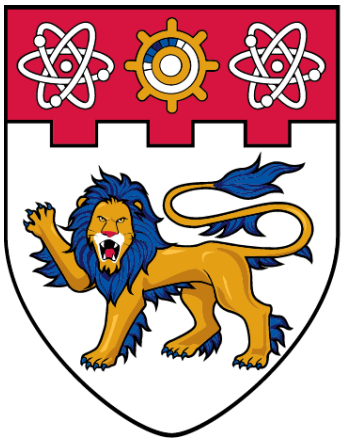

NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

CZ4042 Neural Networks and Deep Learning

Individual Assignment 1 Part A

Zhang Yuhan

U1823060F

Table of Contents

Part A: Classification Problem

Introduction

This problem aims at finding the optimal parameters (batch size, number of hidden neurons, decay parameter, and number of hidden layers) for the classification of the Cardiotocography dataset based on accuracy. Then, the performances of a 3-layer network versus a 4-layer network using the optimal parameters are analysed. The loss function used in the models for this question is the categorial cross entropy function. The dataset used in this question consists of 2126 entries classified by expert obstetricians, each entry contains a feature space of 21 and has 2 class labels. However, we will only be using the NSP label for this problem.

All figures and codes provided in this report is present in the corresponding jupyter notebook files for each question.

Methods

```
# scale data (set range min and max of 0 and 1 respectively)
def scale(X, X_min, X_max):
    return (X - X_min)/(X_max-X_min)
```

```
#read train data
train_input = np.genfromtxt('ctg_data_cleaned.csv', delimiter= ',')
trainX, train_Y = train_input[1:, :21], train_input[1:,-1].astype(int)
print("trainX max: "+str(np.max(trainX))+", trainX min: "+str(np.min(trainX)))
trainX = scale(trainX, np.min(trainX, axis=0), np.max(trainX, axis=0))
print("After scaling:")
print("trainX max: "+str(np.max(trainX))+", trainX min: "+str(np.min(trainX)))
```

```
trainX max: 269.0, trainX min: -1.0
After scaling:
trainX max: 1.0, trainX min: 0.0
```

Figure 1: Normalization

Feature scaling is an important pre-processing step in machine learning. As can see from Figure 1, the range of values in the original dataset is quite large, from -1 to 269. Because of this, the distance between data points can vary significantly, this may negatively affect the cost function and the gradient descent. Thus, the dataset X is normalized so all values lie in the range of 0 to 1.

```
#Split train and test data

train_X, test_X, train_Y, test_Y = train_test_split(trainX, trainY, test_size=0.3)
```

Figure 2: Splitting train and test data

The original dataset was split into a training set and a testing set with size ratio of 70:30. To achieve this, the train_test_split() function from the sklearn library was used, as can be see from Figure 2 above. This resulted in two training sets with 1488 entries and two testing sets with 638 entries.

```
def k_fold_CV(k, dataX, dataY, epochs, batch_size):
    interval_length = len(dataX) // k
    k_accuracy = []
    k_accuracy_test = []
    k_loss = []
    k_loss_test = []
    time_log = []
    model = get_model()
    for i in range(k):
        print("fold number: ",i)
        start_time = time.time()

        # get train and test data
        train_X = np.concatenate([dataX[:i * interval_length],dataX[(i + 1) * interval_length:]],axis=0)
        train_Y = np.concatenate([dataY[:i * interval_length],dataY[(i + 1) * interval_length:]],axis=0)

        test_X = dataX[i * interval_length: (i + 1) * interval_length]
        test_Y = dataY[i * interval_length: (i + 1) * interval_length]

        history, timeTaken = fit_model(train_X, train_Y, test_X, test_Y, epochs, batch_size, model)

        time_log.append(timeTaken)
        k_accuracy.append(history.history['accuracy'])
        k_accuracy_test.append(history.history['val_accuracy'])
        k_loss.append(history.history['loss'])
        k_loss_test.append(history.history['val_loss'])

    avg_time=np.mean(time_log)
    return k_accuracy, k_accuracy_test, k_loss, k_loss_test, avg_time
```

Figure 3: K-fold implementation

Figure 3 shows the implementation of k-fold cross validation for sub questions 2 to 5. The reason for this procedure is to partition the data into k subsets, where each subset is used as the testing set for the model, the average of the k results will be used as the final result. This method is effective in preventing overfitting since all partitions of the original dataset will be used for testing.

```
def get_model():
    model = keras.Sequential([
        keras.layers.Dense(hidden_layer, use_bias = True, input_shape = (NUM_FEATURES,), activation='relu',
                           kernel_regularizer = tf.keras.regularizers.l2(decay_beta)),
        keras.layers.Dense(NUM_CLASSES, use_bias=True, activation='softmax')
    ])

    GD_opt = keras.optimizers.SGD(learning_rate=learning_rate)

    model.compile(optimizer=GD_opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Figure 4: keras model generation

In this problem, the Sequential function from tensorflow.keras will be used to implement the models. I will be using the SGD (stochastic gradient descent) function from keras.optimizers with the learning rate specified in the questions as the optimizer, which is then passed into the compile(). I will also be using categorial cross entropy as the loss function for the models.

```
def fit_model(train_X, train_Y, test_X, test_Y, epochs, batch_size, model):
    timeTaken=timeCallback()
    history = model.fit(train_X, train_Y,
                        epochs=epochs,
                        callbacks = [timeTaken],
                        verbose = 2,
                        batch_size=batch_size,
                        validation_data = (test_X, test_Y),
                        shuffle=True)
    timeTaken = np.mean(timeTaken.logs)
class timeCallback(tf.keras.callbacks.Callback):

    def on_train_begin(self, logs={}):
        self.logs=[]

    def on_epoch_begin(self, epoch, logs={}):
        self.epoch_start = time.time()

    def on_epoch_end(self, epoch, logs={}):
        self.logs.append(time.time() - self.epoch_start)
```

Figure 5: model fitting and epoch timing

In order to train the model, I will be using the fit() function from keras, which slices the data into partitions based on the 'batch_size' parameter which repeatedly iterates over the given training set for the given number of epochs. The model is validated using the testing sets and can be evaluated based on its accuracy and loss for both the training set and testing set. In order to keep track of the time taken per epoch, the callback parameter is used, which begins timing at the beginning of each epoch and saves its duration into a log which is return at the end of execution.

Experiments and Results

Question 1:

Design a feedforward neural network which consists of:

- 1 hidden layer with 10 neurons, using ReLU activation function
- An output layer using softmax
- Learning rate = 0.01
- L2 regularization with weight decay = 10^-6

- Batch size = 32

The hyperparameters used for this question is as follows:

```
NUM_CLASSES = 3
NUM_FEATURES = 21

epochs = 1000
batch_size = 32
num_neurons = 10
seed = 10
decay_beta = 0.000001
learning_rate = 0.01
hidden_layer = 10
```

Part a)

- Use the training dataset to train the model and plot accuracies on training and testing data against training epochs.

Since the range of data from the original dataset is considerably large, the data was normalized using feature scaling as shown in Figure 1, then split into 70:30 portions for training and testing datasets respectively. Then, the model was generated and trained using the parameters specified above. The code used for this section is shown below:

```
model = keras.Sequential([
    keras.layers.Dense(hidden_layer, use_bias = True, input_shape = (NUM_FEATURES,), activation='relu',
                       kernel_regularizer = tf.keras.regularizers.l2(decay_beta)),
    keras.layers.Dense(NUM_CLASSES, use_bias=True, activation='softmax')
])

GD_opt = keras.optimizers.SGD(learning_rate=learning_rate)

model.compile(optimizer=GD_opt,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_X, train_Y,
                    epochs=epochs,
                    verbose = 2,
                    batch_size=batch_size,
                    validation_data = (test_X, test_Y),
                    shuffle=True)
```

The final loss and accuracy after 1000 epochs are:

```
loss, accuracy = model.evaluate(test_X, test_Y)
```

```
20/20 [==============================] - 0s 450us/step - loss: 0.2391 - accuracy: 0.9028
```

The accuracy against the number of epochs for the training and testing data can be observed from the graph below:

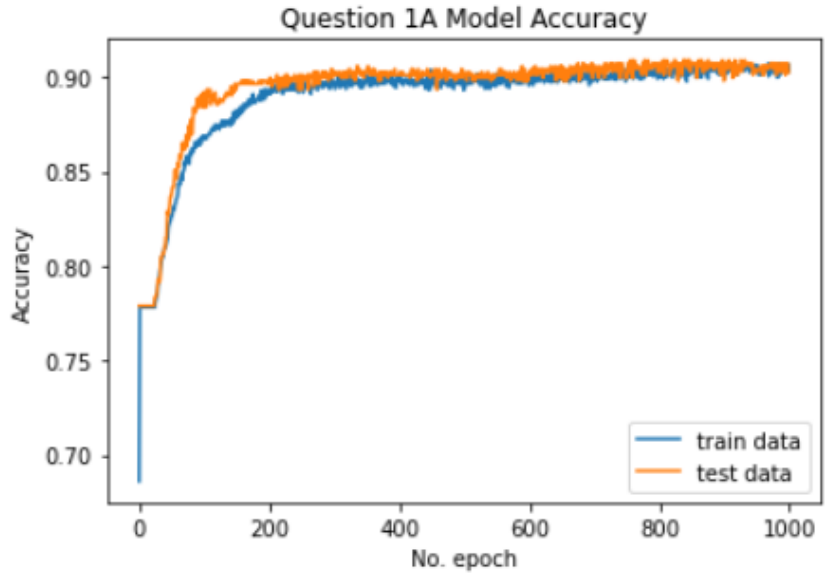

Figure 6: Question 1A accuracy

Part b)

- State the approximate number of epochs where the test error begins to converge.

In order to answer this question, the losses of the training and testing sets were also graphed:

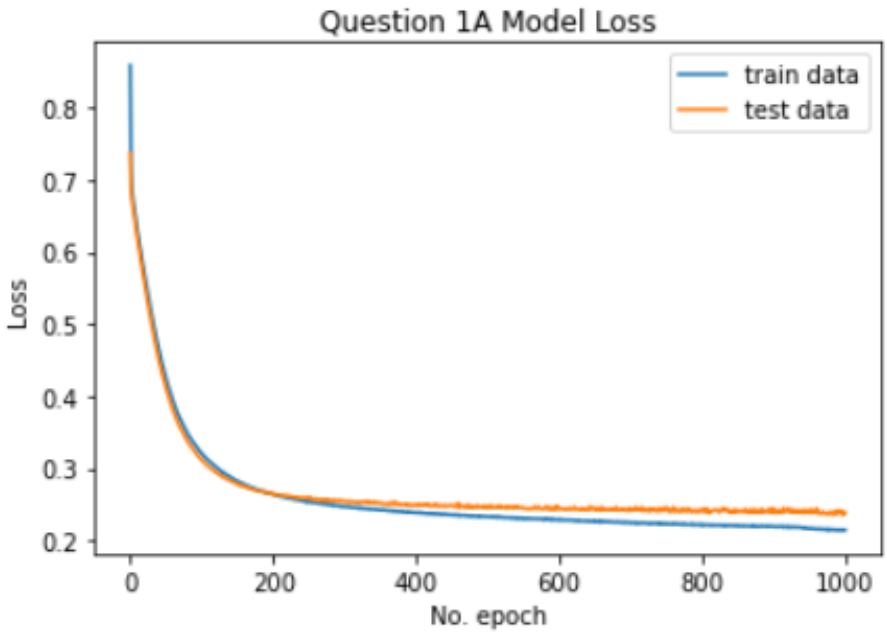

Figure 7: Question 1A loss

As can be observed from Figure 7, the loss of both training and testing sets begins to converge at around 250 to 300 epochs. Thus, 300 epochs will be used as the default value for the rest of the questions.

Question 2:

Find the optimal batch size by training the neural network and evaluating the performances for different batch sizes.

Part A)

- Plot cross-validation accuracies against the number of epochs for different batch sizes. Limit search space to batch sizes {4,8,16,32,64}. Plot the time taken to train the network for one epoch against different batch sizes.

Instead of splitting the data using train_test_split() function, the k-fold cross validation method is used using the k_fold_CV() function in Figure 3 to avoid over-fitting. This method will be used for the rest of the questions in part A.

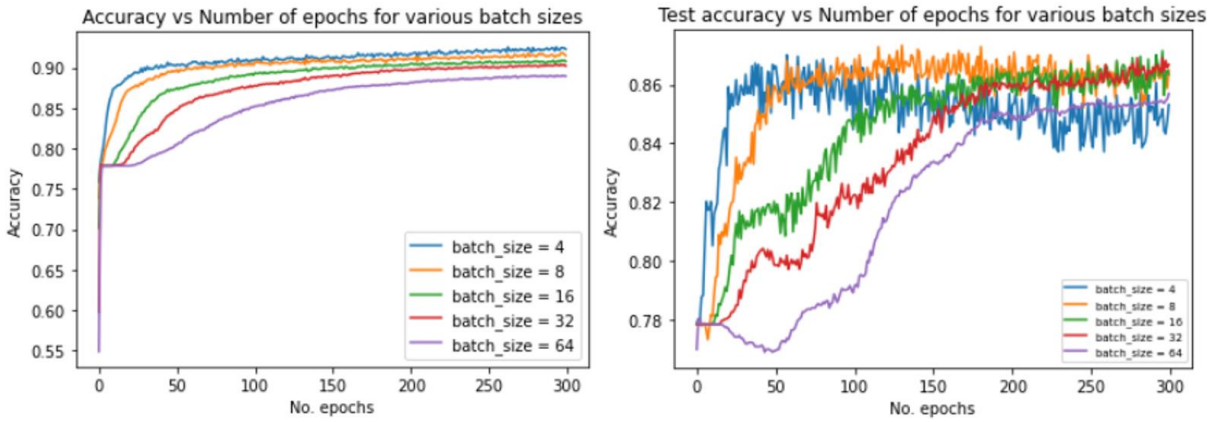

Figure 8: Question 2A Train and test accuracy vs epochs for various batch sizes

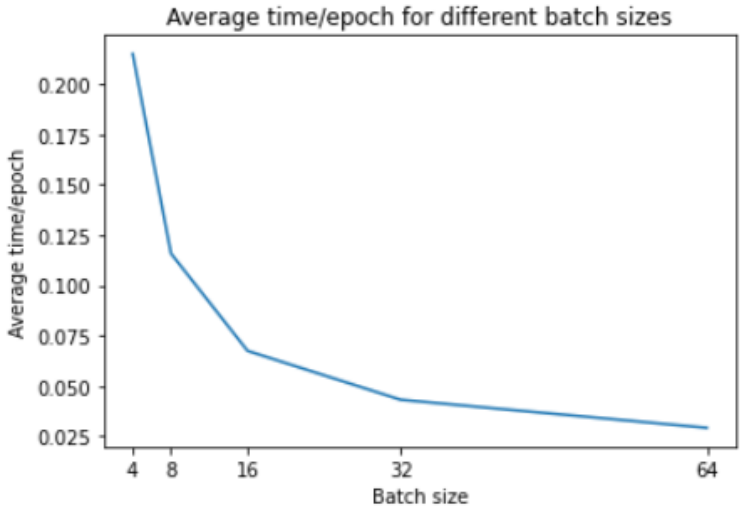

Figure 9: Question 2A time/epoch for various batch sizes

Part B)

- Select the optimal batch size and state reasons for your selection.

As can be seen from Figure 8, the accuracy achieved with batch size of 4 was significantly better than the rest of the batch sizes. Although this is true, it can be observed that the accuracy of the test set for batch size 4 peaks at around 50 epochs, then gradually starts to decline. This may be due to the noise added during the learning process and/or the large learning rate specified in the question. Furthermore, the average time taken per epoch is also significantly higher than that of other batch sizes. The testing accuracies for batch size of 8, 16, and 32 are relatively similar, but the average time per epoch for batch size of 32 is significantly less than that for batch size of 8 and 16. Thus, I chose to use batch size of 32 for the rest of the question.

Part C)

Plot the train and test accuracies against epochs for the optimal batch size. Note: use this optimal batch size for the rest of the experiments.

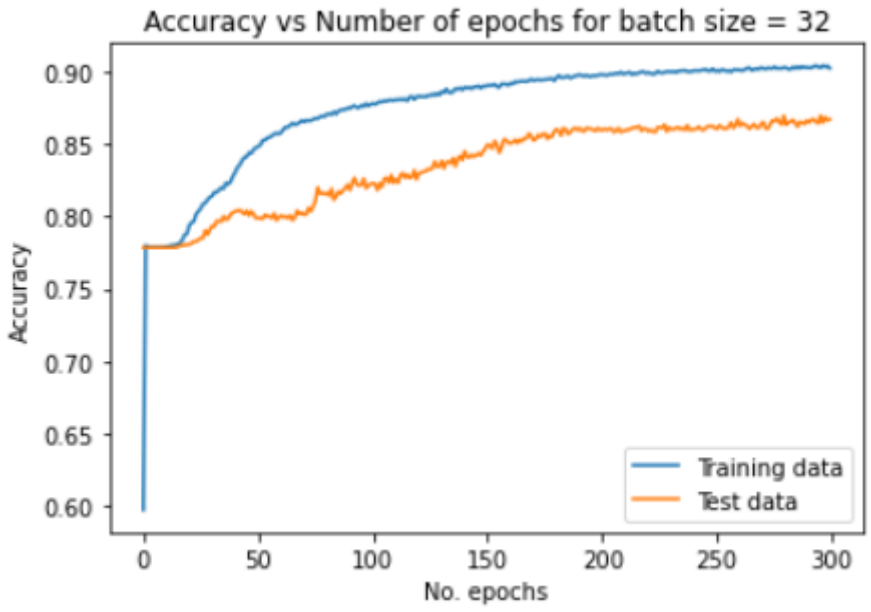

Figure 10: Question 2C Accuracy of raining and testing data for optimal batch size

Question 3:

Find the optimal number of hidden neurons for the 3-layer network designed in part (2)

Part A)

- Plot the cross-validation accuracies against the number of epochs for different number of hidden-layer neurons. Limit the search space of number of neurons to {5,10,15,20,25}.

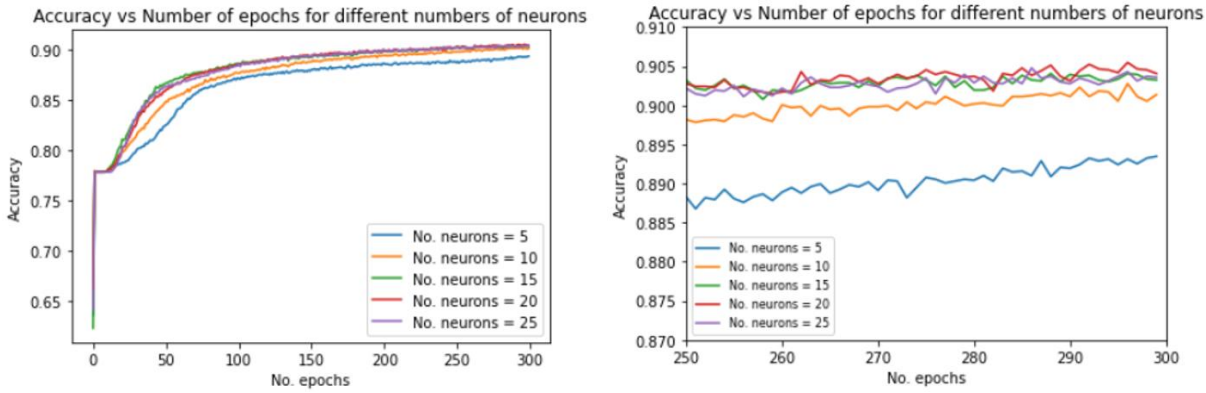

Figure 11: Question 3A Accuracy vs epochs for various numbers of neurons

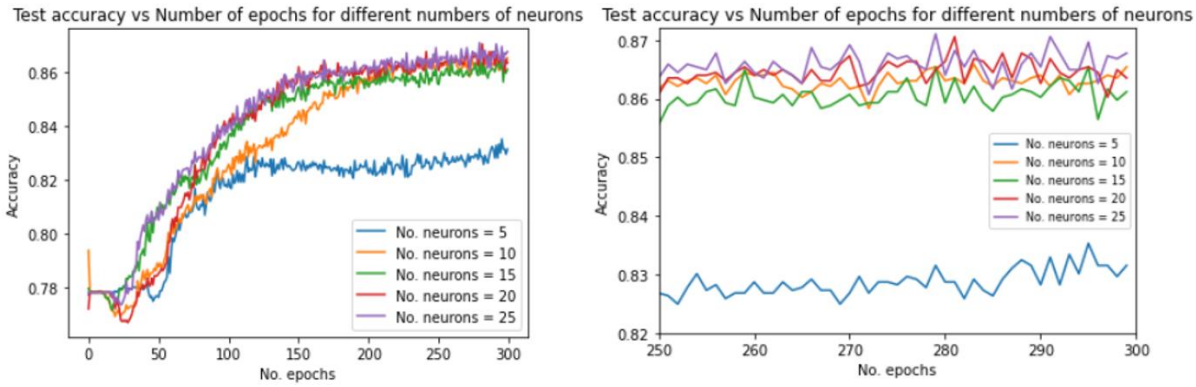

Figure 12: Question 3A Test accuracy vs epochs for various numbers of neurons

Part B)

- Select the optimal number of neurons for the hidden layer. State the rationale for your selection.

As can be observed in Figures 11 and 12, changing the number of neurons in the hidden layer has a significant effect on the accuracy of the model. However, the effects of increasing number of neurons in order to increase model accuracy seem to drop exponentially. This can be seen from both Figures 11 and 12, where the difference in accuracy for the models with 5 and 10 neurons is considerable larger than the difference

in accuracy for the models with 10 and 15 neurons. Furthermore, there was not much observable difference between the performance of the model with 20 neurons compared to that of the model with 25 neurons. This holds true for both the training and testing data. Thus, I have chosen to use 20 neurons as the optimal number of neurons for each hidden layer for the rest of this question.

Part C)

Plot the train and test accuracies against epochs with the optimal number of neurons.

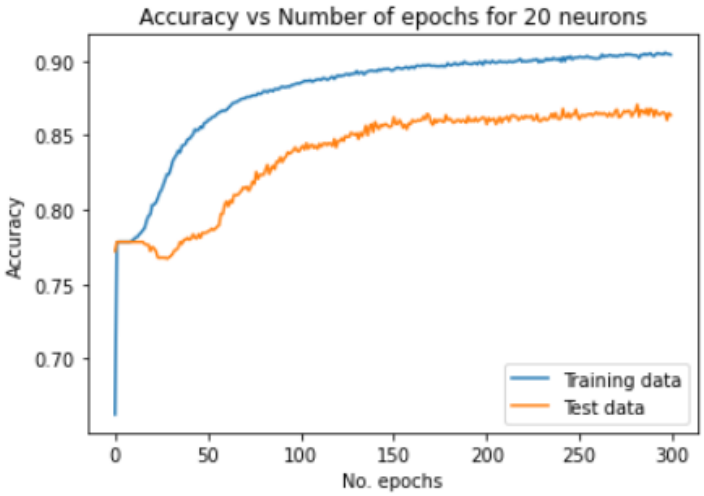

Figure 13: Question 3C Train and test accuracies for optimal number of neurons (20)

Question 4:

Find the optimal decay parameter for the 3-layer network designed with optimal hidden neurons in part (3).

Part A)

- Plot cross-validation accuracies against the number of epochs for the 3-layer network for different values of decay parameters. Limit the search space of decay parameters to {0, 10^-3, 10^-6, 10^-9, 10^-12}.

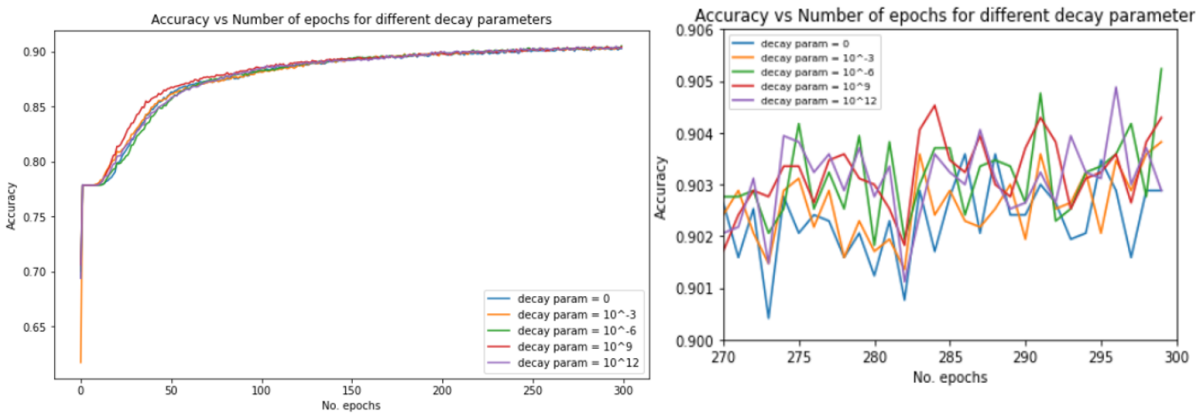

Figure 14: Question 4A Accuracy vs epochs for different decay parameters

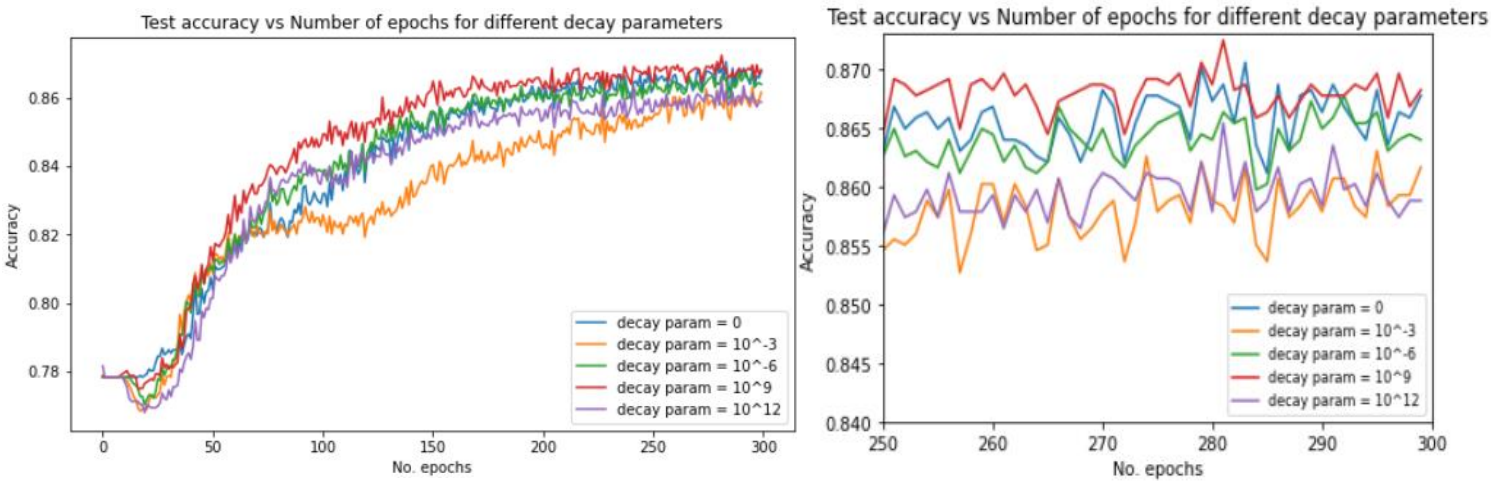

Figure 15: Question 4A Test accuracy vs epochs for different decay parameters

Part B)

- Select the optimal decay parameter. State the rationale for your selection.

As seen in Figures 14 and 15, there was not a substantial correlation between decay parameters and the accuracy of the models. For example, the performance of the model with decay parameter of 0 was better than that of the model with decay parameter 10^-3 but worse than of the model with decay parameter 10^-9 in terms of test accuracy. While this is true, it can also be observed that the model with decay parameter 10^-9 was able to converge at a slightly faster rate than the rest of the models for both the training and testing data. Thus, I have chosen 10^-9 to be the optimal decay parameter.

Part C)

- Plot the train and test accuracies against epochs for the optimal decay parameter.

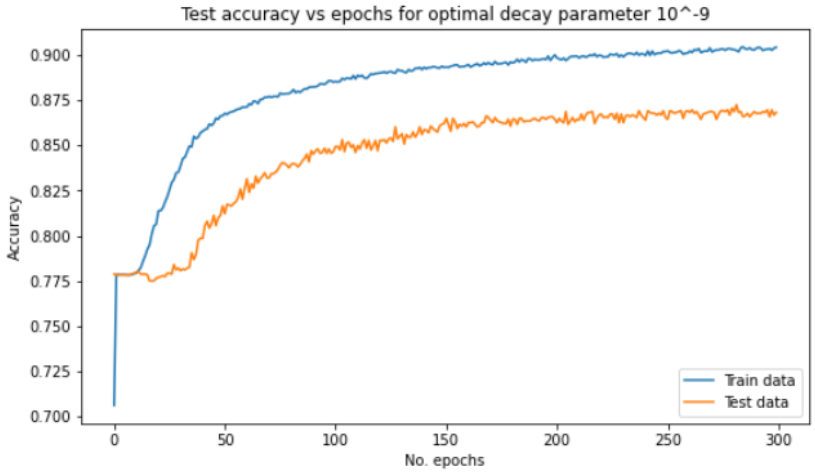

Figure 16: Question 4C Train and test accuracy vs epochs for optimal decay parameter

Question 5:

After you are done with the 3-layer network, design a 4-layer network with two hidden layers, each consisting 10 neurons, and train it with a batch size of 32 and decay parameter 10^-6.

Part A)

- Plot the train and test accuracy of the 4-layer network

The following parameters are used for this question:

```
NUM_CLASSES = 3
NUM_FEATURES = 21

epochs = 300
batch_size = 32
num_neurons = 10
decay_beta = 10**-6
learning_rate = 10**-3
hidden_layer = 10

seed = 10
```

For implementing 4-layer networks, another layer was added with identical properties to the hidden layer used in previous questions:

```
if num_hidden_layers == 1:
    model = keras.Sequential([
        keras.layers.Dense(hidden_layer, use_bias = True, input_shape = (NUM_FEATURES,), activation='relu',
                           kernel_regularizer = tf.keras.regularizers.l2(decay_beta)),
        keras.layers.Dense(NUM_CLASSES, use_bias=True, activation='softmax')
    ])
elif num_hidden_layers == 2:
    model = keras.Sequential([
        keras.layers.Dense(hidden_layer, use_bias = True, input_shape = (NUM_FEATURES,), activation='relu',
                           kernel_regularizer = tf.keras.regularizers.l2(decay_beta)),
        keras.layers.Dense(hidden_layer, use_bias = True, input_shape = (NUM_FEATURES,), activation='relu',
                           kernel_regularizer = tf.keras.regularizers.l2(decay_beta)),
        keras.layers.Dense(NUM_CLASSES, use_bias=True, activation='softmax')
    ])
```

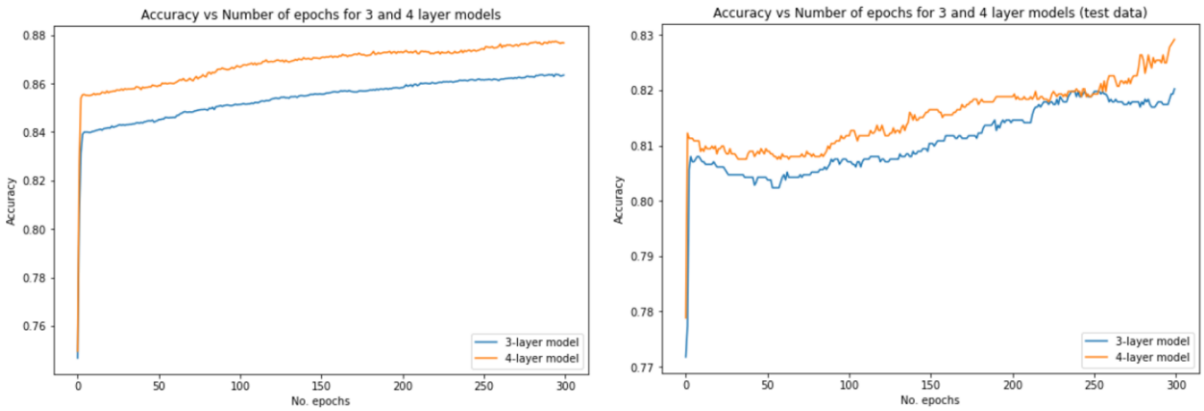

Figure 17: Train and test accuracy vs epochs for 3- and 4-layer models

Part B)

- Compare and comment on the performances of the optimal 3-layer and 4-layer networks.

From the results in Figure 17, the following observations about 3-layer and 4-layer models can be made:

- The rate at which the 3-layer and 4-layer models learn was relatively similar.
- The trajectory of accuracy for both the training and testing data was also similar for both models.
- Although the second hidden layer in the 4-layer model is identical to its first hidden layer, it constantly out-performed the 3-layer model in terms of accuracy for both the training and testing data.
    - This suggests that the original dataset is large enough to perform better with a more complex model.

Conclusion

From questions 1 through 5, an optimal set of hyperparameters for this Cardiotocography dataset was found:

Batch size	32
Number of hidden neurons	20

Decay parameter	10^-9
Number of layers	4

Table 1: Optimal hyperparameters

These parameters were found based on the accuracy of the training and testing set. However, these parameters were gathering after only running the above models a handful of times. Since model training is subject to randomness, the results across different iterations may vary. Thus, there is no absolute guarantee that the above parameters were indeed the most optimal ones.

Among these parameters, the number of hidden neurons and number of layers seems to have the largest effects on the model accuracy. Since only 3-layer and 4-layer networks were compared in question 5, the effects of increasing the hidden layer count further was not observed. However, it is known that increasing the model complexity beyond its optimal point for a given dataset may result in overfitting as thus hinder its performance.