Deep Learning – TensorFlow/Keras Assignment

Name: Shane Quinn Date: 10/04/2021

Student Number: R00144107

Question 1 – TensorFlow and the Low-Level API

Part (i) – Neural Networks

In this section we will look at the main functions used to implement the 2 neural network architectures described in the assignment specification.

Forward Pass

```
def forward_pass(X, w1, w2, b1, b2):
    """
    Push feature data through neural network. Returns 10 class probabilities for all feature instances

    Parameters
    ----------
    X : tf.Variable
        Pre-processed input data.
    w1 : tf.Variable
        Layer 1 learnable weights.
    w2 : tf.Variable
        Layer 2 learnable weights.
    b1 : tf.Variable
        Layer 1 bias.
    b2 : tf.Variable
        Layer 2 bias.

    Returns
    -------
    H : tf.Variable
        Softmax layer output predicted probability of each class.

    """

    #Layer 1 - 200 Relu Neurons
    A = tf.matmul(w1, tf.transpose(X)) + b1                    #A1 = x1.X + b1
    H = tf.keras.activations.relu(A)                           #H1 = act(A1)
    #Layer 2 - Softmax Layer
    A = tf.matmul(w2, H)+b2                                    #A2 = w2.H1 + b2
    #Softmax = (e^A2)/sum(A2)
    H = tf.exp(A) / tf.reduce_sum(tf.exp(A), axis=0)           #H2 = act(A2)

    return H
```

Figure 1 – Forward pass function

The forward pass function is designed to take in feature data and push it through our neural network returning the output of the final SoftMax layer which will give us the predicted probabilities of each feature instance to each possible class. In the code seen above in Figure 1 is related to network architecture A in the assignment specification, we have 2 layers:

- Layer 1: Is comprised of 200 Relu activation neurons
- Layer 2: Is comprised of a SoftMax layer which in this 10-class classification problem, meaning there is 10 SoftMax activated neurons in this layer.

This design is a basic densely connected neural network our inputted feature data is passed through layer 1 first, which has 200 ReLu activated neurons. Rectified linear unit (ReLu) activation function is designed to allow the neural network learn nonlinear patterns, each ReLu activated neuron takes an arbitrary input and returns an output between 0 and infinity (negative numbers become 0). ReLu is a popular activation function because it is computationally lightweight.

The output of layer 1 is piped into layer 2 which is a SoftMax layer. This transforms input feature into ‘n’ class probabilities where ‘n’ is the number of classes in the classification problem (in our case 10). These probabilities denote the likelihood of each feature being each class label.

Cross Entropy Loss

```
def cross_entropy(pred_y, y):
    """
    Take in softmax probabilities (output of forward_pass) and true class labels and calculate cross entropy loss

    Parameters
    ----------
    pred_y : tf.Variable
        Predictions (Output of softmax layer in forward_pass()).
    y : tf.Variable
        One-hot encoded true class labels.

    Returns
    -------
    cross_ent : Cross entropy loss
        tf.Variable.

    """

    #Cross entropy loss per class = -sum((True class encoded values)*log(predicted probabilities))
    a = -tf.reduce_sum(y * tf.math.log(pred_y), axis=0)
    #Cross entropy loss = mean of all losses calculated above.
    cross_ent = tf.reduce_mean(a, axis=0)

    return loss
```

Figure 2 – Cross entropy function

The cross-entropy loss function calculates the cross-entropy loss of the model using the output of the forward pass function (in this case 10 class SoftMax output, or 10 class probabilities for each instance) and the true class values. This is achieved by satisfying the following equation

$$\frac{\sum -y * \log(pred_y) - (1 - y)\log(1 - pred_y)}{n}$$

Where ‘n’ is the number of training examples. With cross entropy loss if the target class value is 0 the first portion ($y * \log(pred_y)$) is disregarded (multiplied by 0), in this case however the second portion is taken into account. This way only one or the other is used, depending on the true class label.

However, in this implementation while including the second portion of the equation the loss was driven to 0 or NaN while training, leading to the model stopping training. Removing the effect of all except the true class label mitigated against this so our new equation is:

$$\frac{\sum -y * \log(pred_y)}{n}$$

Disregarding these values allowed the model to train to a more optimal solution.

Calculate Accuracy

```
def calculate_accuracy(pred_y, y, datatype=tf.float32):
    """
    Calculate the model accuracy given predicted probabilities and true class labels

    Parameters
    ----------
    pred_y : tf.Variable
        Predicted class probabilities, output of forward pass/softmax layer.
    y : tf.Variable
        True class values.
    datatype : tf.float32/tf.float64, optional
        One of the above tf datatypes. The default is tf.float32.

    Returns
    -------
    accuracy : float32
        Model Accuracy.

    """

    # Convert predicted probabilities to 0 or 1
    pred_y = tf.round(pred_y)
    # Boolean True (1) if prediction is correct, cast to tf.Variable
    predictions = tf.cast(tf.equal(pred_y, y), datatype)
    #Mean value of correct predictions
    accuracy = tf.reduce_mean(predictions)

    return accuracy
```

Figure 3 – Calculate accuracy) function

The calculate accuracy function is designed to take in the output of forward pass (SoftMax output) and calculate the model accuracy by first converting the predicted probabilities from the SoftMax layer to 0 or 1. This is compared to the true class labels of 0 or 1 and cases where the prediction was correct we save a Boolean 1, in cases where the prediction was incorrect we save a Boolean 0. This array is then casted to a tf.Variable and the mean is found. The accuracy (mean correct predictions) is then returned.

```python
def main():

    #Retrieve feature data/class labels
    X, y, X_val, y_val = pre_process()

    #Initialise Learning rate and iterations.
    learning_rate = 0.05
    iterations = 2000
    datatype = tf.float64

    #Initialise lists for saving accuracies/loss
    te_acc = []
    tr_acc = []
    te_loss = []
    tr_loss = []

    # Create tf variables from data
    X = tf.cast(X, datatype)
    y = tf.cast(y, datatype)
    X_val = tf.cast(X_val, datatype)
    y_val = tf.cast(y_val, datatype)

    #Initialise Adam Optimizer
    adam = tf.keras.optimizers.Adam()

    #Initialise weights and bias
    zeros = tf.zeros_initializer()
    layer1_weights = tf.Variable(tf.random.normal([200,784], stddev=0.05, dtype=datatype))
    layer2_weights = tf.Variable(tf.random.normal([10, 200], stddev=0.05, dtype=datatype))
    layer1_bias = tf.Variable(0, dtype=datatype)
    layer2_bias = tf.Variable(0, dtype=datatype)

    #Repeat gradient descent loop 'iterations' times
    for i in range(iterations):

        with tf.GradientTape() as tape:
            #Create instance of gradient tape to record forward pass and calculate gradients for learnable weights and biases
            pred_y = forward_pass(X, layer1_weights, layer2_weights, layer1_bias, layer2_bias)
            loss = cross_entropy(pred_y, y)

        tr_loss.append(loss)
        gradients = tape.gradient(loss, [layer1_weights, layer2_weights, layer1_bias, layer2_bias])     #Calculate gradients
        accuracy = calculate_accuracy(pred_y, y, datatype)                                              #Calculate accuracy of model
        tr_acc.append(accuracy)
        print("Iteration {}: Training Loss = {} Training Accuracy = {}".format(i, loss.numpy(), accuracy.numpy()))

        #Apply gradients using adaptive movement estimation, see accompanied report for more details
        adam.apply_gradients(zip(gradients, [layer1_weights, layer2_weights, layer1_bias, layer2_bias]))

        #Test model on validation data
        test_pred_y = forward_pass(X_val, layer1_weights, layer2_weights, layer1_bias, layer2_bias)
        test_loss = cross_entropy(test_pred_y, y_val)
        te_loss.append(test_loss)
        te_acc.append(calculate_accuracy(test_pred_y, y_val, datatype))
```

Figure 4 – Gradient descent loop

In our gradient descent loop we first record a forward pass using gradient tape, this allows us to calculate gradients.

The use of gradient tape enables us to take advantage of TensorFlow's automatic differentiation functionality. This is achieved by recording the operations within the gradient tape and later pulling the partial derivatives with respect to each learnable weight and bias. The gradients are then passed into our ADAM optimizer which updates all our learnable parameters. ADAM optimizer is discussed in detail in a later section.

Performance Evaluation

Below we can see the training and validation loss/accuracy plotted for 1000 epochs for network architecture A and B

Network architecture A consists of 2 layers:

- 200 ReLu activated neurons
- Softmax

Network architecture B consists of 3 layers:

- 300 ReLu activated neurons
- 100 ReLu activated neurons
- Softmax layer

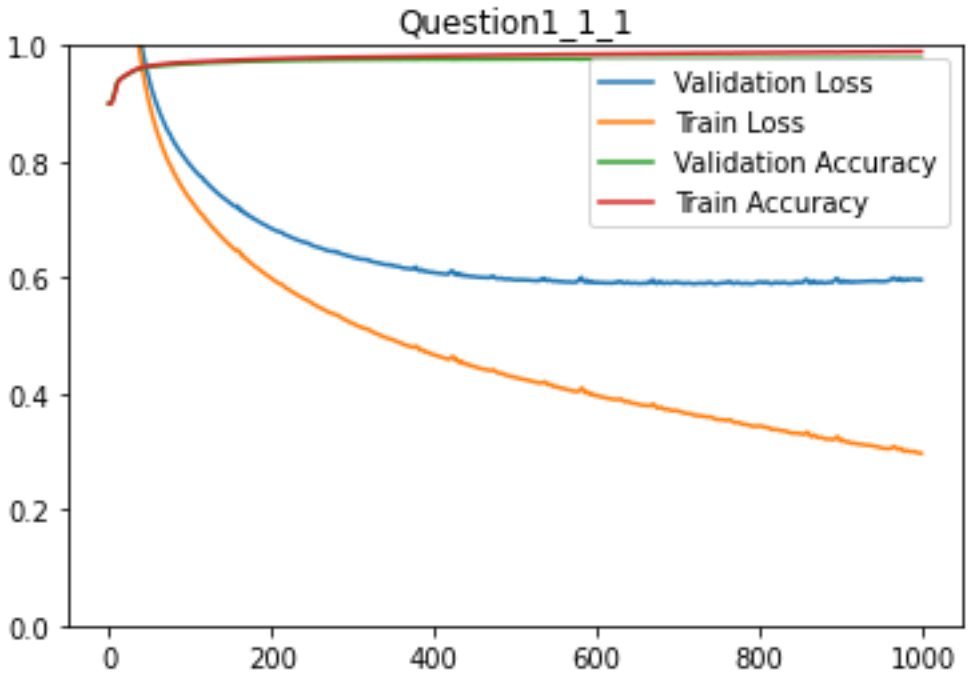

Figure 5 - Question 1_1_1 (execution time: 256 seconds)

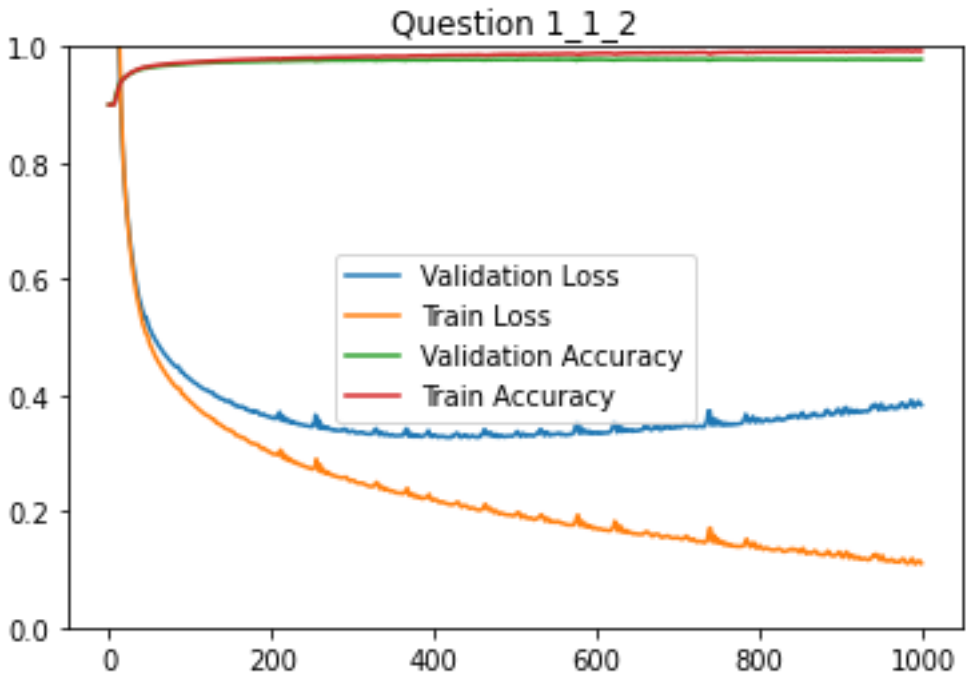

Figure 6 - Question 1_1_2 (execution time: 384 seconds)

We can see at the 100-epoch mark overfitting starts to take place as the training and validation loss begin to diverge. To combat overfitting, some regularisation techniques can be introduced, in the next section we will discuss the advantages of including drop out regularisation.

From the graphs above we can also see that although overfitting occurs sooner into training of network architecture 2, the validation/training loss is lower than in architecture 1. From this we can infer that the increase in dimensionality of the neural network or addition of layers can increase the chances of overfitting occurring but also increase the performance of the model.

Part (ii) – Dropout

In this section we examined the effect of introducing regularization in the form of a dropout layer into the neural network. Below is the new forward pass function with a dropout layer introduced after layer 1.

```
def forward_pass(X, w1, w2, w3, b1, b2, b3, dop=0.1):
    """
    Push feature data through neural network. Returns 10 class probabilities for all feature instance

    Parameters
    ----------
    X : tf.Variable
        Pre-processed input data.
    w1 : tf.Variable
        Layer 1 learnable weights.
    w2 : tf.Variable
        Layer 2 learnable weights.
    w3 : tf.Variable
        Layer 3 learnable weights.
    b1 : tf.Variable
        Layer 1 bias.
    b2 : tf.Variable
        Layer 2 bias.
    b3 : tf.Variable
        Layer 3 bias.
    dop : Float32, optional
        Drop out probability, probability of each neuron to be dropped. The default is 0.1.

    Returns
    -------
    H : tf.Variable
        Softmax layer output predicted probability of each class.

    """


    #Layer 1- 300 Relu Neruons
    A = tf.matmul(w1, tf.transpose(X)) + b1                    #A1 = w1.X + b1
    H = tf.keras.activations.relu(A)                           #H1 = act(A1)
    #Layer 2 - Dropout Layer
    datatype = w1.dtype                                        #Get datatype used (float32/64)
    probThres = 1-dop                                          #Threshold of neurons to be dropped
    #Create matrix of shape H with boolean 1s in probThresh locations, and 0s in dop locations
    dropMat = np.random.rand(H.shape[0], H.shape[1]) < probThres
    #Cast to tf variable for interacting with H
    dropMat = tf.cast(dropMat, datatype)
    #Elementwise multiply mask matrix by H, 'dropping' neurons
    dropResult = tf.math.multiply(H, dropMat)
    #Scale up values in H to compensate for dropped values
    H = tf.math.divide(dropResult, probThres)
    #Layer 3 - 100 Relu Neurons
    A = tf.matmul(w2, H) + b2                                  #A2 = w2.H1 + b2
    H = tf.keras.activations.relu(A)                           #H2 = act(A2)
    #Layer 4 - Softmax = (e^A2)/sum(A2)
    A = tf.matmul(w3, H)+b3                                    #A3 = w3.H2 + b3
    H = tf.exp(A) / tf.reduce_sum(tf.exp(A), axis=0)           #H3 = act(A3)

    return H3
```

Figure 7 – Forward pass function with dropout layer

Our forward pass with dropout layer is shown above. This function is like the forward pass function discussed earlier however here we include a drop out layer after the first layer. In the dropout layer we first create a dropout matrix or mask matrix which is the same shape as the output of the previous layer. Each element in this mask matrix has DOP (drop out probability) chance of being 0 (dropped), all other elements become 1 and pass through. We (elementwise) multiply this matrix with the output of our previous layer, 'dropping' certain values from progressing to the next layer. We then (elementwise) divide by the probability threshold (1-DOP) to scale up values and compensate for dropped values.

Below, several graphs can be seen describing the effect of increasing the drop out probability (DOP) on training and test accuracy/loss.

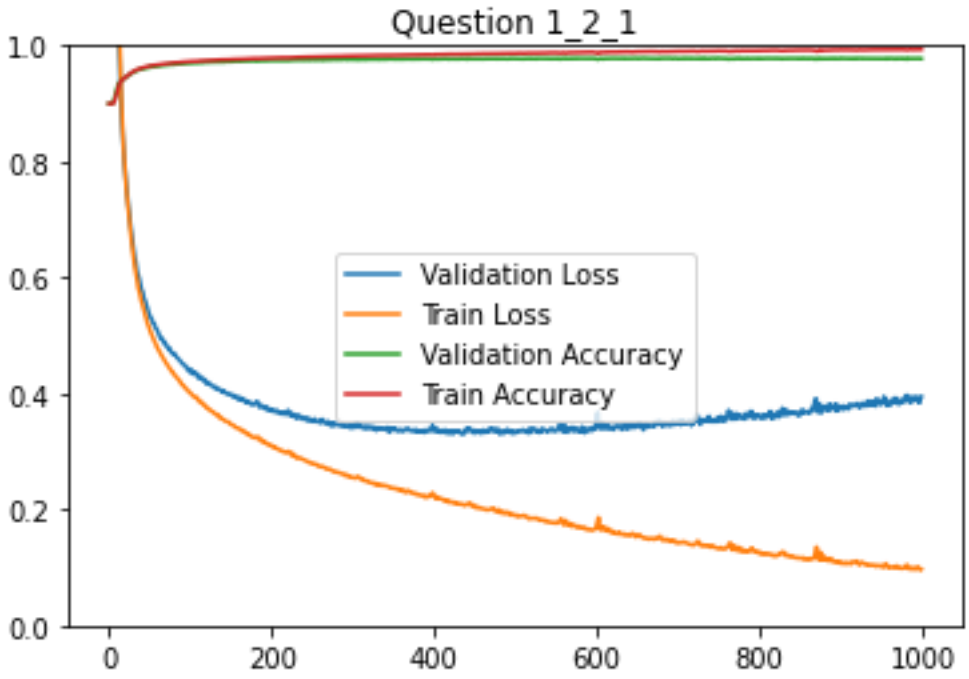

Figure 8 – Question 1_2_1, ***DOP=0.05*** *(execution time: 427 seconds)*

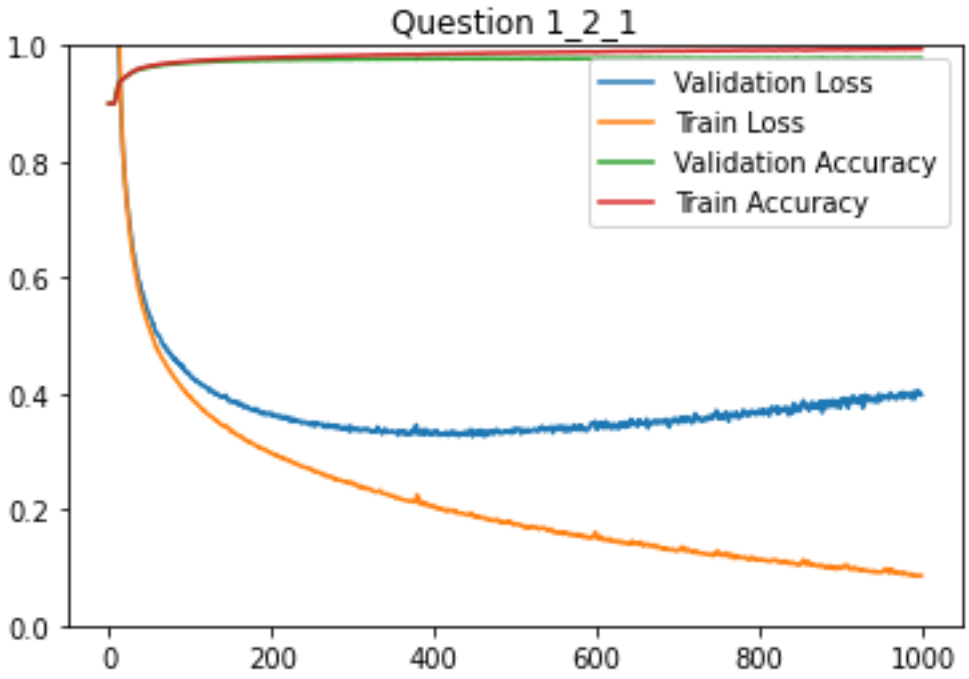

*Figure 9 – Question 1_2_1, **DOP=0.1** (execution time: 424 seconds)*

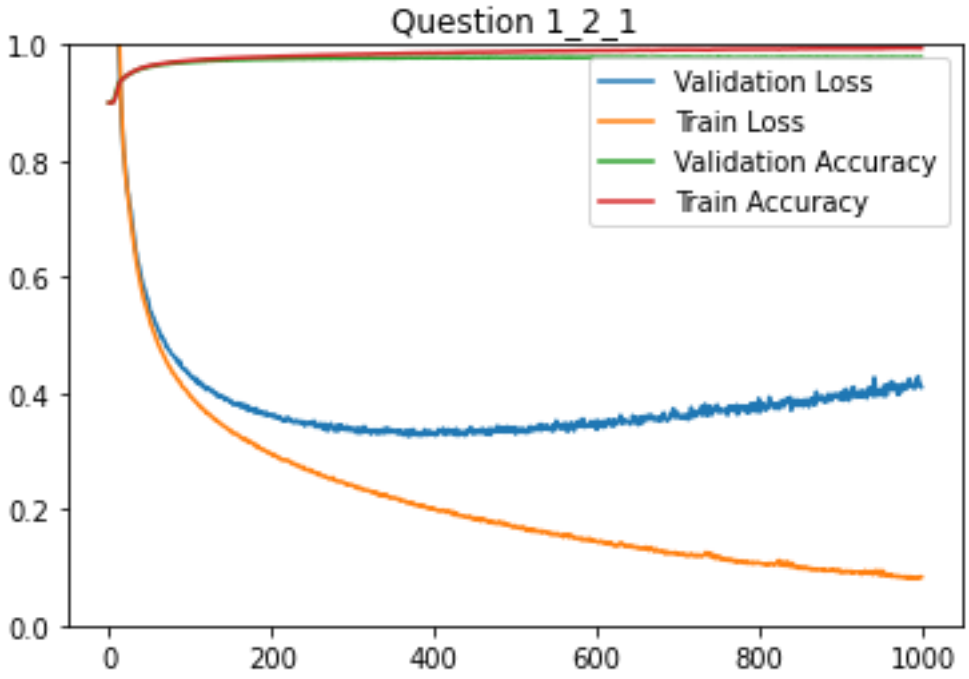

*Figure 10 – Question 1_2_1, **DOP=0.25** (execution time: 422 seconds)*

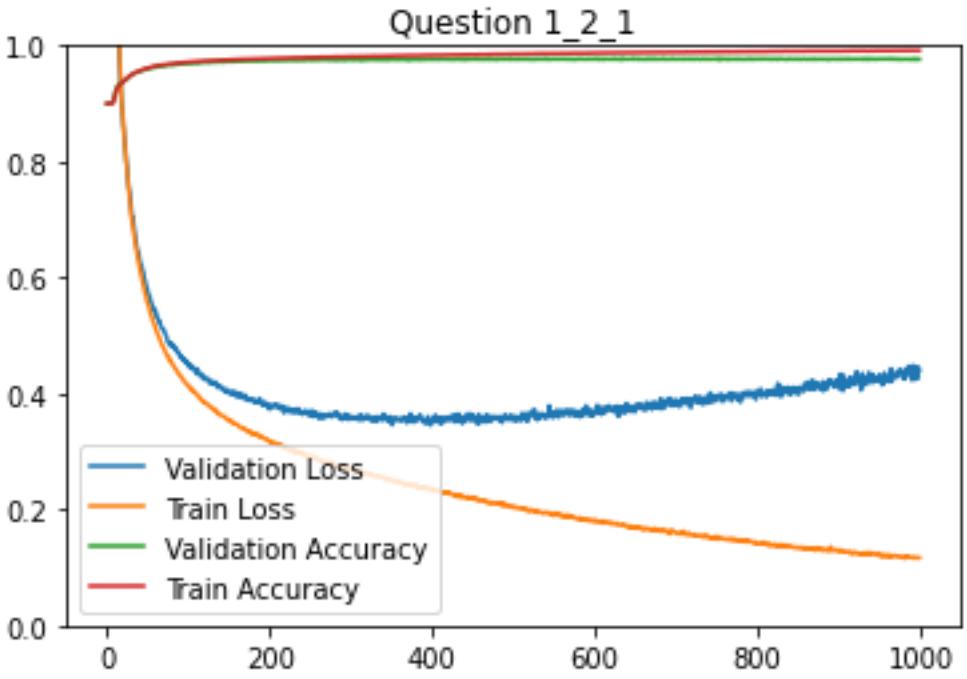

*Figure 11 – Question 1_3_1, **DOP=0.5** (execution time: 424 seconds)*

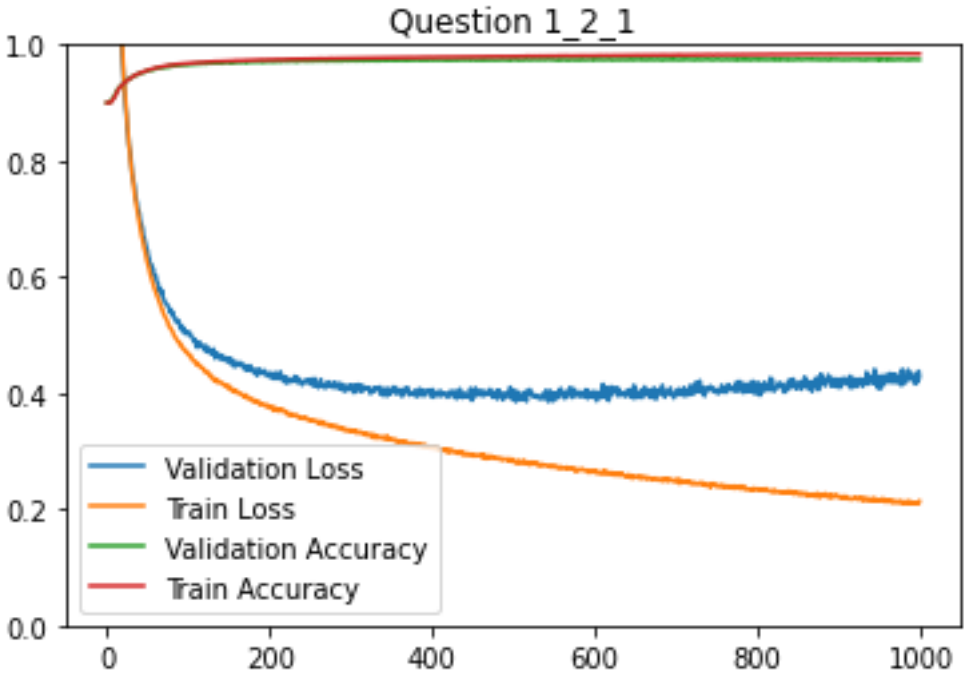

*Figure 12 – Question 1_2_1, **DOP=0.75** (execution time: 425 seconds)*

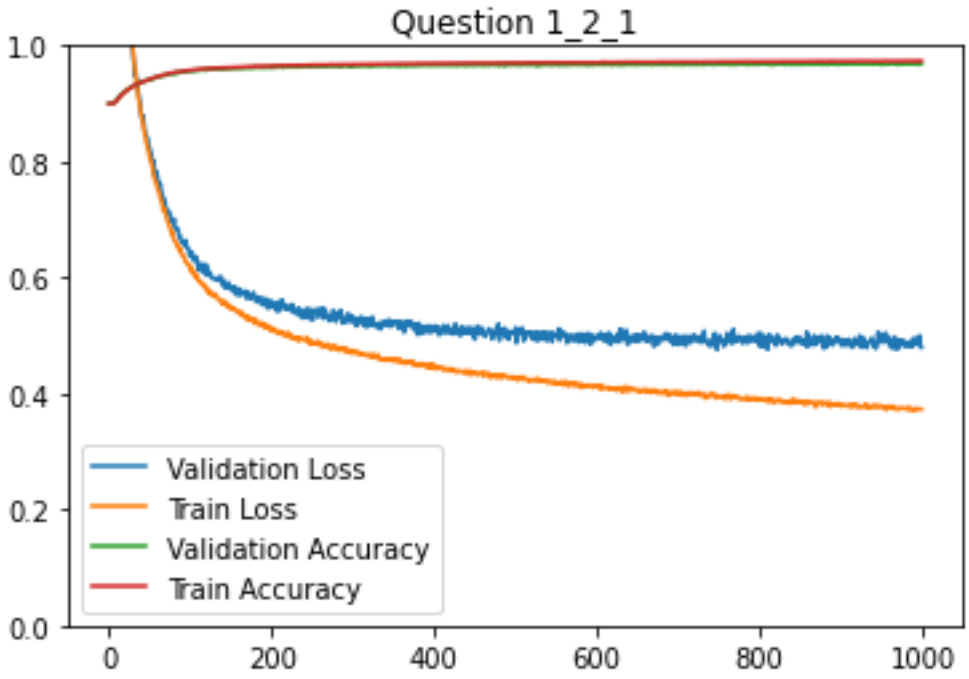

*Figure 13 - Question 1_2_1, **DOP=0.9** (execution time: 420 seconds)*

We can see from the graphs above that as we increase the drop out probability the effect of overfitting is decreased. This is evident when comparing Figure 13 (DOP = 0.9) to Figure 8 (DOP = 0.05), the most extreme example. With respect to the model with a very low drop out probability, the training loss improves constantly while the validation loss begins to diverge and increase after several epochs. This is a sign of over-fitting. The model with a high drop out probability still has a certain degree of overfitting, the training and validation loss still diverge, however the validation loss does not increase. They begin to diverge after a greater number of epochs.

Drop out regularisation prevents a model from becoming overly reliant on any one neuron.

Part (iii) – Autoencoder Implementation

The objective of this autoencoder is to attempt to eliminate noise from images. Images are taken from the Fashion MNIST dataset and processed to include noise, the autoencoder implemented removes noise from images by passing noisy images through our neural network architecture defined below. The loss function compares the encoded images to the original images using mean squared error.

Forward Pass

```
def forward_pass(X, w1, w2, w3, w4, w5, w6, b1, b2, b3, b4, b5, b6):
    """
    Push feature data through neural network. Final sigmoid activation layer
    returns flattened encoded pixel values
    between 0 and 1 (black and white).

    Parameters
    ----------
    X : tf.Variable
        Pre-processed input data.
    w1 : tf.Variable
        Layer 1 learnable weights.
    w2 : tf.Variable
        Layer 2 learnable weights.
    w3 : tf.Variable
        Layer 3 learnable weights.
    w4 : tf.Variable
        Layer 4 learnable weights.
    w5 : tf.Variable
        Layer 5 learnable weights.
    w6 : tf.Variable
        Layer 6 learnable weights.
    b1 : tf.Variable
        Layer 1 bias.
    b2 : tf.Variable
        Layer 2 bias.
    b3 : tf.Variable
        Layer 3 bias.
    b4 : tf.Variable
        Layer 4 bias.
    b5 : tf.Variable
        Layer 5 bias.
    b6 : tf.Variable
        Layer 6 bias.

    Returns
    -------
    H : td.Variable
        Flattened autoencoded pixel values between 0 and 1.

    """

    #Layer 1: 128 Relu Neruons
    A = tf.matmul(w1, tf.transpose(X)) + b1                    #A1 = w1.X + b1
    H = tf.keras.activations.relu(A)                           #H1 = act(A1)
    #Layer 2: 64 Relu Neruons
    A = tf.matmul(w2, H) + b2                                  #A2 = w2.H1 + b2
    H = tf.keras.activations.relu(A)                           #H2 = act(A2)
    #Layer 3: 32 Relu Neurons
    A = tf.matmul(w3, H) + b3                                  #A3 = w3.H2 + b3
    H = tf.keras.activations.relu(A)                           #H3 = act(A3)
    #Layer 4: 64 Relu Neurons
    A = tf.matmul(w4, H) + b4                                  #A4 = w2.H1 + b4
    H = tf.keras.activations.relu(A)                           #H4 = act(A4)
    #Layer 5: 128 Relu Neurons
    A = tf.matmul(w5, H) + b5                                  #A5 = w5.H1 + b5
    H = tf.keras.activations.relu(A)                           #H5 = act(A5)
    #Layer 6: 784 Sigmoid Neurons
    A = tf.matmul(w6, H) + b6                                  #A6 = w6.H5 + b6
    H = tf.sigmoid(A)                                          #H6 = sigmoid(A6)

    H = tf.transpose(H)

    return H
```

Figure 14 - Forward pass function

The autoencoder forward pass function is like the forward pass function discussed above in Question 1. The main difference being the number of layers and neurons within each layer.

Autoencoders first encode input data into a reduced dimension array (in our case 32 neurons). This is the first stage of autoencoding, the encoding stage. The next stage is to decode this into a higher dimensional space (in our case 784 neurons, flattened image, the same as our input data).

This allows the network to learn the defining features of the input data

Mean absolute error

```
def mean_absolute_error(X_encoded, X_true):
    """
    Returns mean absolute error given encoded image and true image data

    Parameters
    ----------
    X_encoded : tf.Variable
        DESCRIPTION.
    X_true : tf.Variable
        DESCRIPTION.

    Returns
    -------
    mae : float
        Mean absolute error.
    """

    # Find absolute error between all true and encoded images.
    # Find mean by finding sum of all absolute errors and dividing by total number of images
    mae = tf.math.divide(tf.reduce_sum(abs(tf.math.subtract(X_true, X_encoded))), X_encoded.shape[0])

    return mae
```

Figure 15 – Mean absolute error function

The mean absolute error function calculates the mean absolute error by the following steps

1. Find the absolute error between the real and encoded images
2. Find the mean of these errors by dividing the sum of all errors by the total number of images.

Note, the mean absolute error always returns a positive number due the use of absolute error instead of just the error.

Below we can see 5 images in 3 different states

- State 1: Encoded
    - Here the noisy images have been passed through the autoencoder
- State 2: Noisy
    - Here we have the noisy images used as input for our autoencoder
- State 3: Original images
    - Original image before noise was added.

As we can see when comparing the encoded images to the original images, they share a general shape, however some of the finer details are lost in the encoding and decoding process.

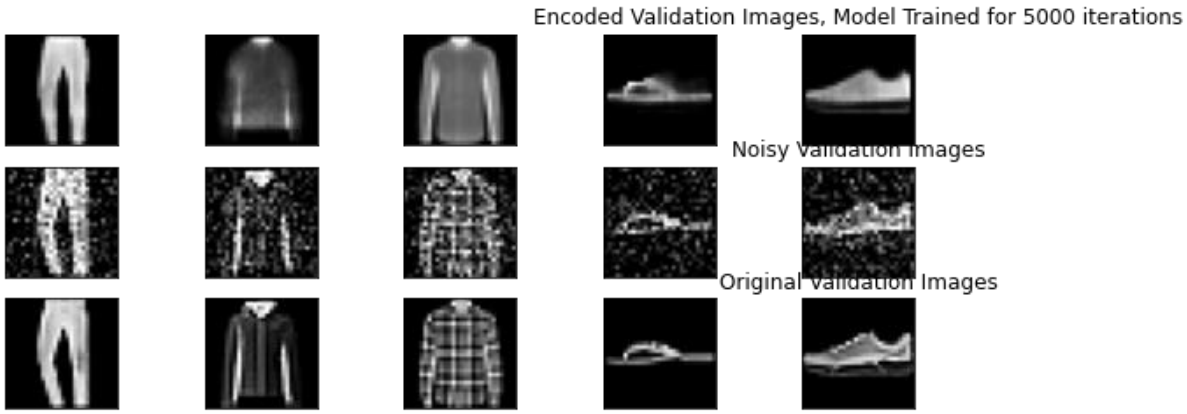

Figure 16 – Encoded, noisy and original validation images

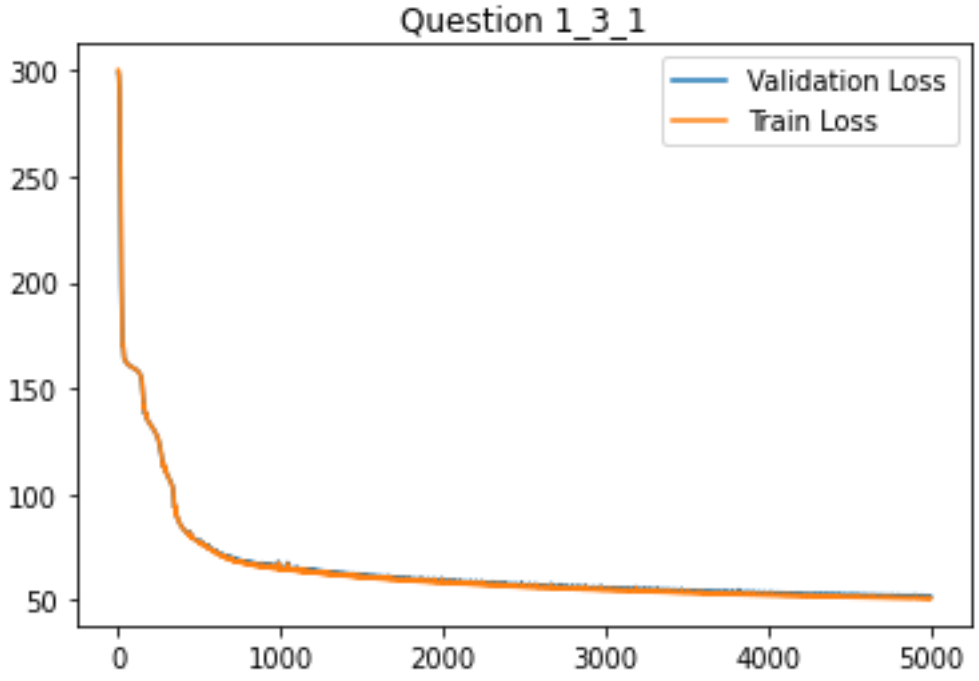

Figure 17 - Autoencoder training/validation loss (execution time: 294 seconds)

Part (iv) – Autoencoder

Put simply an autoencoder learns how to make a copy of the input data (in this case Fashion MNIST images). It does this by encoding the image in question into a lower dimensional code (or bottleneck) and then decoding this or reconstructing the image from the lower dimensional code.

Autoencoders are typically used as a dimensionality reduction technique as part of the preprocessing stage of the machine learning lifecycle. In this case however we are using an autoencoder to remove noise.

Autoencoders learn lower dimensional representations of the input data. The goal of an autoencoder is to minimize reconstruction loss, or for the output image to look as similar to the input image as possible.

Disadvantages of autoencoders

While autoencoders are beneficial for the reasons mentioned above, they have a couple of draw backs.

When used as a dimensionality reduction technique autoencoders are a computationally expensive when compared to other pre-processing techniques. This is because the autoencoder must be trained and encode new data before being piped into the actual model. In our example above, with 5000 iterations the execution time (to train and test our model) was 294 seconds using Tesla T4 GPU

Autoencoders are heavily reliant on the training and validation data (and of course new data) being similar or share the same underlying features that the autoencoder learns. If unseen data is drastically different from training data, the autoencoder will perform poorly.

Autoencoders cannot be used to generate new images, this is because is that the latent space in which the decoder uses as input are not continuous. This leads to difficulties in the decoding phase where challenges arise if latent space is discontinuous. Variational autoencoders, discussed below, attempt to address this issue.

Variational autoencoder (VAE)

A variational autoencoder is a type generative model and differs from autoencoders described above in that they can generate new data, not just reconstructing data. A VAE consists of 3 parts:

- Encoder
- Decoder
- Loss Function
    - Latent loss
    - Reconstruction loss

As we can see a VAE is like an autoencoder in that they share an encoder and decoder however they differ within the loss function. A VAE includes a latent loss function which describes how latent space is organized for generating new images, or how well the model learns distributions within latent space.

Another difference is instead of treating input into our decoder as a vector it is pointed to a distribution in latent space, or an area to sample from.

A disadvantage of VAEs when compared to other generative models such as GANs, is that they can create blurry images. This is because there are 2 loss functions we are trying to optimize, latent loss and reconstruction loss. In trying to optimize both losses, VAEs can meet in the middle, leading to blurrier images than would be generated using a GAN.

ADAM Optimizer

First described in 2015 [1], the goal of ADAM is to optimize the update of learnable parameters in our neural network using gradient descent.

One of the issues with gradient descent is the picking the correct learning rate. A learning rate too small will take too long to reach the global minimum however a learning rate too high may never reach the global minimum. ADAM addresses this problem by combining RMSprop with AdaGrad.

AdaGrad deals with sparse gradients while RMSprop (root mean squared prop) attempts to speed up learning.

while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
  $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
  $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 18 – Adam algorithm [1]

The above sudo-code describes the adam optimizer flow of events. For each iteration we first retrieve our gradients (in our above implementations this was done using gradient tape and auto diff).

When implementing ADAM, there are several hyperparameters that must be configured first.

- α = needs to be tuned (learning rate)
- β1 = 0.9 (recommended by authors) – AdaGrad with momentum
- β 2 = 0.999 (recommended by authors) – RMS prop
- 3 = 10^-8 (recommended by authors) – Very small number added to denominator to avoid dividing by 0

Adam optimizer is designed for large high dimensional datasets