

w2 = tf.Variable(tf.random.normal([100, 300] , mean=0.1, stddev=0.08,dtype=tf.dtypes.float64))
b2 = tf.Variable(tf.random.normal([100], dtype=tf.dtypes.float64))

w3 = tf.Variable(tf.random.normal([10, 100] , mean=0.2, stddev=0.03,dtype=tf.dtypes.float64))
b3 = tf.Variable(tf.random.normal([10], dtype=tf.dtypes.float64))
```

Here we are creating 3 weights matrix, the first weight will be used to multiply with the input feature data and drop out will be applied the resultant of the drop out will be multiplied with second weights w2 and added with bias the third weights are used for output layer before sending it to softmax activation function.

Applying the drop out: The drop is applied only on the training data not on validation data and test data.

```
def forward_pass(x, w1, b1, w2, b2, w3, b3, condition):
    # We need to mutliply each training example by the weights and add bias
    y_pred = tf.matmul(x, tf.transpose(w1)) + b1
    relu_res = tf.keras.activations.relu(y_pred)

    # --------- Drop out layer starts ----------
    if condition==0:
      probThreshold = 1 - 0.5
      neuronsize = relu_res.shape[0]
      trainingsize = relu_res.shape[1]

      dropmatrix = tf.cast(tf.Variable(tf.random.uniform([neuronsize, trainingsize],
 dtype=tf.dtypes.float64)) < probThreshold, tf.float64)
      relu_res = relu_res * dropmatrix
    # --------- Drop out layer ends -------------
```

```
    y_pred1 = tf.matmul(relu_res, tf.transpose(w2)) + b2
    relu_res1 = tf.keras.activations.relu(y_pred1)


    y_pred2 = tf.matmul(relu_res1,tf.transpose(w3)) + b3
    act = softmax(y_pred2)
    # act = tf.keras.activations.softmax(y_pred1)

    return act
```

once the training data is sent to the forward pass, the feature data will be multiplied with 300 neurons and resultant of each neuron is added with 300 bias value, this result will be then sent to the relu activation function, one we get the result of the relu activation function then drop out is applied on the training data, as this function is also used for validation data and test data the condition is checked before applying the drop out.

A probability of 0.5 is commonly used for retaining the output of each node in a hidden layer and a value close to 1.0, such as 0.8, for retaining inputs from the visible layer. A new hyperparameter is introduced that specifies the likelihood that the layer's outputs will be dropped out, or, conversely, the likelihood that the layer's outputs will be retained. The interpretation is a technical implementation detail that may differ from paper to code library. When making a prediction with the fit network, dropout is not used after training. Because of dropout, the network's weights will be larger than usual. As a result, before finalizing the network, the weights are scaled by the selected dropout rate. The network can then be used to make predictions as usual.

Geoffrey Hinton, et al. in their 2012 paper that first introduced dropout titled "Improving neural networks by preventing co-adaptation of feature detectors" applied used the method with a range of different neural networks on different problem types achieving improved results, including handwritten digit recognition (MNIST), photo classification (CIFAR-10), and speech recognition (TIMIT). [1]

Steps for dropout:

The first step is to randomly generate the Boolean array that will dictate which neurons are removed from consideration for each training example

1. Generate an array of random numbers between 0-1

   0.5 0.9 0.5 0.1
   0.2 0.1 0.2 0.7
   0.8 0.2 0.6 0.8

2. Execute conditional statement comparing random number to 1- dropout probability

   0.4 **0.9** 0.3 0.1
   0.2 0.1 0.2 **0.6** < 0.5 (because in this model 0.5 is used)
   0.4 0.2 **0.6** 0.4

3. The conditional statement returns a Boolean array

   T F T T
   T T T F
   F T T F

4. Now we will multiply the random number with the generated array

0.4	**0.9**	0.3	0.1		T	F	T	T
0.2	0.1	0.2	**0.6**	*	T	T	T	F
0.4	0.2	**0.6**	0.4		F	T	T	F

Result

0.4	**0**	0.3	0.1
0.2	0.1	0.2	**0**
0.4	0.2	**0**	0.4

5. Finally, the resultant will be rescaled by dividing by the drop out probability

0.4/0.5	**0**/0.5	0.3/0.5	0.1/0.5
0.2/0.5	0.1/0.5	0.2/0.5	**0**/0.5
0.4/0.5	0.2/0.5	**0**/0.5	0.4/0.5

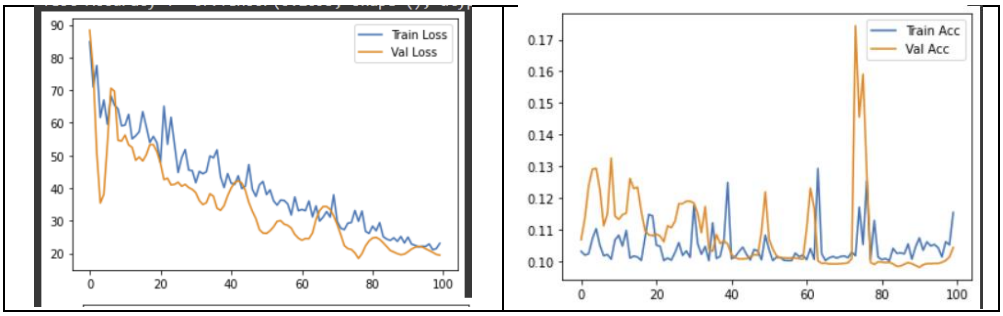
After dividing we can see that 3 neurons are dropped out like in this model **50% of the neurons will be dropped before moving to next layer.**

Then this result will be now multiplied with second hidden layer.

Evaluation report

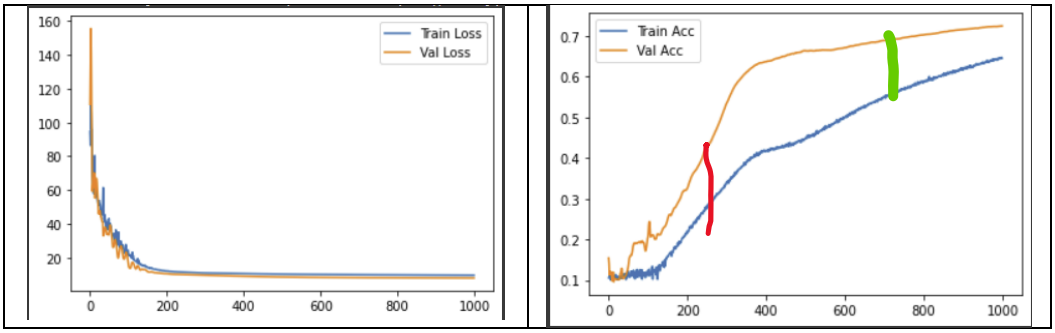
1. With 100 iterations, and 80:20 split in data

```
Iteration  0

: Training Loss =  84.84036095764985 : Validation Loss =  88.3710310311645
Training Accuracies:  0.10327083333333334 Validation Accuracies:  0.107
----------------
Iteration  50

: Training Loss =  38.02659073130758 : Validation Loss =  26.113306655443264
Training Accuracies:  0.10385416666666666 Validation Accuracies:  0.10716666666666666
----------------
Test Accuracy :  tf.Tensor(0.1095, shape=(), dtype=float64)
```



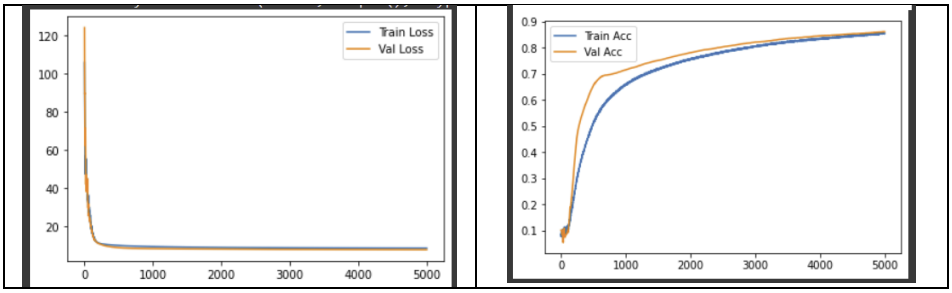
2. With 1000 iterations, and 80:20 split in data

```
Iteration  900

: Training Loss =  9.795763835023667 : Validation Loss =  8.113925609755212
Training Accuracies:  0.6164583333333333 Validation Accuracies:  0.7160833333333333
----------------
Iteration  950

: Training Loss =  9.745192843917302 : Validation Loss =  8.08529848241922
Training Accuracies:  0.6362291666666666 Validation Accuracies:  0.7215833333333334
----------------
Test Accuracy :  tf.Tensor(0.7193, shape=(), dtype=float64)
```



3. With 5000 iterations, and 70:30 split in data

```
Iteration  4800

: Training Loss =  8.785068482265094 : Validation Loss =  8.00642392831713
Training Accuracies:  0.849547619047619 Validation Accuracies:  0.8576666666666667
----------------
Iteration  4900

: Training Loss =  8.778925546506413 : Validation Loss =  7.999720721839722
Training Accuracies:  0.8528571428571429 Validation Accuracies:  0.86
----------------
Test Accuracy :  tf.Tensor(0.8528, shape=(), dtype=float64)
```



From all the three different graphs which was applied with different iteration and changing the ratio of the training and validation data, it can be said that increasing the number of iteration will reduce the distance between the training loss and validation loss.

From second graph with 1000 iteration we can clearly see that the accuracy gap started to increase after 200 iteration and reduce gap after 700 iterations, so same model was ran for 5000 iteration by changing the percentage of training and validation data and from third set of graph we can clearly see that the loss between training and validation is very less and even the accuracy gap is increased in the beginning of the iterations and after 1300 iteration it started to reduce its gap, it means that by using drop out regularisation we can reduce the overfitting of data by comparing Question1_1_1 and Question 1_1_2, where the feature data overfitting.

By building this model it can be stated that from question 1 the data was overfitting and it was reduced by applying the drop out technique.

■ Next Page

Question 1_3_1
Mini Batch

Mini Batch is technique where the data will split into small chunks and sent into the neural network for training, in this model the data is divided into 10 small set that is each mini batch will have 6000 features of data.

```
batch_size = 10
```

Steps

1. The training features and its labels are concatenated before shuffling and splitting

```
# Concatenate the training data and its labels before splitting
transpose_y = tr_y.T
feature_x_y = np.concatenate((tr_x, transpose_y), axis=1)
```

2. Concatenated data is then shuffled and it is divided by 10

```
shuffl_data = np.random.shuffle(feature_x_y)
each_batch = feature_x_y.shape[0] / batch_size
print("Each Batch Size", each_batch)
```

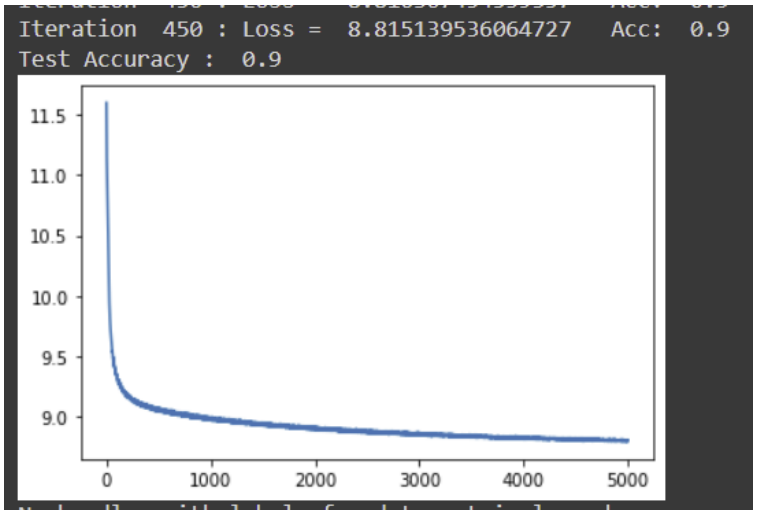
3. Number of iterations (epochs) and inside each iteration the inner loop runs for 10 times. There the data is again split into training features and training labels and then sent for the forward pass and the process will be same like previous question, but here 60000 data is sent into 10 batches i.e., 6000 first this data will pass the hidden layers they will be multiplied with weights and bias and in the output layer is goes to SoftMax activation, then the send batch of 6000 data will be sent, this does not means the next batch of the data is sent only after the previous batch SoftMax calculation, instead the data will be one after the other. In this way 10 set of mini batches of data was trained for 1000 iteration.

```
# Iterate our training loop
for i in range(num_Iterations):
 for j in range(batch_size):
   batch_data = feature_x_y[j*int(each_batch) : (j* int(each_batch)) + int(each_batch), :]
   train_x = batch_data [ : , : 784]
   train_y = batch_data [ : , 784:]

   train_x = tf.convert_to_tensor(train_x, tf.dtypes.float64)
   train_y = tf.convert_to_tensor(train_y, tf.dtypes.float64)

   # 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 = forward_pass(tr_x, w1, b1, w2, b2)
       y_pred = forward_pass(train_x, w1, b1, w2, b2, w3, b3)
       currentLoss = cross_entropy(train_y, y_pred)
       # print("Current Loss", currentLoss)
       trainingLoss.append(currentLoss)
```

```
        gradients = tape.gradient(currentLoss, [w1, b1, w2, b2, w3, b3])
        accuracy = calculate_accuracy(y_pred, tf.transpose(train_y))
        adam_optimizer.apply_gradients(zip(gradients, [w1, b1, w2, b2, w3, b3]))
```

Evaluation report

With 500 iterations



After 500 iteration this model has achieved accuracy of 90% and the loss has gradually reduced

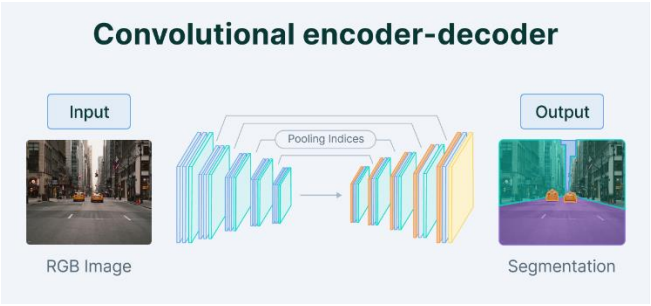
****

Question 1_4_1

Part 1: Implementation of Auto-Encoder:

What is Autoencoder?
It is type of ANN which is built to learn the encodings of data in unsupervised manner. This network architecture is also called **Bottle neck architecture.**

The main aim of it is to learn lower dimension encodings for higher dimension data, basically for dimensionality reduction, all this happens after training the network to capture the important parts of the image.



Implementation and Explanation

1. Forward pass

```
def forward_pass(x, w1, b1, w2, b2, w3, b3, w4, b4, w5, b5, w6, b6):
    # We need to mutliply each training example by the weights and add bias
    y_pred1 = tf.matmul(x, tf.transpose(w1)) + b1

    relu_res1 = tf.keras.activations.relu(y_pred1)
    # print("relu_res1 = ", relu_res1.shape)

    y_pred2 = tf.matmul(relu_res1, tf.transpose(w2)) + b2
    relu_res2 = tf.keras.activations.relu(y_pred2)
    # print("relu_res2 = ", relu_res2.shape)


    y_pred3 = tf.matmul(relu_res2, tf.transpose(w3)) + b3
    relu_res3 = tf.keras.activations.relu(y_pred3)
    # print("relu_res3 = ", relu_res3.shape)

    y_pred4 = tf.matmul(relu_res3, tf.transpose(w4)) + b4
    relu_res4 = tf.keras.activations.relu(y_pred4)
    # print("relu_res4 = ", relu_res4.shape)

    y_pred5 = tf.matmul(relu_res4, tf.transpose(w5)) + b5
    relu_res5 = tf.keras.activations.relu(y_pred5)
    # print("relu_res5 = ", relu_res5.shape)

    y_pred6 = tf.matmul(relu_res5,tf.transpose(w6)) + b6
    act = Sigmoid(y_pred6)
    # print("sigmoid activation shape = ", act.shape)

    return act
```

The first layer of the network consists of 128 neurons the resultant will be sent to the relu activation function, followed by second hidden layer with 64 and third with 32 neurons, so here we can see that the size of the neurons are reducing and middle layer is having 32 neurons which are minimum in this architecture, this itself is the **Bottle neck layer** which contains compressed knowledge representations and **it is most important part of the network**. Followed by this again neurons will start increasing with 64, 128 and the output layer consists of 784 neurons, the prediction of this will be sent to sigmoid activation function.

2) Mean absolute error

```
def mean_absolute_error(tr_y, y_pred):
    return (1/tr_y.shape[0]) * (tf.reduce_sum(tf.abs(tr_y - y_pred)))
```

Working of mean absolute error: explanation is given by taking an example

Here class refers to which class the given image belongs

Actual Cost:
Feature 1 data– class 3
Feature 2 data – class 6
Feature 3 data – class 2
Feature 4 data – class 0

Predicted Cost:
Feature 1 data– class 1
Feature 2 data – class 5
Feature 3 data – class 3
Feature 4 data – class 2

Step 1: mean average error
Prediction error = actual cost – predicted cost

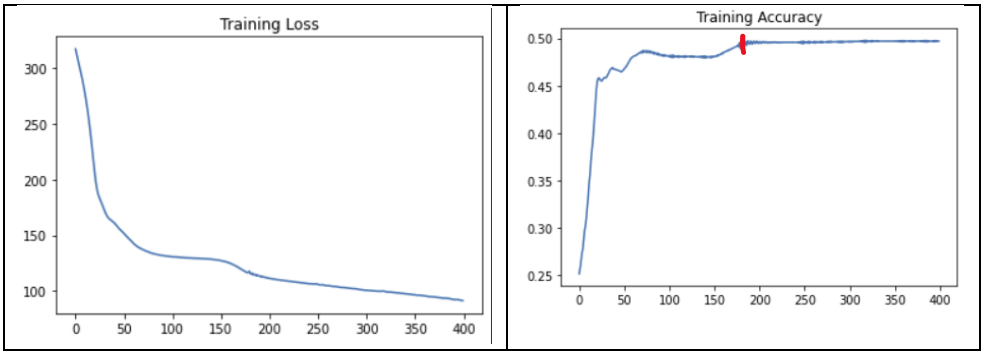
Feature 1 data
Actual class 3, predicted class 1
Absolute Error 1 = |Error| (Absolute value or positive value of error
---
Feature 2 data – class 6
Actual class 6, predicted class 5
Absolute Error 2 = |Error| (Absolute value or positive value of error
---
Feature 3 data – class 2
Actual class 2, predicted class 3
Absolute Error 3 = |Error| (Absolute value or positive value of error
---
Feature 4 data – class 0
Actual class 0, predicted class 2
Absolute Error 4 = |Error| (Absolute value or positive value of error

Considering n = 4
Mean Absolute error = (Absolute Error 1 + Absolute Error 2 + Absolute Error 3 + Absolute Error 4)
MEA = 2 + 1 + 1 + 2 / 4
MEA = 1 (even though the calculation is 1.5 we are considering absolute value)
so, it can be said that out model predictions are off by approximately 1

Evaluation Report: Predicted image with Actual image

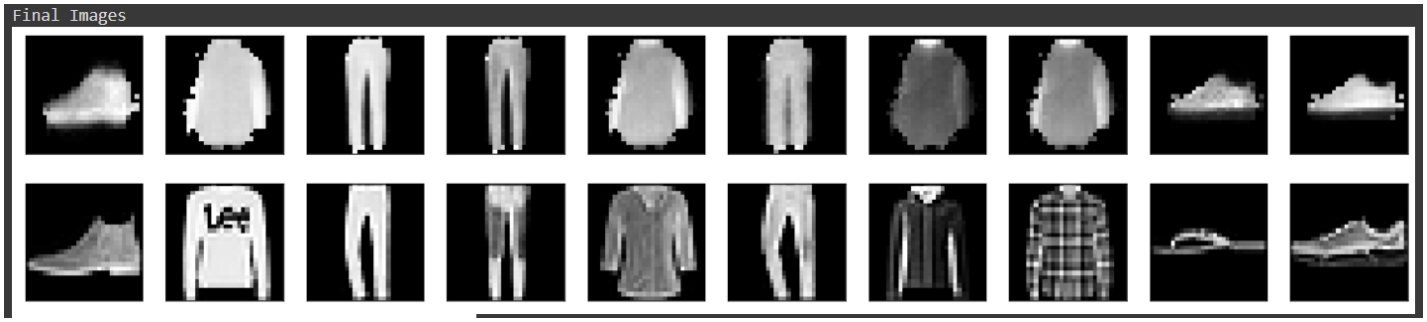
```
Iteration  0
 : Training Loss =  309.40134   Training Acc:  0.2853596726190476
Iteration  50
 : Training Loss =  151.13014   Training Acc:  0.4681441326530612
Iteration  100
 : Training Loss =  131.99744   Training Acc:  0.4807202593537415
Iteration  150
 : Training Loss =  129.03143   Training Acc:  0.481669068877551
Iteration  200
 : Training Loss =  113.89745   Training Acc:  0.49605072278911566
Iteration  250
 : Training Loss =  107.40083   Training Acc:  0.4975918579931973
Iteration  300
 : Training Loss =  103.12847   Training Acc:  0.4973389668367347
Iteration  350
 : Training Loss =  99.43734   Training Acc:  0.4981495535714286
: Test Loss =  95.46983   Test Acc:  0.4981638605442177
```

With 400 iteration this model had achieved the accuracy of around 50%, so this model can be trained by increasing the number of iterations.



From above graph we can see that loss is decreasing and accuracy is increasing, after 200 iteration the accuracy is becoming constant that is there is not much spike in the increase of accuracy.

First line images are predicted images from the model built and below are actual image



****

Question 1_4_1

Part 2: Research and Short report on auto encoder and variation encoder

1) Main application and potential disadvantages of traditional auto encoders.

a) Introduction and working of traditional Auto-encoder

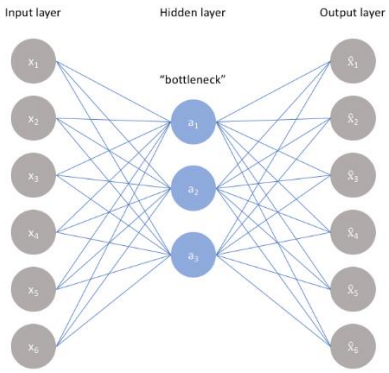


Image source: https://www.v7labs.com/blog/autoencoders-guide#:~:text=An%20autoencoder%20is%20an%20unsupervised,even%20generation%20of%20image%20data.

Input Layer: This layer is called as encoder layer, where the input data will be compressed by the module and convert it into encoded representation.

Hidden Layer: This layer is called as Bottle neck, there reason for saying this architecture as bottle neck because if we see the half shape of the network it is in the shape of bottle. This layer will have compressed representation of the data.

Output Layer: This layer is called as output layer it will decompress the representation of knowledge from hidden layer and reconstructs the data back from its encoded form. The predicted output will be then compared with real data.

Before we could move to the disadvantages of auto-encoder let us see the math behind the networks, the network is split into two segment the encoder and the second is decoder.

$$\phi : \mathcal{X} \to \mathcal{F}$$
$$\psi : \mathcal{F} \to \mathcal{X}$$
$$\phi, \psi = \underset{\phi,\psi}{\arg\min} \|X - (\psi \circ \phi)X\|^2$$

Original data X will be denoted by encoder function ϕ, to the hidden space which is called as bottle neck. Then decoder function comes into action which is denoted by ψ, this will maps the hidden space F which is at the bottle neck to the output layer. *In generally we are basically trying to re-create the same original image after doing generalised non-linear compression.*

After being passed through an activation function, the encoding network is represented by the standard neural network function, where z is the latent dimension.

$$\mathbf{z} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Likewise, the decoding network can be represented similarly, but with different weight, bias, and possibly activation functions.

$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}')$$

The loss function can then be written in terms of these network functions, and we will use this loss function to train the neural network using the standard backpropagation procedure.
Below is the loss function which is written in terms of network function, and this will be use to train the neural network using standard back propagation procedure

$$\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2$$

If we look at the working of auto-encoder it can seen that it is neither supervise or unsupervised learning instead it can be called as self-supervised learning, the reason for this is the input and output image are same. The most important thing in auto-encoder is to select the encoder and decoder function such that it must use minimum requirement to encode the input image and later regenerate from the other end.

If few neurons are used in the bottle neck layer, then the capacity to for the decoder while recreating the image will be limited, therefore the output images will be blur or cannot be recognised. In other case if many neurons are used then there is little point in using compression at all.

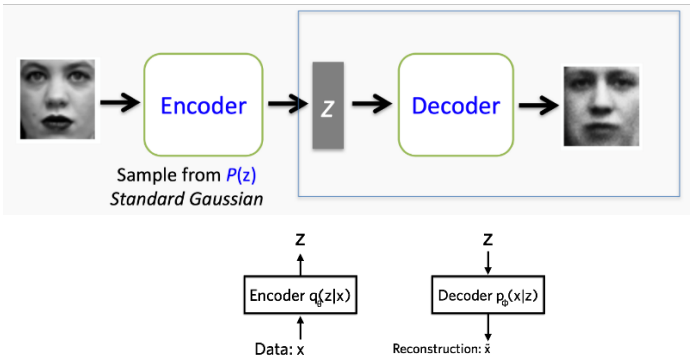
Day to day Application: we watch the videos in the youtube, Netflix and many other streaming sites, the data that is sent to the user system will be in compressed form, once it reaches to user system that video will go through the decompression algorithm and later it will be displayed to the user. This concept is also used in compression of files but in live streaming or when watching videos streaming algorithms are used.

Disadvantages:

1) The auto-encoder has to be trained, this needs lot and lots of data after that pre-processing time, hyper-parameter tuning, model validation all this has to be done before even building the real model.
2) An autoencoder learns how to efficiently represent a manifold that contains the training data. If your training data is not representative of your testing data, you end up obscuring rather than clarifying information. I'm currently working on such a problem. I wanted to use an autoencoder, but it would reduce performance.
3) An autoencoder learns to capture as much data as possible rather than as much relevant data as possible. That is, if the information most relevant to your problem constitutes only a small (in magnitude) portion of the input, the autoencoder may discard a significant portion of it.
4) Using an autoencoder also eliminates any interpretability your model may have had. This may or may not be significant to you.

2) Variational auto encoder

VAE uses the traditional autoencoder architecture to learn the data generation distribution. This allows to get a random sample from the latent space. These random samples can then be decoded in the decoder network to generate a unique image with properties similar to those used for network training.



We know that Bayesian statistics will recognize that the encoder is learning an approximation of the posterior distribution. This distribution is usually difficult to analyze because there is no closed-form solution. This means that you can use either a computationally intensive sampling method such as Markov Chain Monte Carlo (MCMC)

or a variational method. As expected, the variational autoencoder uses variational inference to generate an approximation to the posterior distribution.

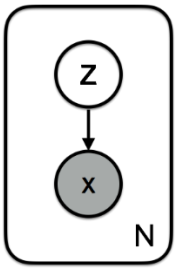
A variational autoencoder is made up of an encoder, a decoder, and a loss function in neural net language. The input data is 'x', in the minst fashion data set image is of 28x28 pixel, therefore here x is also of 28x28 pixel image which will be having its own weight and the bias. The encoder will encodes the 784 pixel image into smaller dimension in the latent space 'z' which is mentioned as bottle neck. The encoder will gives the output q(z/x) which is gaussian probability density. The decoder is another neural net, where as its input is compressed image and this will also have weights and bias different from the any other layers, this decoder is represented by p(x/z). considering the mnist fashion dataset all the pixel values will be between 0 or 1. Here Bernoulli distribution is used to represent the single pixel. The decoder get the input parameter of fashion dataset z and it will output 784 bernoulli parameter, where one for each of the 784 pixels in the image. Here decoder will decodes the real value of x into 784 pixel between 0 and 1.

The loss function used in the variational auto encoder is negative log of likelihood with regularizer.

$$l_i(\theta,\phi)=-\mathrm{E}_{z\sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)]+\mathrm{KL}(q_\theta(z|x_i)||p(z))$$

here the first term is reconstruction loss of i-th feature data point.

The variational autoencoder under the probability model contains a specific probability model for the data x and the latent variable z. The joint probabilities of the model can be described as:
P(x,z) = p(x/z) p(z) this can also be represented as



Difference between Variational Encoder and Traditional Encoder:

Deep neural autoencoders and deep neural variational autoencoders have similar architectures but perform distinct functions. Autoencoders frequently work with numerical or photographic data. Data visualization, statistics denoising, and statistics anomaly detection are three common applications for autoencoders. Variational autoencoders frequently work with image or textual content (document) statistics. The most commonplace use of variational autoencoders is to generate new image or textual content statistics.

Considering the minst data set with each image is of 28x28 pixel.
Deep neural AEs are trained by having them predict their own input. An AE for MNIST has 784 input nodes and 784 output nodes, with one for each pixel. To illustrate the concept and keep the diagram size reasonable, the image below uses only 5 input/output nodes.

In most cases, the AE in the center of the example is a vector with two values, 0.0-1.0. These two values, along with the AE weights (which are not shown in the figure), represent a compressed numerical representation of the input image. A 784 pixel value in the range 0-255, for example, can be expressed as (0.5932, 0.1067). Because the two values can be represented in a xy chart, they can be used to generate an image visualization. This application does not make use of the decoder component of the AE architecture, which is required for training.

It is unclear why this VAE scheme is beneficial. When the mean and standard deviation are added to the representative values, the volatility that the standard AE lacks is added. In the UAE, for example, if the core representation of the "5" digit is (0.1234, 0.5678), the representation was generated from a large number of values rather than a few. So, if you send a value close to (0.1234, 0.5678) (0.1200, 0.5500), the result will most likely be the number "5".

Another reason why the mean and standard deviation of the core representations are useful for data generation is that the variability component of the VAE architecture works to cover the entire range of possible decoder inputs. In other words, a normal AE has a large gap in the decoder's possible input values, whereas a VAE has a smaller gap.

Note: reference for this whole section ***[2] [3] [4]***

****

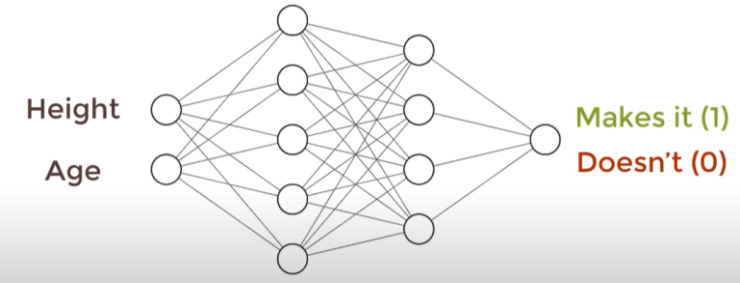
Question 2

Batch Normalisation and Skip connectors

1) Batch Normalisation:

Let me consider the example of NBA selection data and explain the Batch normalisation in simple words before explaining the mathematics behind it.

Consider there are 2000 people and once the data is passed into neural network it will predict whether that personal is selected or not selected into NBA



Person 1 → height 1.5 m, age 19 years makes it into NBA

Person 2 → height 2.0 m, age 18 years makes it into NBA

Person 3 → height 1.8 m, age 20 years makes it into NBA

Person 4 → height 1.9 m, age 45 years Does not make it into NBA

So now here neural network is trained and it will predict whether person is selected or not selected into NBA based on his height and age, the problem over here is time taken by the network to process data is more so what the simple solution is to normalise the data, so below is the normalised data of all the 4 people above

.

Person 1 → height 0.6 m, age 0.38 years makes it into NBA

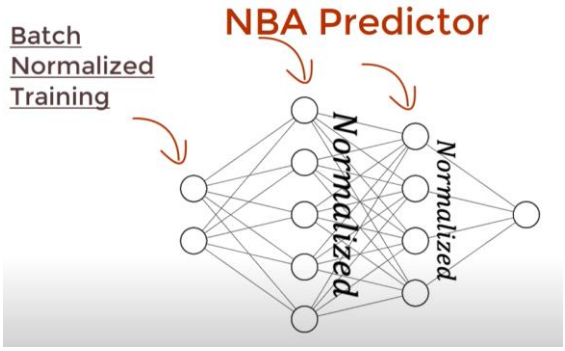
Person 2 → height 0.8 m, age 0.36 years makes it into NBA

Person 3 → height 0.72 m, age 0.4 years makes it into NBA

Person 4 → height 0.76 m, age 0.9 years Does not make it into NBA

After dividing the height by 18 and age by 5 the data looks very small then the network takes less time to train the given data.

So here as we have normalised the data to train the network faster in the same we, the data for **activation** functions can also be normalised using batch normalisation.



Here we call this as batch normalisation because we normalise the values with respect to the batch inputs.

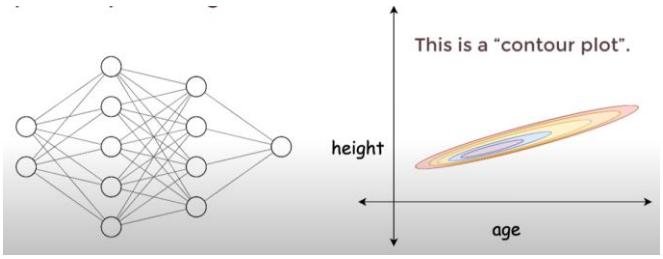
Why we have to do batch noramalisation?

1) Speeds up the training
2) Decreases importance of initial weights
3) Also regularises the model (a little bit)

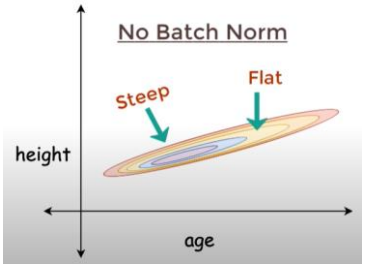
Let's looks at each of the above in detail

1) Speeds up training

Graph before normalisation



We had the two features height and age, feeding these values will make the loss look as shown in the above graph and it is elongated. This is because the age is have small range from 0 ,.. 2.0 and the age is had larger range like 0 to 100 years, from the above graph it can said the very small variations in the heights can greatly change the loss in order to effectively learn with the gradient descent we need a smaller learning rate to ensure we don't overshoot the minimum.

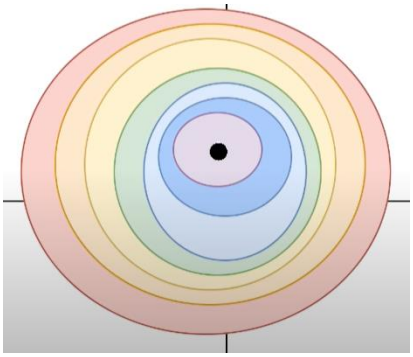


We can see the from the centre point on the left side edges of the circle are closer when compared on the right side, in other words one side it is steep and other side it is flat, which make it difficult to learn for the model

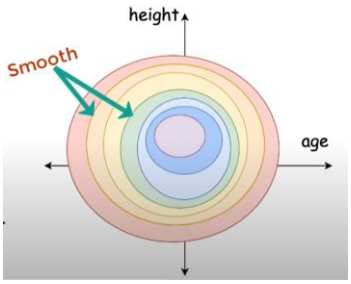
Normalisation will also speed up training How?

we can normalise the data by subtracting the mean value and divide it by standard deviation.

Normalised value = (data – mean) / standard deviation

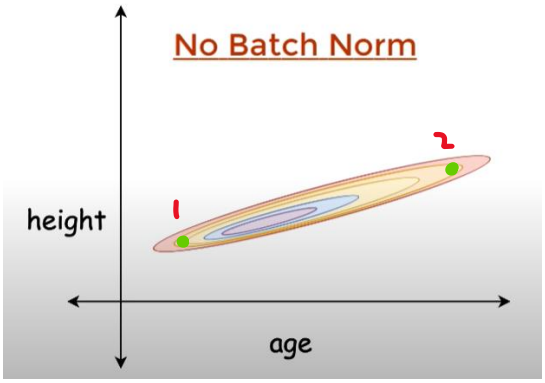


After normalising the data, the graph looks more symmetric where in now we can use much larger learning rate getting to the minimum faster, hence it can be said the normalisation speeds up the training.

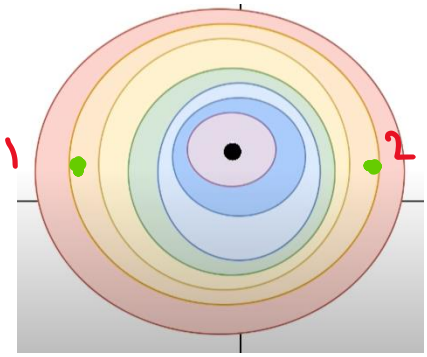


Batch normalisation has the effect of smoothing out the terrain making the concentric rings more evenly spread which makes it must easier to traverse.

2) Decreases importance of initial weights



In the above graph there are two points 1 and 2, if we start from left point it might takes 100 iterations to get to the minimum where as if we start from the 2nd point it might take 1000 iterations to get to the minimum. We cannot define its initial value since its range is very large.



Let us look at the above graph, here we can randomly sample the number between negative 1 and positive 1 for an initial start to the feature, and no matter where we start we will end up at the minimum in a similar number of iterations, **so it can said that batch normalisation help initial weights less importance.**

3) Acts as regulariser (a little bit)

When the topic of regularisation comes in the neural network we think of the drop out, that is we drop some of the neurons while training the data by making it to 0 or 1, where 0 means the neuron is de-active and no calculation will happen and 1 means this neuron is active.

With batch normalisation to there is element of randomness the mean invariants for every neuron activation. These values are highly dependent of batch, the batch which is composed of random samples and in that regard batch normalisation does induce some regularisation. Despite this we use the batch normalisation with drop out for better results

So above were the 3 main reasons why batch normalisation helps in performance

Working of batch normalisation

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.

From above picked up algorithm we can see that the model is trained with the network with mini batch gradient descent this means the parameters are updated only after we see a batch of some **'M'** samples.

Let us consider the neuron what is coloured yellow, assuming that batch size is 3 it means that we will update the parameters after seeing 3 samples passing the first sample the neuron has activation of 4, passing the second sample the neuron will have activation of 7 and for third the activation is 5.

From above activation

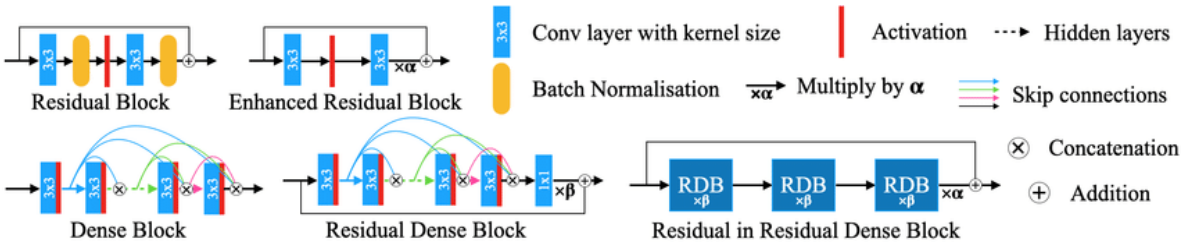
Mini-batch mean = (1/3) * (4+7+5) = 5.33

Variance = (1/3) * $(4 - 5.33)^2$ + $(7 - 5.33)^2$ + $(5 - 5.33)^2$ = 1.555

Now we can normalise the activation x1, x2, and x3 by subtracting the mean and dividing by the standard deviation. So now we after we normalise the data we will get the mean of 0 and unit standard deviation of 1 for the activation, so this speeds up training. So this cannot be left like this we can see from above algorithm that the mean and variance are heavily dependent on the samples of the batch and so we can calibrate the mean and variance by introducing two learnable parameters for each neuron **gamma, beta**. Training over multiple batches gamma should approximate to the true mean of the neuron activation and beta must approximate to the true variance of the neuron activation. So by doing this we get activations not only speeds up the execution but also gives the good results.

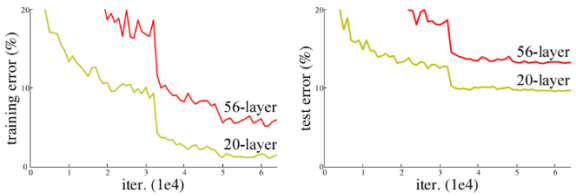
(References for this section are [6],[7], [8], [9], [10]

2) Skip Normalisation:



https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.researchgate.net%2Ffigure%2FResidual-and-dense-blocks-These-blocks-consist-of-convolution-layers-and-activation_fig2_351841317&psig=AOvVaw0WTt814uIb6trr_dCDxzNx&ust=1649106283555000&source=images&cd=vfe&ved=0CAsQjRxqFwoTCNjwu-jI-PYCFQAAAAAdAAAAABAY

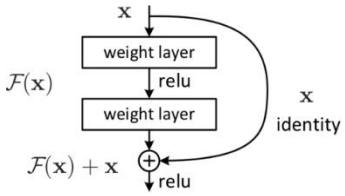
The deep neural networks can learn complex function more efficiently than their shallow counterparts, sometimes while training the network the performance of the model will drops down as the depth of the architecture increases, this is know as **Degradation problem**. The possible reason for this could be overfitting of the training data, the model tends to overfit with as it goes deeper and deeper into the network, but the problem is now just of deep of the network, from below graph we can clearly see that 56 layers has more training erro than the shallow layer of 20. The deeper model doesn't perform as well as the shallow one. So the conclusion is overfitting is not the problem here.



https://www.analyticsvidhya.com/blog/2021/08/all-you-need-to-know-about-skip-connections/

skip connection was introduced to solve different problem in variety of different neural network architecture. ResNet solved the degradation problem.

ResNets was proposed by He et al. in 2015 to solve the image classification problems. The information from the initial layers is passed to deeper layers in ResNets via matrix addition. Because the output from the previous layer is added to the layer ahead, this operation requires no additional parameters. A single residual block with a skip connection appears as follows:



https://www.analyticsvidhya.com/blog/2021/08/all-you-need-to-know-about-skip-connections/

As pre-trained weights from this network, the deeper layer representation of ResNets can be used to solve multiple tasks. It is not only limited to image classification, but it can also solve a wide range of image segmentation, keypoint detection, and object detection problems. As a result, ResNet is regarded as one of the most influential architectures in the deep learning community.

Basically the working of skip net is lets say we have 10 layers architecture, and we want to skip from the 4^{th} layer to 7^{th} layer that is the model is trained in the 4^{th} layer and skips 5^{th}, 6^{th} layers and directly jumps to the 7^{th} layer for the next calculation, this concept is called skip connections, and the block from 4^{th} to 7^{th} layers are called residual blocks.

Reference for this section is [11]

Reference

1. https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/
2. https://www.quora.com/What-are-the-disadvantages-or-drawbacks-of-using-autoencoders-for-feature-extraction
3. https://towardsdatascience.com/generating-images-with-autoencoders-77fd3a8dd368#:~:text=Autoencoders-,Traditional%20Autoencoders,principal%20component%20analysis%20(PCA).
4. https://jaan.io/what-is-variational-autoencoder-vae-tutorial/
5. https://jamesmccaffrey.wordpress.com/2020/05/07/the-difference-between-an-autoencoder-and-a-variational-autoencoder-2/
6. https://arxiv.org/abs/1502.03167
7. https://arxiv.org/abs/1805.11604
8. https://arxiv.org/abs/1905.05928
9. https://www.quora.com/Is-there-a-theory-for-why-batch-normalization-has-a-regularizing-effect
10. https://www.youtube.com/watch?v=DtEq44FTPM4
11. https://www.analyticsvidhya.com/blog/2021/08/all-you-need-to-know-about-skip-connections/