Name: Melwyn D Souza
Student Number: R00209495
Date: 3/April/2022
Module: Deep Learning
Lecturer: Dr Ted Scully
Course: MSc in Artificial Intelligence

Dataset

Fashion-MNIST is a dataset which consists images of different fashion accessories 28x28 pixels, the following are the labels of the dataset

- T-shirt/top (0)
- Trouser (1)
- Pullover (2)
- Dress (3)
- Coat (4)
- Sandal (5)
- Shirt (6)
- Sneaker (7)
- Bag (8)
- Ankle boot (9)

It consists of 60,000 training instances and 10,000 test instances; the goal of this assignment is to classify the right class for the given instance and to apply the gradients to the test data for measuring accuracies and losses.

Q 1_1_1 Network A:

The network is made up of two layers

- Layer 1: 200 Neurons (ReLu activation)
- Layer 2: 10 Neurons (SoftMax activation)

I have followed instructions from Coursera Andrew Ng [1] instructions on initializing the weights and bias

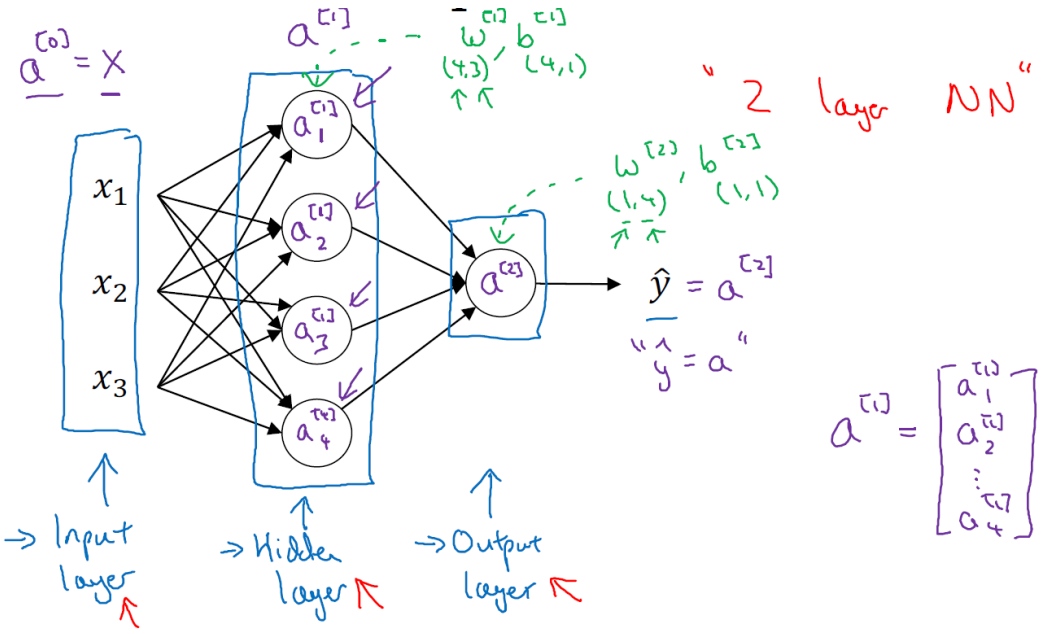


Figure 1: Snippet from Andrew Ng's Deep Learning course as an example [1]

Referring figure 1, the weights for Network A of assignment 1 are initialised as follows.

W -> weights, W1 -> weights for layer 1, B-> bias, B1 -> Bias for layer 1

- W1: (200,784)
- B1: (200,1)
- W2: (10,200)
- B2: (10,1)

```
#when using RELU, multiply the weights with sqrt(2/n) to mitigate vanishing/exploding weights for large 'n' - Andrew Ng
W1 = tf.Variable(tf.random.normal([200,784]) * tf.sqrt(2/784))
W2 = tf.Variable(tf.random.normal([10,200]) * tf.sqrt(2/200))
B1 = tf.Variable(tf.zeros([1]))
B2 = tf.Variable(tf.zeros([1]))
```

Figure 2: Weight and Bias initialisation in main() function of the code

There are three main functions used in question 111

Forward_pass, cross entropy loss and calculate accuracy

```python
def forward_pass(X, W1, B1, W2, B2):
  """
  Performs forward pass for the neural network

  Parameters
  ----------
  X : 60000,784 TensorFlow Variables of training data
  W1 : 200,784 TensorFlow Variables of layer 1 weights
  B1 : 200,1 TensorFlow Variables of layer 1 bias
  W2 : 10,200 TensorFlow Variables of layer 2 weights
  B2 : 10,1 TensorFlow Variables of layer 2 bias

  Returns
  -------
  A2 : 60000,10 tensorflow variables of softmax layer output, probabilities of 10 classes of 60000 images
  """

  Z1 = tf.matmul(X,tf.transpose(W1)) + B1
  A1 = Activation('relu')(Z1) #shape 60,000 X 200

  Z2 = tf.matmul(A1, tf.transpose(W2)) + B2
  A2 = tf.exp(Z2)/tf.reduce_sum(tf.exp(Z2), axis=-1).reshape(Z2.shape[0],1)
  # print(tf.reduce_sum(A2, axis=-1)) # checking if the probabilities add to 1 on all rows
  return A2
```

Figure 3: Forward Pass function for question 1_1_1

From figure 3, the function takes 5 arguments, X is the input image with 60000X28X28 (flattened to 60000X784) features, so ‘n’ or number of features are 784. These 784 features are passed to layer 1 which is a ReLu layer with 200 neurons, each neuron will get 784 connections hence the total number of weights will be 200X784, the output from this layer is passed to layer 2 which is a SoftMax layer with 10 neurons, each neuron receives 200 connections hence the total weights will be 10X200

For a single neuron, there are two operations to be performed as shown in figure 4, for the fashion MNIST I have used vectorization which reduces the usage of for loops hence largely reducing the total runtime

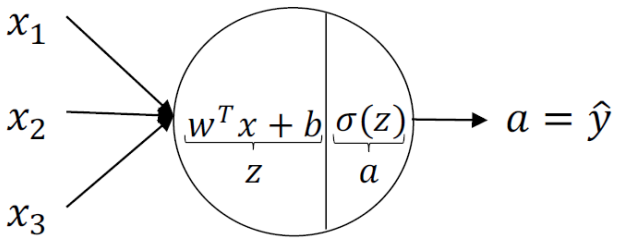


Figure 4: Operations inside a single neuron

In the forward_pass function, Z1 performs matrix multiplication to calculate the state of 200 neurons, these states are then activated using ReLu function (A1), this completes L1 operations. The A1 are the inputs for L2, Z2 performs matrix operations to calculate state of 10 neurons, these states are passed through A2 which is a SoftMax activation function. Since L2 is the last layer (Output layer), A2 is also our Y’ or the predicted outputs

Cross entropy

```python
def cross_entropy(pred_y, true_y):
  """
  Performs Categorical Cross Entropy Loss calculation

  Parameters
  ----------
  pred_y : 60000,10 tensorflow variables from softmax layer - forward_pass()
  true_y : 60000,10 tensorflow variables of input image categorical lables

  Returns
  -------
  loss : float32 value of Categorical Cross Entropy loss calculated as shown in the figure above
  """
  # cce = tf.keras.losses.CategoricalCrossentropy()
  # print("This is the loss from built in function", cce(true_y, pred_y).numpy())
  loss  = -tf.reduce_mean(tf.reduce_sum(true_y * (tf.math.log(pred_y)), axis=-1))
  return loss
```

Figure 5: Cross Entropy Loss function

The predicted output will consist of 10 classes for Fashion MNIST data instances, the loss is calculated using the cross-entropy formula below, the implementation of the formula in my code is shown in figure 5, log value of the predicated output is found using, multiplied with the true instance and finally mean of all ‘m’ instance losses is calculated. The loss is expected to be decreasing on every epoch.

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

Figure 6: Cross Entropy Loss

Accuracy:

```python
def calculate_accuracy(pred_y, true_y):
  """
  Calculate accuracy

  Parameters
  ----------
  pred_y : 60000,10 tensorflow variables from softmax layer - forward_pass()
  true_y : 60000,10 tensorflow variables of input image categorical lables

  Returns
  -------
  acr : float32 value of Accuracy between predicted class and True class

    """
  pred_y = tf.round(pred_y)
  pred_correct = tf.cast(tf.equal(pred_y, true_y), tf.float32)
  acr = tf.reduce_mean(pred_correct)
  return acr
```

Figure 7: Accuracy function

The accuracy function finds if the predicted class equals to the true class, the output from the forward pass (A2 – SoftMax 10 neurons) will be giving 10 probabilities for each example, the highest probability will be the predicted class. The pred_correct variable will have 1's and 0's depending if the true labels match with predicted labels, the mean of these values is the accuracy that we want.

In the code, the 'main' function is used to run epochs, Gradients are taped for backpropagation in order to update the weights and bias on every epoch, tensorflow GradientTape() is used in the code, the input or training examples are pushed to forward_pass and the predicted outputs are then used in calculating loss and accuracy, the losses, weights and bias is then zipped together and given to Adam optimizer for backpropagation. The neural network is trained for num_Iters number of epochs

```
def main():
  tf.random.set_seed(50)
  num_Iters = 500
  adam_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
  train_loss, train_acc, test_loss, test_acc, epoc = [], [], [], [], []

  #get processed data from get_data()
  train_x, train_y, test_x, test_y = get_data()

  # Create tensorflow variables and change datatype to float64
  dt = tf.float32
  train_x = tf.cast(train_x, dt)
  train_y = tf.cast(train_y, dt)
  test_x = tf.cast(test_x, dt)
  test_y = tf.cast(test_y, dt)

  #when using RELU, multiply the weights with sqrt(2/n) to mitigate vanishing/exploding weights for large 'n' - Andrew Ng
  W1 = tf.Variable(tf.random.normal([200,784]) * tf.sqrt(2/784))
  W2 = tf.Variable(tf.random.normal([10,200]) * tf.sqrt(2/200))
  B1 = tf.Variable(tf.zeros([1]))
  B2 = tf.Variable(tf.zeros([1]))

  for i in range(num_Iters):
    epoc.append(i)
    with tf.GradientTape() as tape:
      predicted_y = forward_pass(train_x, W1, B1, W2, B2)
      crossEntLoss = cross_entropy(predicted_y, train_y) # check this out
      train_loss.append(crossEntLoss)
    gradients = tape.gradient(crossEntLoss,[W1, W2, B1, B2])
    accuracy = calculate_accuracy(predicted_y, train_y)
    train_acc.append(accuracy)
    print("Iteration: {}, Loss: {}, Accuracy: {}".format(i, crossEntLoss, accuracy))

    adam_optimizer.apply_gradients(zip(gradients,[W1, W2, B1, B2]))

    #test data - validate
    pred_y = forward_pass(test_x, W1, B1, W2, B2)
    loss = cross_entropy(pred_y, test_y)
    test_loss.append(loss)
    accuracy = calculate_accuracy(pred_y, test_y)
    test_acc.append(accuracy)
```

Figure 8: main() function to run the specified epochs, forward and backpropagation on the networks

Q 1_1_2 Network B:

The only difference between n/w A and B is an extra layer, the architecture of this network is as follows.

- Layer 1: 300 neurons (ReLu)
- Layer 2: 100 neurons (ReLu)
- Layer 3: 10 neurons (SoftMax)

The forward pass, loss and accuracy functions are the same as network A, except for the parameters are more because of the extra layers

The train accuracy and test accuracies of both networks are shown in figures below

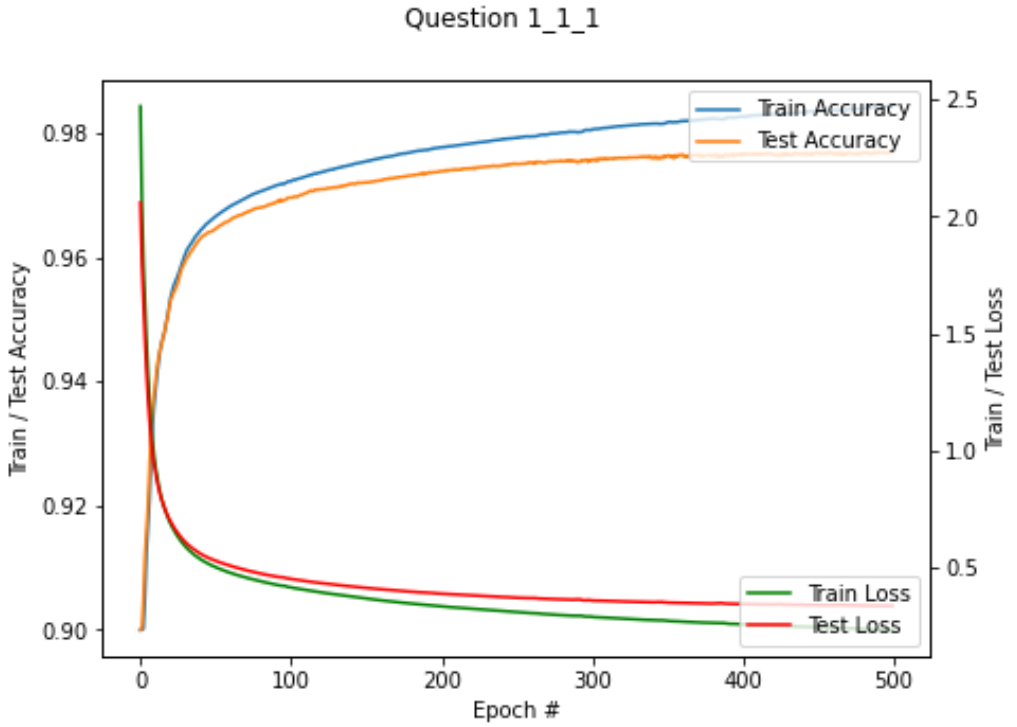


Figure 9: learning curve for q111 – network A

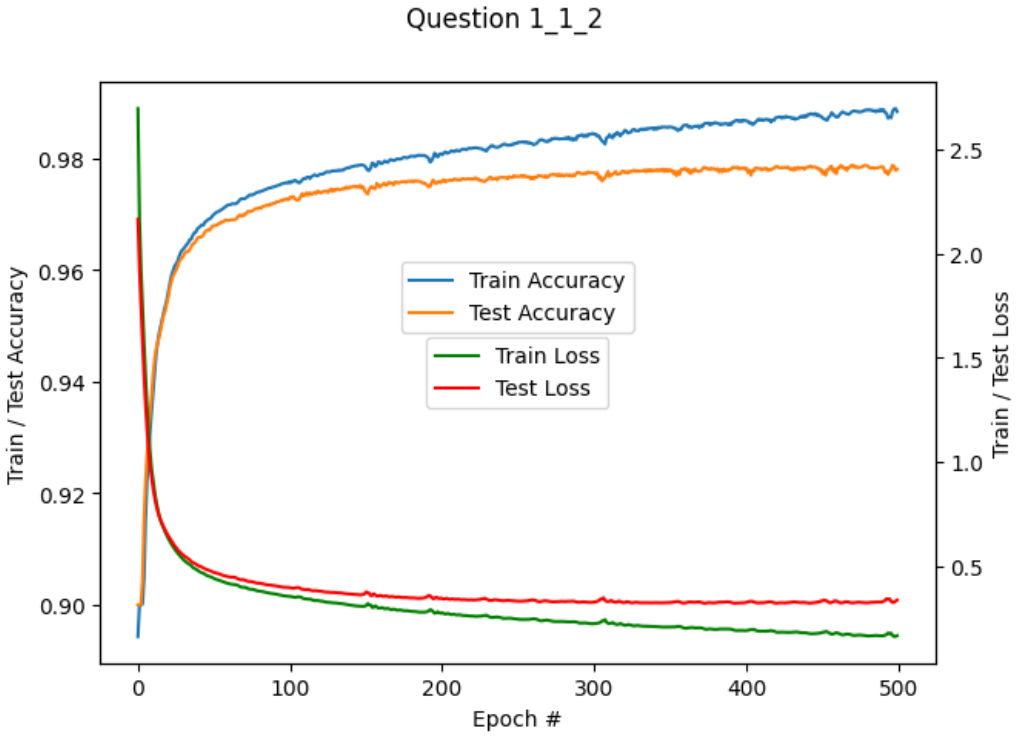


Figure 10: learning curve for q112 – network B

From the graphs above, the training loss and test loss are diverging with the number of epochs, this is a good sign of overfitting of a model, network B with an extra layer is attaining higher accuracies on train set (~99%) whereas network A is giving 97% accuracy, this is because of the extra layer in the network, the computational expense is higher for network B with extra weights and bias gradient, small ripples can also be noticed on the curves for network B.

The network is trained for 500 epochs which is a large number, so the overfitting of the model is expected when the network is over trained, but the divergence can be noticed at ~30 epoch for both networks, the losses start diverging slowly at this point as shown in the figures below

Another good comparison is the accuracy curves between two networks, for network A, the accuracy at epoch 20 is around 0.945 whereas network B attains 0.955

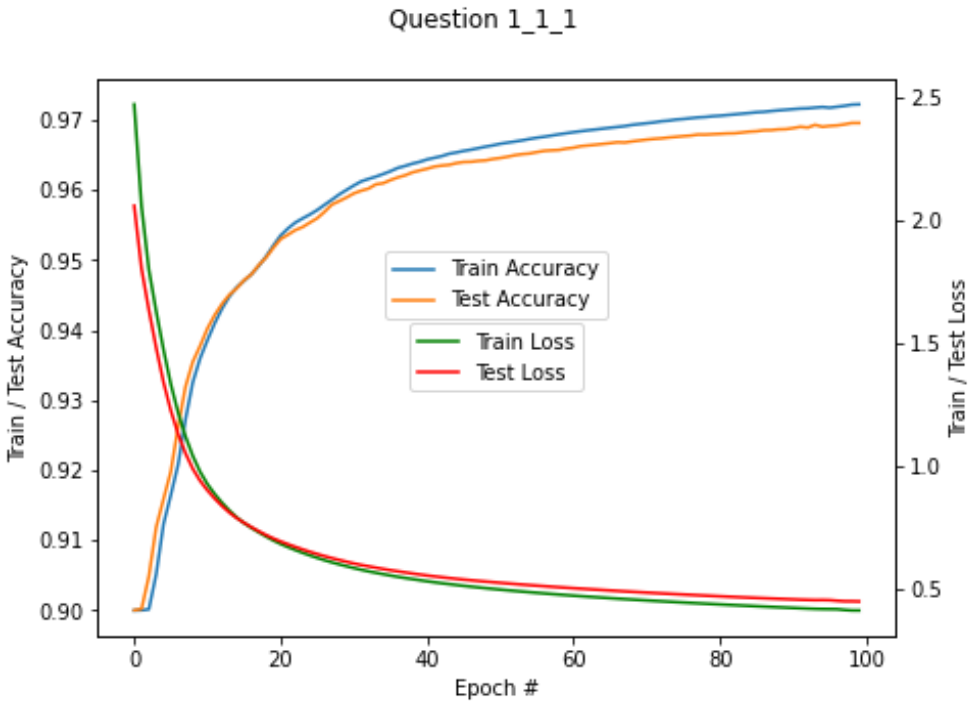


Figure 11: Learning curves of network A with 100 epochs

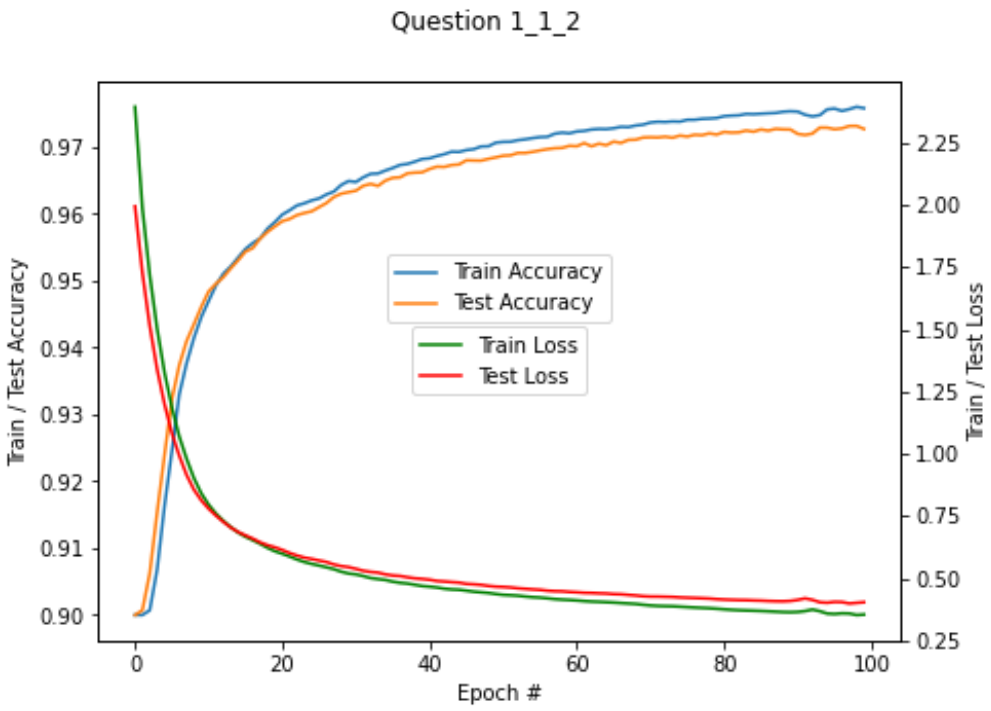


Figure 12: Learning curves of network B with 100 epochs

During the execution, vanishing gradients for network B was noticed, in conclusion, adding layers to the networks can cause overfitting, longer execution times and additional computational resource.

Q 1_2_1

Usually, the neural networks are prone to overfitting, there are different ways to reduce overfitting of a deep neural network as listed below

- Remove features
- Early stopping of the network
- Regularization
- Ensembling

In the assignment, Q 1_2_1 we will focus on Dropout Regularization, dropout is a technique where in every layer we drop a neuron or we make it zero as if it never existed, this is randomly done on each layer depending on the dropout probability, so if dropout probability is 0.2, 20% of the neurons in the layer will be turned off for current epoch

For Q 1_2_1, the dropout layer is applied on layer 1, the network architecture is lister below

- Layer 1: 300 neurons (ReLu)
- Layer 2: Dropout Layer
- Layer 2: 100 neurons (ReLu)
- Layer 3: 10 neurons (SoftMax)

Forward Pass:

```
def forward_pass(X, W1, B1, W2, B2, W3, B3, dropoutProb):
  """
  Performs forward pass for the neural network

  Parameters
  ----------
  X : 60000,784 TensorFlow Variables of training data
  W1 : 300,784 TensorFlow Variables of layer 1 weights
  B1 : 300,1 TensorFlow Variables of layer 1 bias
  W2 : 100,300 TensorFlow Variables of layer 2 weights
  B2 : 100,1 TensorFlow Variables of layer 2 bias
  W3 : 10,100 TensorFlow Variables of layer 3 weights
  B3 : 10,1 TensorFlow Variables of layer 3 bias
  dropoutProb : int64, droupout probability in order to drop % of neurons from the layer

  Returns
  -------
  A3 : 60000,10 tensorflow variables of softmax layer output, probabilities of 10 classes of 60000 images
  """

  keepProb = 1 - dropoutProb  #1 - 0.2 = 0.8 (80% neurons ae kept)

  #Layer1, ReLu activation with 300 neurons
  Z1 = tf.matmul(X,tf.transpose(W1)) + B1
  A1 = Activation('relu')(Z1) #shape 60,000 X 200

  #Dropping 20% of the neurons
  dropMatrix = tf.random.uniform([A1.shape[0], A1.shape[1]], minval=0, maxval=1) < keepProb
  A1 = A1 * dropMatrix #20% neurons are set to 0, or dropped for the next layer
  A1 = A1 / keepProb #to scale the 80% neuron to 100%

  #Layer2, ReLu activation with 100 neurons
  Z2 = tf.matmul(A1, tf.transpose(W2)) + B2
  A2 = Activation('relu')(Z2)

  #Layer3, SoftMax layer with 10 neurons (10 classes)
  Z3 = tf.matmul(A2, tf.transpose(W3)) + B3
  A3 = tf.exp(Z3)/tf.reduce_sum(tf.exp(Z3), axis=-1).reshape(Z3.shape[0],1)
  # print(tf.reduce_sum(A2, axis=-1)) # checking if the probabilities add to 1 on all rows

  return A3
```

Figure 13: Forward Pass function for Q121

The function takes droupoutProab as an argument with other arguments, this is the probability of dropping neurons randomly at the specified layer, which is layer 1 neurons in the assignment. The keepProb is used to keep rest of the neurons in their working state. keepProb is used to create a mask with shape same as layer 1, the dropMatrix then consists of (keepProb)% of Boolean True and (dropoutProb)% of Boolean False, the layer 1 matrix is masked with dropMatrix. Now since the next layer state (Z2) will also be reduced by 20% since we are using only 80% of the neurons from layer 1, we will bump the A1 outputs back up to 100%. All other operations of the function are same as discussed earlier in this report.

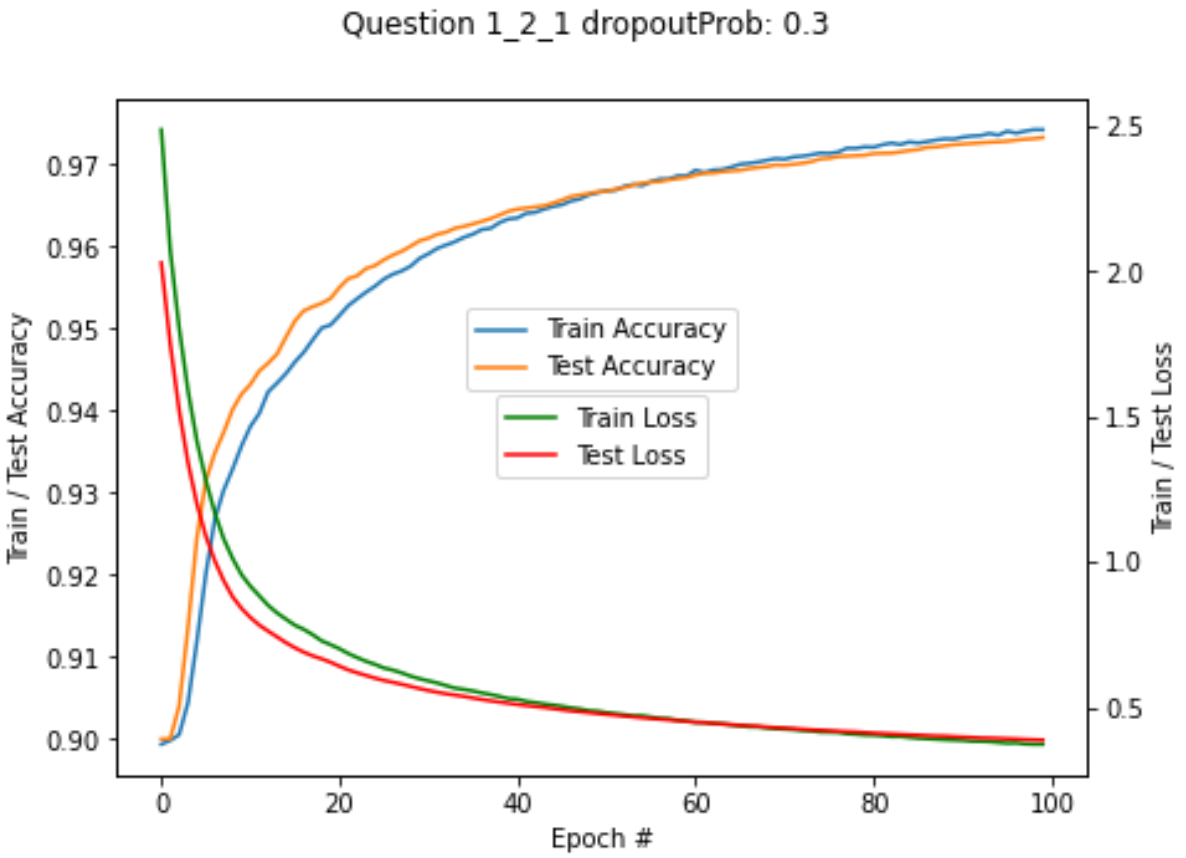


Figure 14: Learning curves with dropout regularization

The figure 14 shows the mitigation of overfitting when compared to network B learning curves in figure 12, network B is the exact same architecture but without dropout regularization, in Figure 12 around 30 epochs the losses start diverging as an indication of overfitting, whereas in figure 14, the network seems to be working very well even at 100th epoch, said that, large number of epochs will introduce overfitting of the network

I have evaluated the network with 5 dropout probabilities 0.1,0.2,0.3,0.4,0.5 and plotted the results as shown below

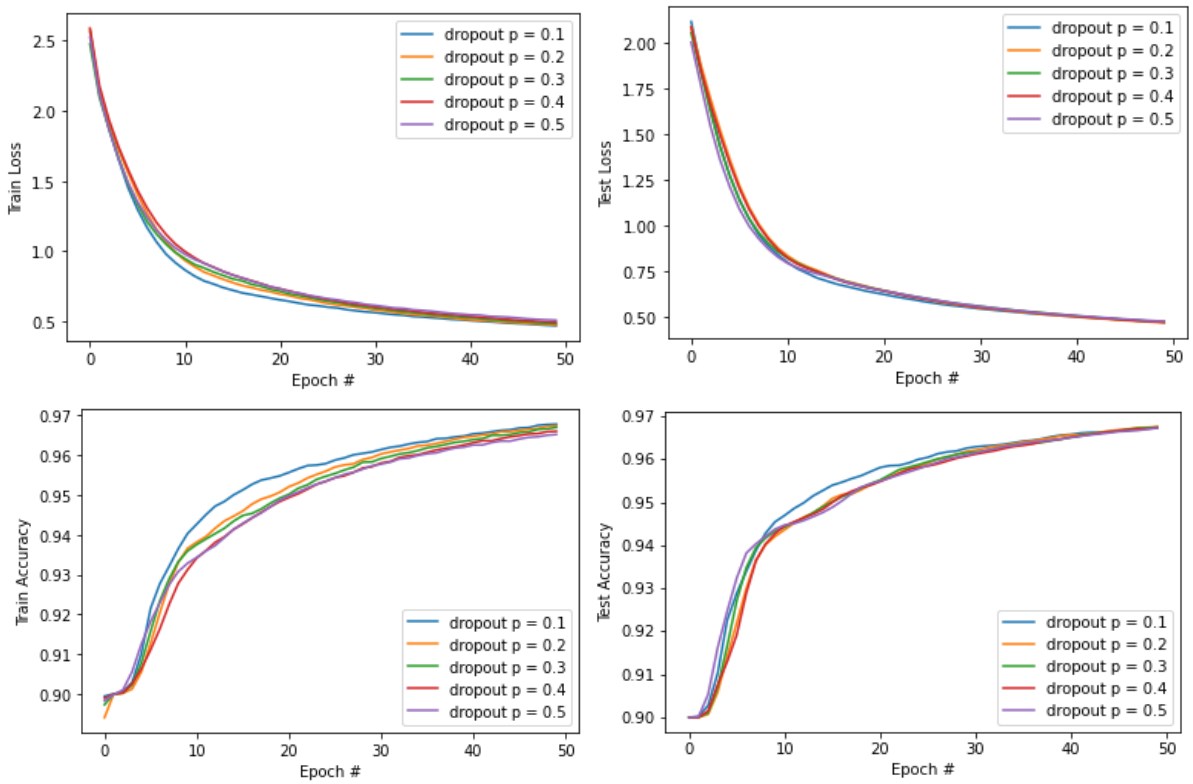
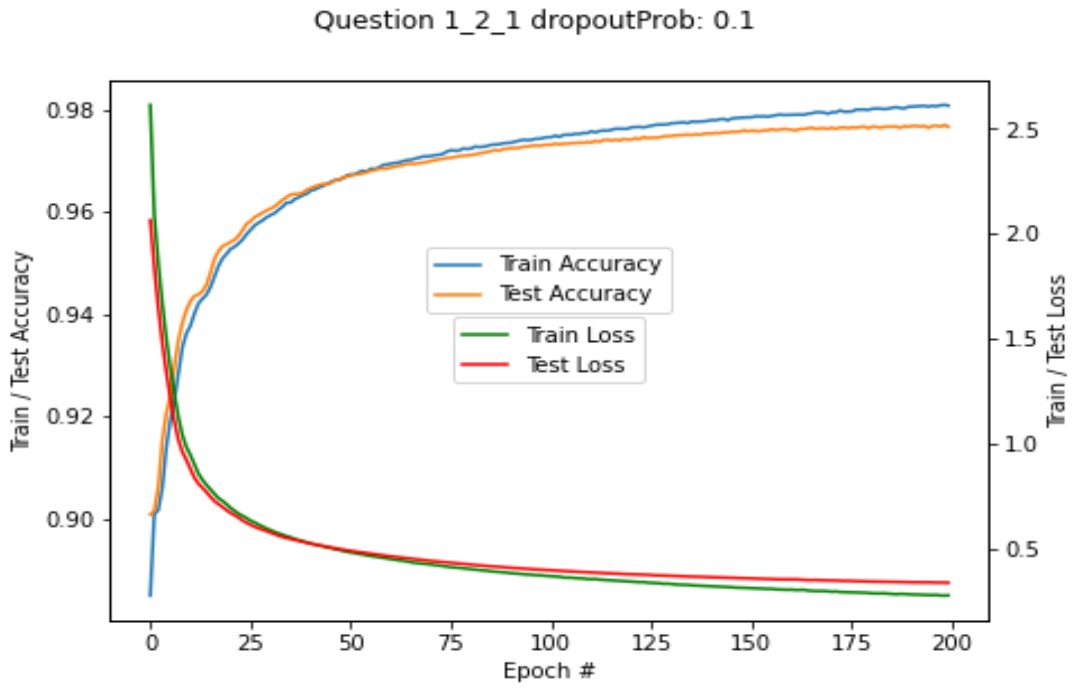
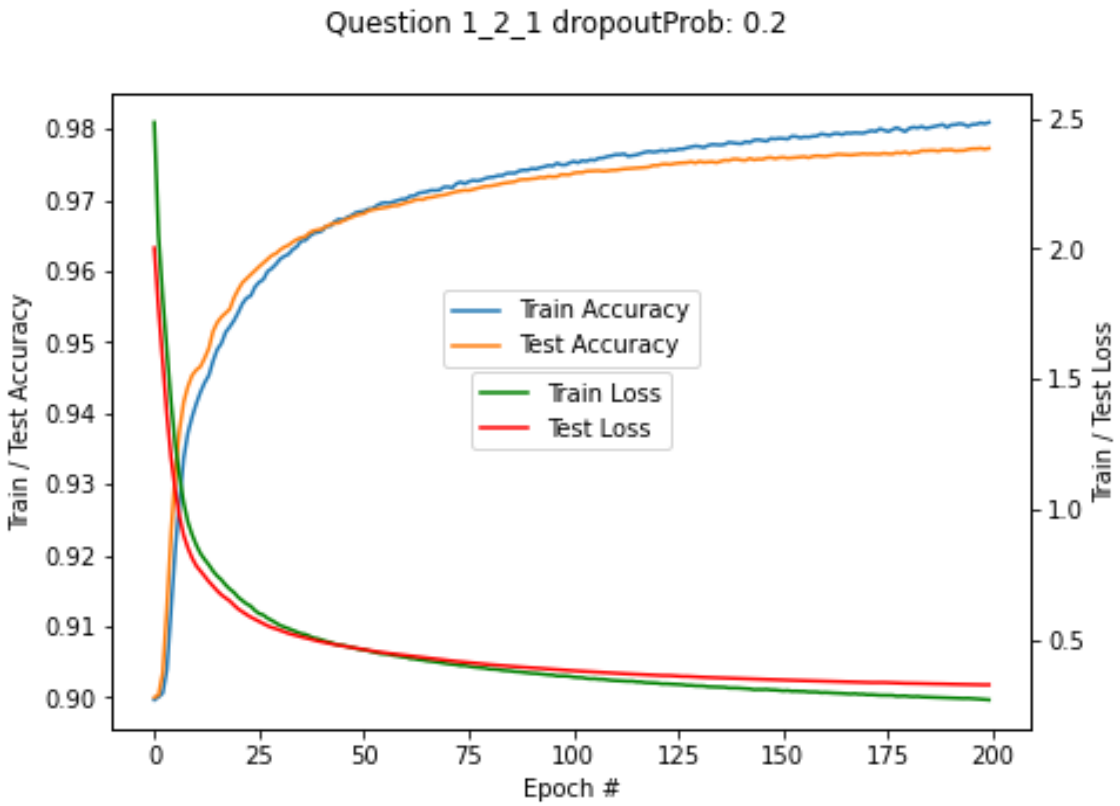


Figure 15: Accuracies and Losses for different dropout probabilities

From the graphs above for different dropout probabilities show the differences in accuracies and losses for training and test data, although they seem very different for initial epochs, they all converge after 30 epochs and give almost same accuracies and losses

To select the best dropout probability, I referred the figures below after training and testing the data for 200 epochs on 5 different probabilities, concluding that the probabilities in the range of 0.1 and 0.2 are better from the graphs for epochs ranging from 1-75





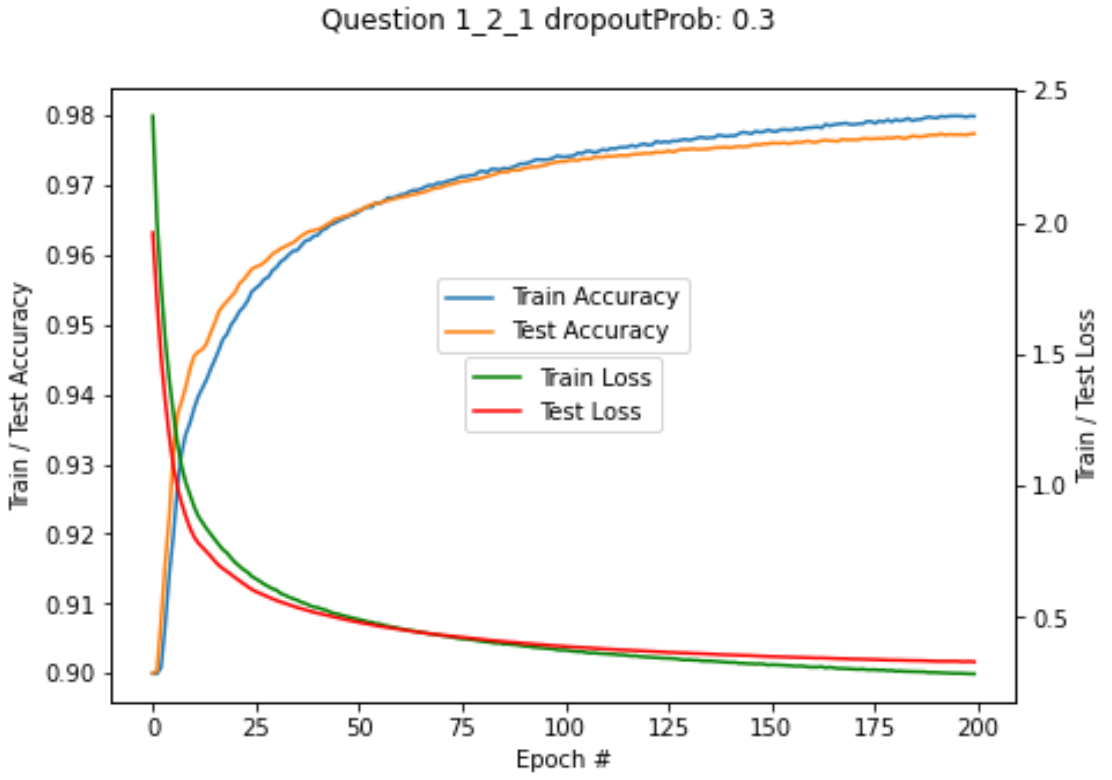


Figure 16: Learning curves for dropout probabilities 0.1, 0.2, 0.3

L2 regularization:

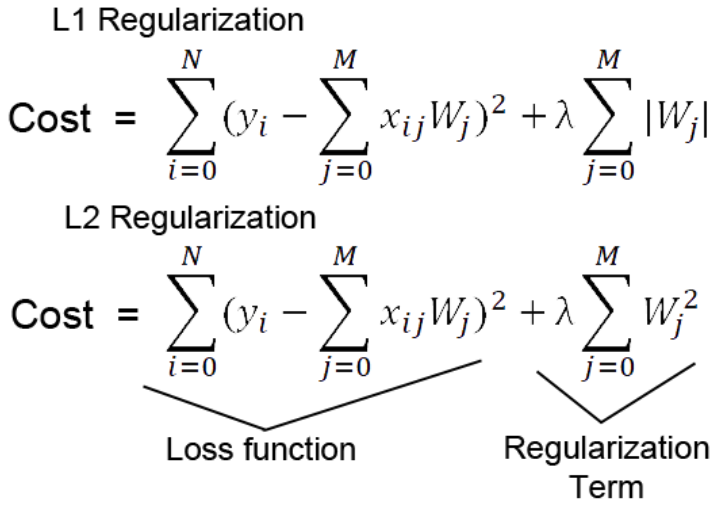


Figure 16: L1 and L2 regularisation formula

Usually Weight 'W' is a very high dimensional parameter and Bias is not, hence focus is more on adding regularisation techniques on the weights

I can add L2 regularization term to the cross_entropy loss function in my code which will then be used by ADAMs optimizer when updating the weights for the next epoch

Q 1_3_1

Mini-Batch gradient descent is a technique which divides the whole dataset into small chunks of data, so the froward pass is carried on these small batches and backpropagation follows. This makes the gradients to be upgraded many times prior to completion of one epoch, we can have very good weights and bias value because the gradients are upgraded after every mini-batch forward pass

Stochastic gradient descent (SGD) on the other hand follows the same principle as the mini-batch but the data which forms these mini batches is selected randomly from the larger dataset, these techniques are used for the network and neurons to stop synchronizing and attain high accuracies and lower losses in very less epochs

The disadvantages of mini-batch is the execution time, mini bath size and execution time are inversely proportional, my implementation of SGD is shown in the figure below.

```
m = train_x.shape[0]
mini = m/minibatch

for i in range(num_Iters):
  epoc.append(i)
  batch_X_axis, batchLoss, batchAcc = [], [], []

  #random shuffle of the dataset before slicing them into batches
  data = list(zip(train_x,train_y))
  random.shuffle(data)
  train_x, train_y = zip(*data)

  for j in range(1,round(mini)):
    batch_X_axis.append(j)
    tr_x = train_x[(j-1)*minibatch : (j)*minibatch] #mini batch of train_x from shuffled data
    tr_y = train_y[(j-1)*minibatch : (j)*minibatch] #mini batch of train_y from shuffled data

    with tf.GradientTape() as tape:
      predicted_y = forward_pass(tr_x, W1, B1, W2, B2, W3, B3)
      crossEntLoss = cross_entropy(predicted_y, tr_y)
      batchLoss.append(crossEntLoss)
    gradients = tape.gradient(crossEntLoss,[W1, W2, W3, B1, B2, B3])
    accuracy = calculate_accuracy(predicted_y, tr_y)
    batchAcc.append(accuracy)
    print( "Mini batch: {}, Loss: {}, Accuracy: {}".format(j, crossEntLoss, accuracy))
    adam_optimizer.apply_gradients(zip(gradients,[W1, W2, W3, B1, B2, B3]))
```

Figure 17: SGD implementation in the main function of Q 1_3_1

Form the figure above, m is the total number of training instances, minibatch is the size of minibatch (16,32,64 etc), mini is the total number of minibatches. The whole dataset i.e., train_x and train_y is zipped together and shuffled on every epoch, the shuffled data is then sliced into mini batches depending on the size specified. The gradients are upgraded on every mini batch using ADAMs optimizer. The output for minibatch size 64 and 1 epoch is shown below

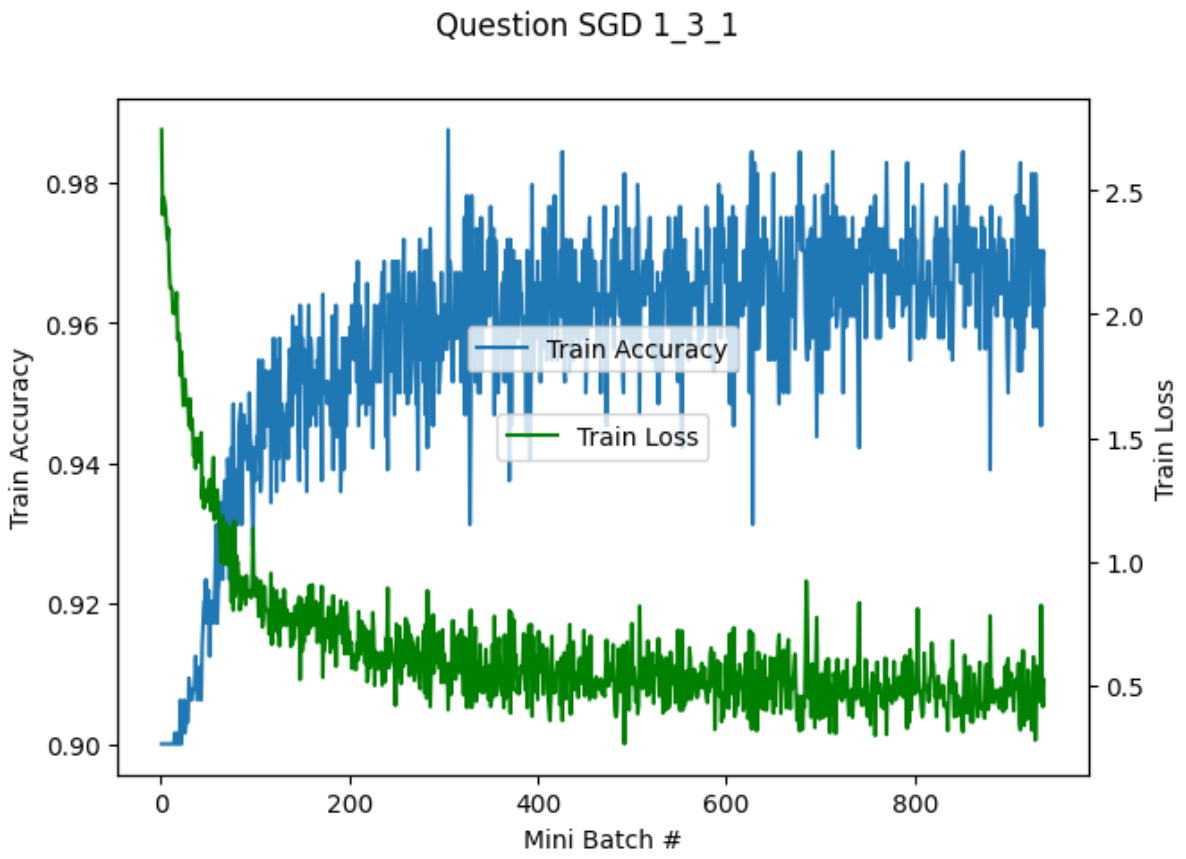


Figure 18 : Learning curve of SGD with mini batch size 64

The total number of examples are 60,000 and mini batch size is 64, therefore the total mini batches available will be ~900 which is the x-axis in the plot, we can see very great reduction in the loss and higher accuracy gain after completion of only one epoch, this is because the gradients were upgraded ~900 times for one epoch, the ripples in the loss and accuracies is because the batch size is very small for the network to be tuned perfectly, hence when it gets new mini batch data on every loop, it might perform better or worse than the previous loop

Q 1_4_1

Autoencoders are used to encode, compress data, it reconstructs the data back from the encoded compressed data close to the original input. They are used in denoising the input data. There are four main steps in autoencoder as listed below

- Encoder – Network reduces the dimensions of the input data
- Bottleneck – Compressed data, this is the lowest dimension of the input data
- Decoder – Reconstruction of the data close to the input data
- Reconstruction Loss – Measure loss between the reconstructed data and original data

In the assignment Q 1_4_1, fashion MNIST data is added with noise and passed through a simple autoencoder whose architecture is discussed below, the loss is calculated between the denoised output of autoencoder and the original fashion MNIST images. The architecture of the autoencoder is as follows

- Layer 1: 128 Neurons (ReLu)
- Layer 2: 64 Neurons (ReLu)
- Layer 3: 32 Neurons (ReLu)
- Layer 4: 64 Neurons (ReLu)
- Layer 5: 128 Neurons (ReLu)
- Layer 6: 784 Neurons (Sigmoid)

The figure below shows the noisy images on top axis and original data without noise on the bottom row

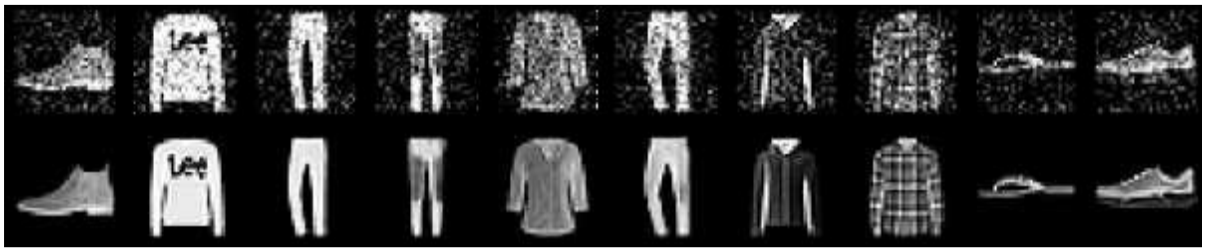


Figure 19: Fashion MNIST data with noisy images and original images

Forward Pass:

The figure below shows the forward pass with 6 layers, these layers consist of encoder, bottleneck, decoder, and reconstruction

```
def forward_pass(X, W1, B1, W2, B2, W3, B3, W4, B4, W5, B5, W6, B6):
  #Layer1, ReLu activation with 300 neurons
  Z1 = tf.matmul(X,tf.transpose(W1)) + B1
  A1 = Activation('relu')(Z1) #shape 60,000 X 200

  #Layer1, ReLu activation with 300 neurons
  Z2 = tf.matmul(A1,tf.transpose(W2)) + B2
  A2 = Activation('relu')(Z2) #shape 60,000 X 200

  #Layer1, ReLu activation with 300 neurons
  Z3 = tf.matmul(A2,tf.transpose(W3)) + B3
  A3 = Activation('relu')(Z3) #shape 60,000 X 200

  #Layer1, ReLu activation with 300 neurons
  Z4 = tf.matmul(A3,tf.transpose(W4)) + B4
  A4 = Activation('relu')(Z4) #shape 60,000 X 200

  #Layer1, ReLu activation with 300 neurons
  Z5 = tf.matmul(A4,tf.transpose(W5)) + B5
  A5 = Activation('relu')(Z5) #shape 60,000 X 200

  #Layer3, SoftMax layer with 10 neurons (10 classes)
  Z6 = tf.matmul(A5, tf.transpose(W6)) + B6
  A6 = tf.sigmoid(Z6)
  # print(A6.shape)

  # print(tf.reduce_sum(A3, axis=-1)) # checking if the probabilities add to 1 on all rows


  return A6
```

Figure 20: forward pass for autoencoder

Mean Absolute Error (MAE) function:

```
def mean_absolute_error(x_denoised, x_train):
  mae = tf.reduce_mean(tf.abs(x_denoised - x_train))
  return mae
```

Figure 21: MAE for autoencoder

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

Figure 22: MAE equation

Mean absolute error is given by the equation shown above, it is the mean of the difference between the predicted or output data and the original data. For out image data with 784 features, the mean of the differences between the 784 features from the autoencoder and the original data will give us the MAE for a single instance. In the function above, MAE for all the 60000 examples is calculated using vectorisation and then divided by 60000 to get MAE of the autoencoder

I have carried on 1000 epochs to denoise the noisy data capturing the output and displaying 10 images at every 200th epoch (0,200,400,600,800) th epochs as shown below

Figure 23: Comparing reconstructed images to original images

The odd rows in the images are displaying the denoised output at every 200th epochs/1000 epochs, likewise the even rows of the figure display the equivalent original images from the data, we can see how noisy the data is in the 0th epoch before passing through the encoder, and the noisy data is denoised and makes more and more sense to viewers eyes after completing number of loops through the encoder as shown above

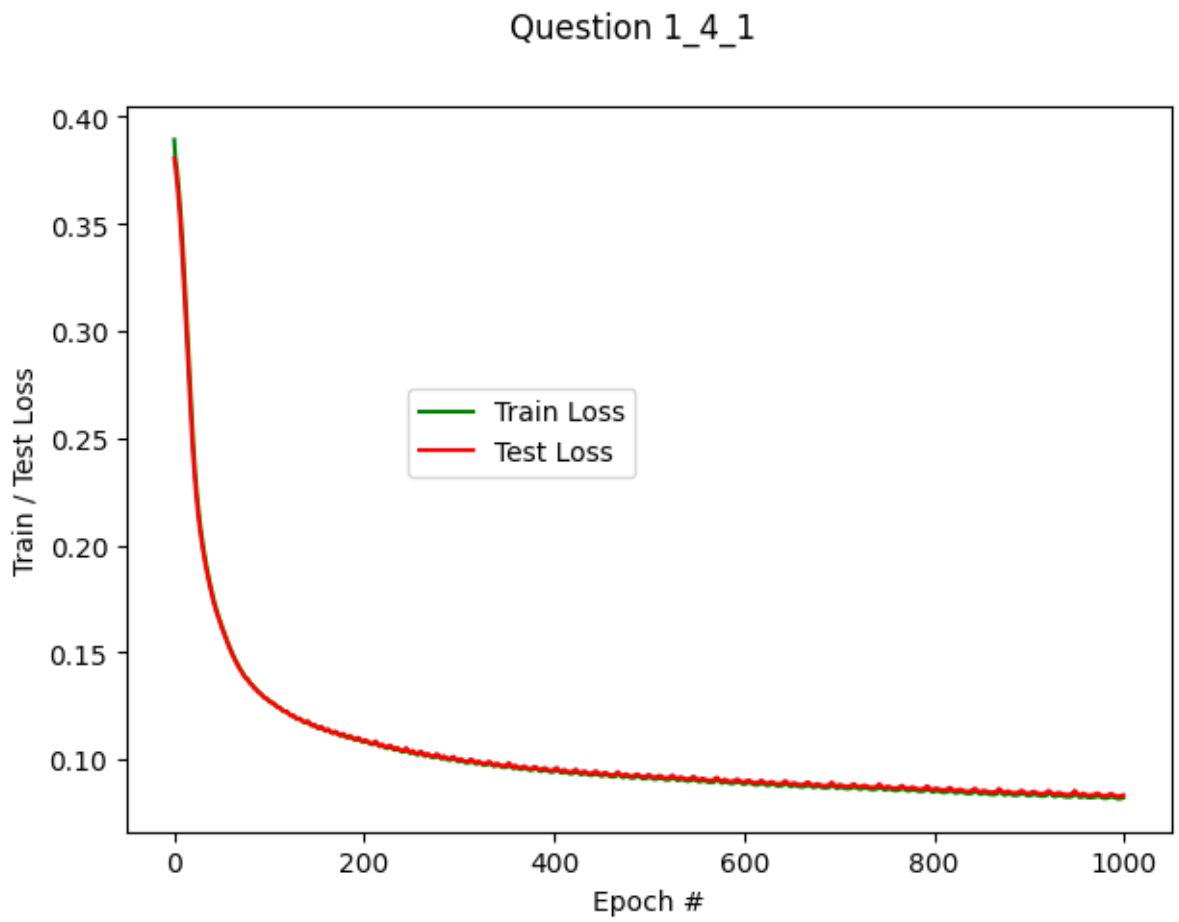


Figure 24: Learning curve, train, and test loss curve of autoencoder Q 1_4_1

The losses are captured and traced for training and validation data, the network seems to work very fine, and no signs of overfitting can be seen from the plot

(IV)

Autoencoders (AE):

These networks are used to copy the inputs to the outputs and the main applications are Dimensionality reduction, Image denoising, Image compression, Feature extraction etc, all these applications are used during the pre-processing stage of the model life cycle.

Main disadvantages of autoencoders are,

- The autoencoders are used as plug in models and are used in pre-processing stage of the ML lifecycle, so the autoencoders will have to be trained using large amount of data which adds greatly to the total processing time of the model, training includes hyperparameter tuning and model validation too before being piped to the main model
- The autoencoder will try to gather as much information as it can which will greatly impact the relevant information which will be lost in the process
- The compression or dimensionality reduction eliminates the important information that was available in the input data although this won't occur to humans
- It is limited to the application it is designed for and need to be retrained for each application
- Autoencoders cannot be used to generate new data since the latent space used in decoding is not continuous, hence variational auto encoders are used which is discussed below.

Variational Autoencoders (VAE):

In the past few years, generative deep neural networks have seen great improvements and are able to produce highly realistic data such as text, audio, or images. The two main deep generative neural network families are Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs)

VAE an autoencoder whose recognition encodings are regularised during the training to make sure that its latent space has good features and are continuous which help in generating quality new data

VAE is a combination of two parameterized models and loss function

1. Encoder or Recognition Model
2. Decoder or Generative Model
3. Loss function – Latent Loss and reconstruction loss

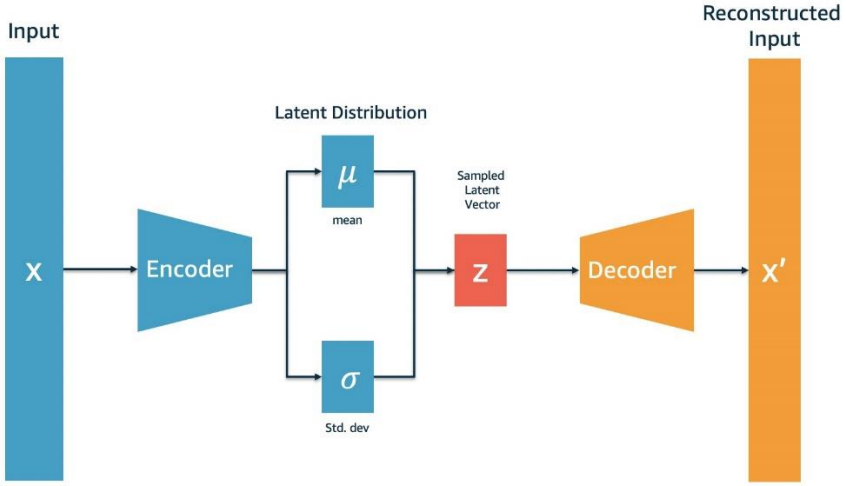


Figure 25: Basic flow of VAE [4]

These models support each other or coupled together, the encoder or recognition model is a stochastic function of the input data and delivers approximation over posterior layers to sampled latent vector layer, these are used in updating the parameters in the decoder.

The layers between the initial data or input space to the bottleneck or the encoded space is called as latent space, hence the information that is lost in the latent space cannot be recovered by the decoder. The goal of the encoder is to maintain maximum information of the input data which helps in reducing reconstruction loss.

If the encoder or the latent space is regular or well organised, the decoder can select any point from the latent space and generate new content, hence the name generative model for decoder as the principle is the same as the generator of a GAN.

For the VAE's, regularization is introduced in the latent space to mitigate overfitting of the autoencoder, this is done by encoding the input space as a distribution over the latent space and not a single point. The figure below shows the difference between AE and VAE

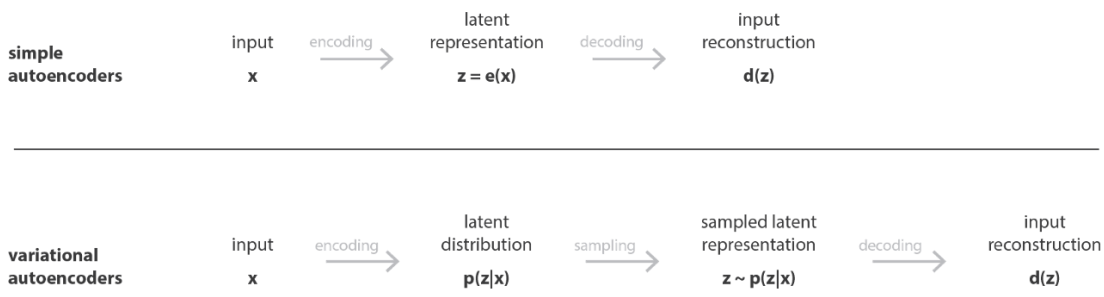


Figure 25: Comparing AE (top) and VAE (bottom) [2]

The encoder is normally distributed to train the encoder to return mean and covariance, this is to ensure that the latent space can be regularised both locally and globally. The loss function is composed of two terms, reconstruction term (final layer) and regularisation term (latent layer). The final layer reconstruction term is used to improve encode-decode scheme performance and the regularisation term is used to make the latent space organised and regular by making sure that the distribution is standard normal distribution [2]

The regularisation term is Kulback-Leibler (KL) divergence between the standard normal distribution and the output distribution of the latent space, this can be expressed in term of covariance and means of two distributions. We will now discuss the general architecture of VAE.

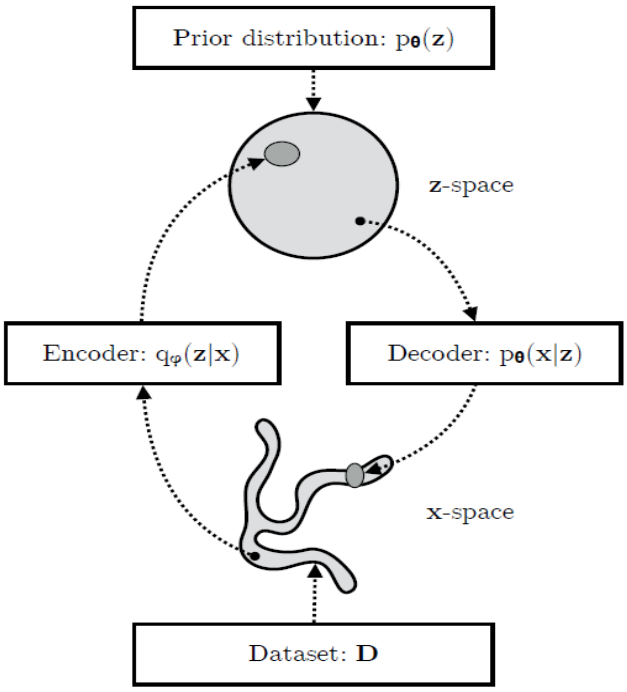


Figure 26: General structure of VAE

In the figure above, the x-space has empirical distribution, which is complicated, but the Z-space has a simple distribution such as a sphere in the figure. The model learns a joint distribution often given by p(x,z) = p(z) p(x|z) where p(z) is the prior distribution over latent space (sphere) and p(x|z) is the stochastic decoder.

The loss function is given by the equation shown below, the first part of the equation is Evidence Lower Bound (ELBO) and the second part is KL divergence between the stochastic encoder and decoder. As discussed earlier, the KL divergence measured the distance between approximate posterior and true posterior, also the gap between L(x) or ELBO and log(p(x)) (this is called the tightness of the bound) [3]

$$\mathcal{L}_{\theta,\phi}(\mathrm{x}) = \log p_\theta(\mathrm{x}) - D_{KL}(q_\phi(\mathrm{z}|\mathrm{x})||p_\theta(\mathrm{z}|\mathrm{x}))$$

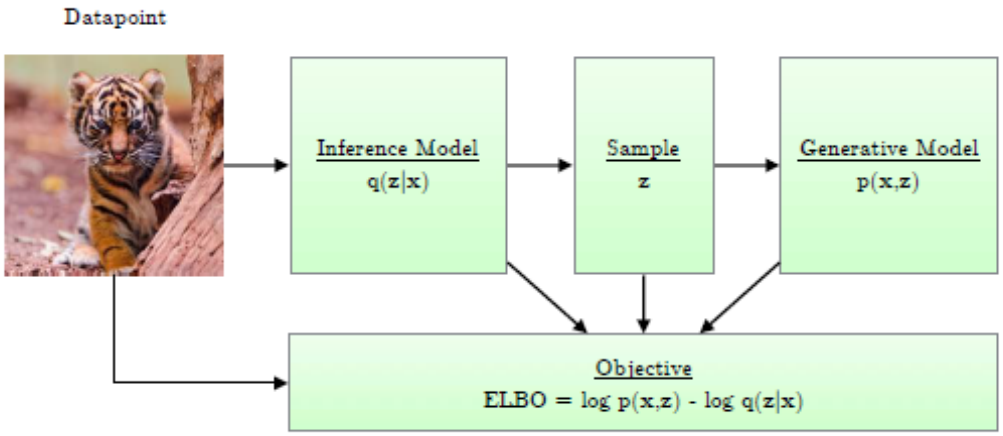


Figure 27: Simple flow of a VAE model [3]

To train any model, we need to upgrade the gradients on every loop, this cannot be done using the original form since backpropagation is not possible on a sampled latent vector, hence Reparameterization Trick (RT) is used.

In the figure below, in the original form we cannot differentiate 'f' w.r.t Φ, as backpropagation through random variable z is not possible, therefore we externalise the random variable z as a deterministic differentiable parameter x and a new random variable ε, this allows backpropagation via z and compute the gradients.

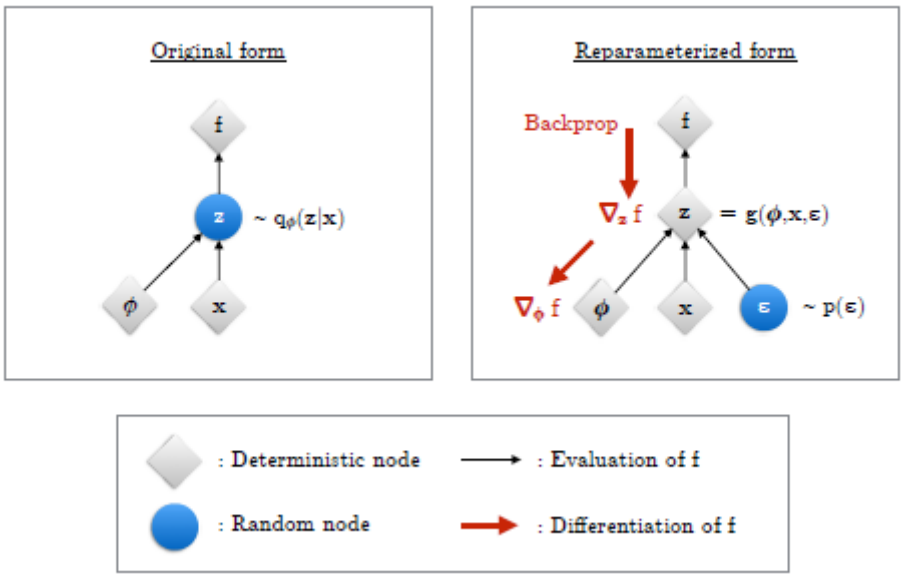


Figure 28: Original form vs Reparametrized form

Now that we have a backpropagation path to upgrade the gradients, the model can be trained to learn the input data and generate new data

Question 2 – Research

ADAM Optimization Algorithm:

The ADAMs algorithm which stands for ADAptive Moment Estimation (ADAM) is a combination of 'Gradient Descent with Momentum' and 'Root Mean Square Prop'

Gradient Descent with Momentum:

If we are trying to optimise a Cosine function as an example, red dot in the figure denotes the minimum cost that we want to attain, and the blue line oscillations are the gradient descent where the weights and bias are upgraded on every epoch

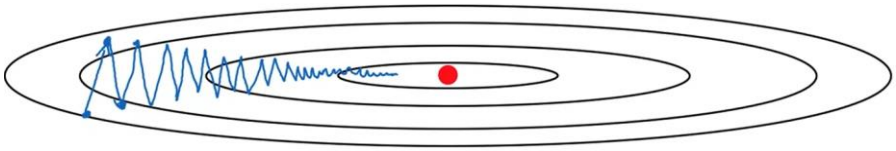


Figure 29: Contour for cost function of a cosine function [1]

We are moving very slowly towards the centre because the learning is more on the vertical axis, if we can move faster on the horizontal axis, we will reach the red dot faster.

This is done using momentum, the steps are as follows, on each mini-batch or batch gradient descent,

1. Compute the moving average of the derivatives of the weights dW and bias dB
    a. $Vdw = \beta\, Vdw + (1-\beta)\, dw$
    b. $Vdb = \beta\, Vdb + (1-\beta)\, db$
2. Update the weights and bias
    a. $W = W - \alpha\, Vdw$
    b. $b = b - \alpha\, Vdb$

This will average out the gradients in the vertical direction and increase the learning in the horizontal direction, it can be compared to a ball that is rolling downhill gaining momentum. The extra hyperparameter is β is the parameter controlling the exponentially weighted average, the common value used in ML community is 0.9 (average on previous ~10 gradients)

RMS Prop:

The idea is same as momentum, to reduce the learning in vertical direction and increase it in horizontal direction. The steps to implement RMS prop are listed below, on every epoch,

1. Compute dW and bias dB
    a. $Sdw = \beta\, Sdw + (1-\beta)\, (dw)^2$
    b. $Sdb = \beta\, Sdb + (1-\beta)\, (db)^2$
2. Update the weights and bias
    a. $W = W - \alpha\, (dw/(\sqrt{Sdw}+\varepsilon))$
    b. $b = b - \alpha\, (db/(\sqrt{Sdb}+\varepsilon))$

Sdw will be relatively small and Sdb will be relatively large, hence this makes the learning move in horizontal direction faster compared to vertical direction. The extra hyperparameters are ε and β

ADAMs Optimization Algorithm:

This algorithm has performed very well compared to other optimization algorithms,

The steps are as follows

1. Set Vdw, Vdb, Sdw, Sdb to zero
2. On every epoch compute moving averages for weights and bias (momentum step)
    a. $Vdw = \beta_1 Vdw + (1-\beta_1)\, dw$
    b. $Vdb = \beta_1 Vdb + (1-\beta_1)\, db$
3. RMS prop step
    a. $Sdw = \beta_2 Sdw + (1-\beta_2)\, (dw)^2$
    b. $Sdb = \beta_2 Sdb + (1-\beta_2)\, (db)^2$
4. Compute corrected Vdw,Vdb,Sdw,Sdb
    a. $Vdw\_corr = Vdw / (1-\beta_1^{\text{epoch no}})$
    b. $Vdb\_corr = Vdb / (1-\beta_1^{\text{epoch no}})$
    c. $Sdw\_corr = Sdw / (1-\beta_2^{\text{epoch no}})$
    d. $Sdb\_corr = Sdb / (1-\beta_2^{\text{epoch no}})$
5. Update the weights and bias
    a. $W = W - \alpha\, (Vdw\_corr / (\sqrt{Sdw\_corr} + \varepsilon))$
    b. $b = b - \alpha\, (Vdb\_corr / (\sqrt{Sdb\_corr} + \varepsilon))$

The algorithm has four hyperparameters ε, β_1, β_2, α. The common values used are $\varepsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, α is the learning rate that needs to be tuned

Skip Connections:

The deep neural networks which go very deep or have many layers in its architecture have a problem with vanishing or exploding gradients, to mitigate this major issue, we use skip connection, the activations of one layer is fed into another layer which is much deeper in the network

A good example is the Residual Networks which is very deep neural network sometimes consists of over 100 layers

Residual block:

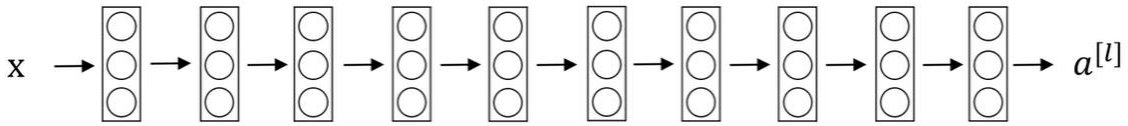


Figure 30: Plain Network [1]

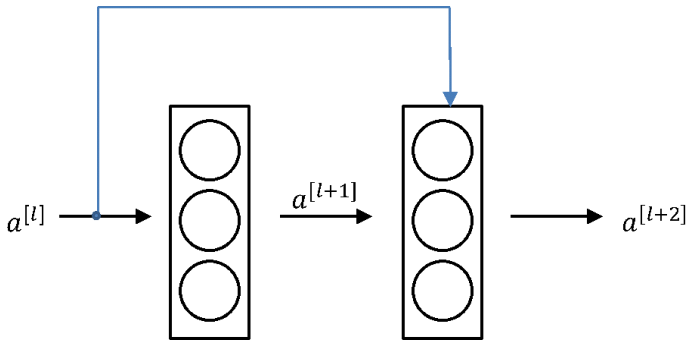


Figure 31: Residual block with skip connections

The figure 30 is a plain network, if we train this network using gradient descent, initially the training error will decrease but with increasing layers it will start to increase, in theory having a deeper network should help reduce the training error but, it gets worse as the number of layers increase as shown in the figure 32

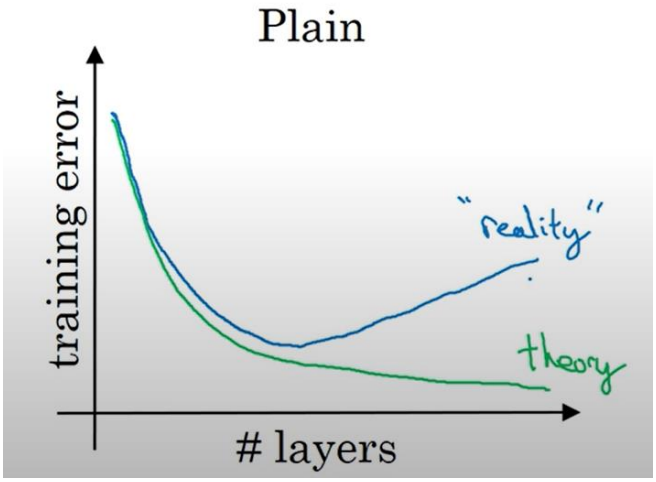


Figure 32: Theory and reality is different when training plain deeper neural networks [1]

But when the skip connections are used (ResNet), we can train deeper networks with 100 to 1000 layers without loss in performance.

From figure 31, the activations of layer 2 are given by

$$A[L+2] = g(Z[L+2] + A[L])$$

where A[L] is the skip connection from input layer 0 activations to layer 2, it can then be expanded

$$A[L+2] = g(W[L+2] *A[L+1] + B[L+2] + A[L])$$

If the Weights and Bias at layer 2 ~0, the output for A[L+2] = g(A[L]), hence it is easy for residual networks to learn the identity functions even when the model is very deep, once thing we must be careful of is the dimension of the layers, this can be done using zero padding etc. usually the resents will have same dimensions as shown below

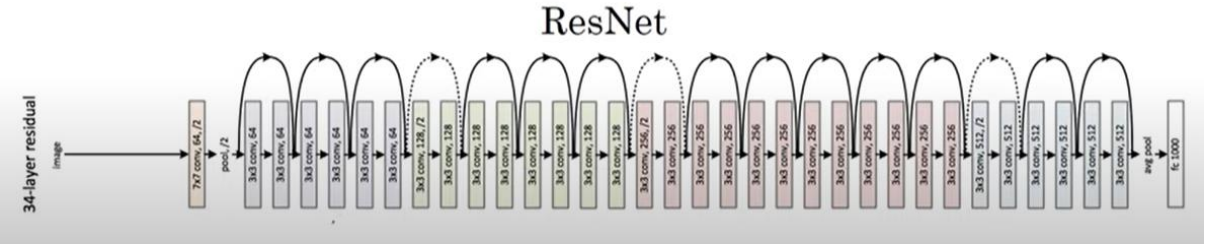


Figure 33: ResNet with 3x3 convoluition in all layers

References

[1] Deep Learning Specialization by Andrew Ng, Coursera

[2] https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

[3] Diederik P. Kingma; Max Welling, An Introduction to Variational Autoencoders, now, 2019.

[4] https://aws.amazon.com/blogs/machine-learning/deploying-variational-autoencoders-for-anomaly-detection-with-tensorflow-serving-on-amazon-sagemaker/