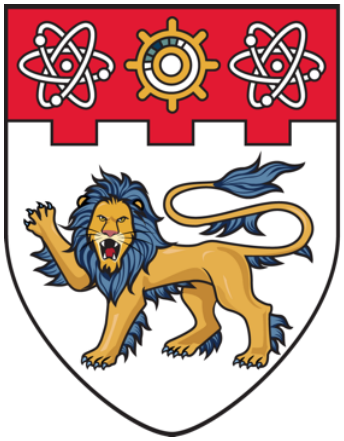


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Project 1:
Classification and Regression

CZ4042 PROJECT REPORT

by Hans Albert Lianto (U1620116K) and Eko Edita (U1620574A)

Nanyang Technological University, AY2019-2020 Semester 1

Project 1A: Classification Problem

Introduction

We are given a Cardiotocography dataset with fetal heart rate and uterine contraction features from https://archive.ics.uci.edu/ml/datasets/Cardiotocography. The dataset is labelled with the fetal state (N: Normal, S: Suspect, P: Pathologic).

Information regarding the dataset is given below:

ctg_data_cleaned.csv
No of entries in dataset = 2126
No of input attributes per row = 21
No of possible output classes = 3 [N, S, or P]
All 21 input attributes are floating point numbers.
Another output label/column is also present in the dataset, but it is ignored.

Visualization of a fragment *(10 tuples)* of the input features of the dataset is shown below:

	LB	AC	FM	UC	DL	DS	DP	ASTV	MSTV	ALTV	MLTV	Width	Min	Max	Nmax	Nzeros	Mode	Mean	Median	Variance	Tendency
0	120.0	0.000	0.000	0.000	0.000	0.000	0.000	73.0	0.5	43.0	2.4	64.0	62.0	126.0	2.0	0.0	120.0	137.0	121.0	73.0	1.0
1	132.0	0.006	0.000	0.006	0.003	0.000	0.000	17.0	2.1	0.0	10.4	130.0	68.0	198.0	6.0	1.0	141.0	136.0	140.0	12.0	0.0
2	133.0	0.003	0.000	0.008	0.003	0.000	0.000	16.0	2.1	0.0	13.4	130.0	68.0	198.0	5.0	1.0	141.0	135.0	138.0	13.0	0.0
3	134.0	0.003	0.000	0.008	0.003	0.000	0.000	16.0	2.4	0.0	23.0	117.0	53.0	170.0	11.0	0.0	137.0	134.0	137.0	13.0	1.0
4	132.0	0.007	0.000	0.008	0.000	0.000	0.000	16.0	2.4	0.0	19.9	117.0	53.0	170.0	9.0	0.0	137.0	136.0	138.0	11.0	1.0
5	134.0	0.001	0.000	0.010	0.009	0.000	0.002	26.0	5.9	0.0	0.0	150.0	50.0	200.0	5.0	3.0	76.0	107.0	107.0	170.0	0.0
6	134.0	0.001	0.000	0.013	0.008	0.000	0.003	29.0	6.3	0.0	0.0	150.0	50.0	200.0	6.0	3.0	71.0	107.0	106.0	215.0	0.0
7	122.0	0.000	0.000	0.000	0.000	0.000	0.000	83.0	0.5	6.0	15.6	68.0	62.0	130.0	0.0	0.0	122.0	122.0	123.0	3.0	1.0
8	122.0	0.000	0.000	0.002	0.000	0.000	0.000	84.0	0.5	5.0	13.6	68.0	62.0	130.0	0.0	0.0	122.0	122.0	123.0	3.0	1.0
9	122.0	0.000	0.000	0.003	0.000	0.000	0.000	86.0	0.3	6.0	10.6	68.0	62.0	130.0	1.0	0.0	122.0	122.0	123.0	1.0	1.0

In this part-project, a multi-layer neural network will be generated to classify a fetus's fetal state based on the 21 input measurements/figures of the fetus. We will explore what model network architecture would give the best test model accuracy, as well as optimizing and tuning the hyperparameters used in training the model.

Methods

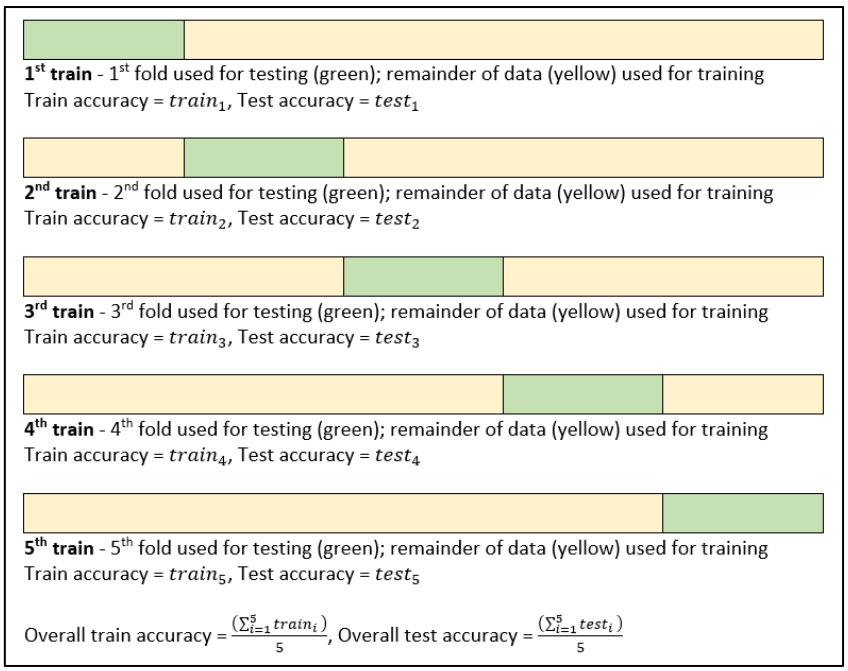
Train and Test Split

In training and testing the neural network model, the dataset is split to a (roughly) 70:30 ratio; 70% of the data is for training the model parameters, while 30% of the data is for testing the final model accuracy. In this part-project, the split is not exactly 70:30, but a split is picked such that **(i) it is closest to 70:30** and **(ii) the resulting split produces training data that can be evenly divided to 5** when 5-fold cross-validation is later applied to optimize the model hyperparameters.

5-fold cross-validation

To choose the optimal hyperparameters for the model, **5-fold cross-validation** is applied to the 70% of data used for training. That is, the training data is further split into **5 folds**, and the network is trained 5 times; at the n^{th} training, the n^{th} fold is used for testing the model and the remaining training data is used to train the model. We then get the average train/test accuracy of these 5 times of training to get a heuristic for the model accuracy. To find the optimal hyperparameter value, training under 5-fold cross-validation is done multiple times, using a different value for the hyperparameter each time, to obtain the average accuracy for that hyperparameter value. Most of the time, the hyperparameter value is then selected based on which value gives the best average test accuracy.

An illustration of training under **5-fold cross-validation** is shown below:



Normalization of features

Before the dataset is fed for training, each input feature is normalized such that they are under the same scale and 'contribute' equally to fitting the model function. Shown below is the function to scale the inputs.

```
In [3]: # Function to scale features
        def scale(X, X_min, X_max):
            return (X - X_min) / (X_max - X_min)
```

Model Architecture and Training

The model architecture and hyperparameters to optimize (in **bold**) are shown below:

3-layer model
Input neurons = 21 **No. of neurons in hidden layer** = 10 Output neurons = 3 Hidden layer is a ReLU layer. Output layer is a softmax layer. **Batch size** = 32 **L2 regularization decay parameter** = 10^{-6} No. of epochs = 5000 Learning rate (α) = 0.01

Training is performed using **mini-batch stochastic gradient descent** with L2 regularization; hence batch size and decay parameter being hyperparameters of the model. Training accuracy/error is obtained by averaging the training accuracies in every gradient update, while test accuracy is obtained by evaluating the test error at the end of each epoch. The loss is also obtained in each epoch. In one case, the gradient descent process is timed.

The model is implemented using the **TensorFlow** library in Python.

In the next section, a multitude of experiments will be performed to optimize the three hyperparameters of the 3-layer model above. Some experiments will also be performed to see if a 4-layer model would perform better than the optimized 3-layer model.

Seed initialization for predictable pseudo-randomness

The seed for initializing weights and biases for the model are always the **same**; so, every time training is run, the initial weights and biases are the same. This causes initialization to be predictable so that the only factor causing training to run differently is solely in the change of hyperparameters. In addition, the seed for shuffling the dataset for mini-batch stochastic gradient descent is also kept the same at the beginning of each training. This causes the script to return consistent results every time it is run.

Below shows the pseudorandom initialization of weights:

```
In [6]: # Create the model with 3 layers (1 hidden layer)
        def model_3_layers(num_neurons, decay_parameter, debug=False):
            x = tf.placeholder(tf.float32, [None, NUM_FEATURES])
            y_ = tf.placeholder(tf.float32, [None, NUM_CLASSES])

            # Build the graph for the deep net
            W1 = tf.Variable(tf.truncated_normal([NUM_FEATURES, num_neurons], stddev=1.0/math.sqrt(float(NUM_FEATURES)), seed=seed),
            b1 = tf.Variable(tf.zeros([num_neurons]), name='b1')
            H1 = tf.nn.relu(tf.matmul(x, W1) + b1)

            W2 = tf.Variable(tf.truncated_normal([num_neurons, NUM_CLASSES], stddev=1.0/math.sqrt(float(num_neurons)), seed=seed), na
            b2 = tf.Variable(tf.zeros([NUM_CLASSES]), name='b2')
            logits = tf.matmul(H1, W2) + b2
            Y = tf.nn.softmax(logits)
```

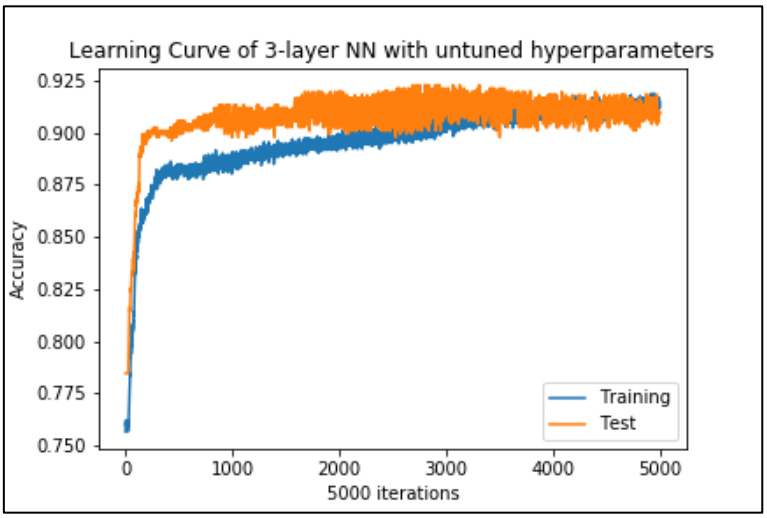
Below shows the pseudorandom initialization of shuffling the dataset during mini-batch stochastic gradient descent:

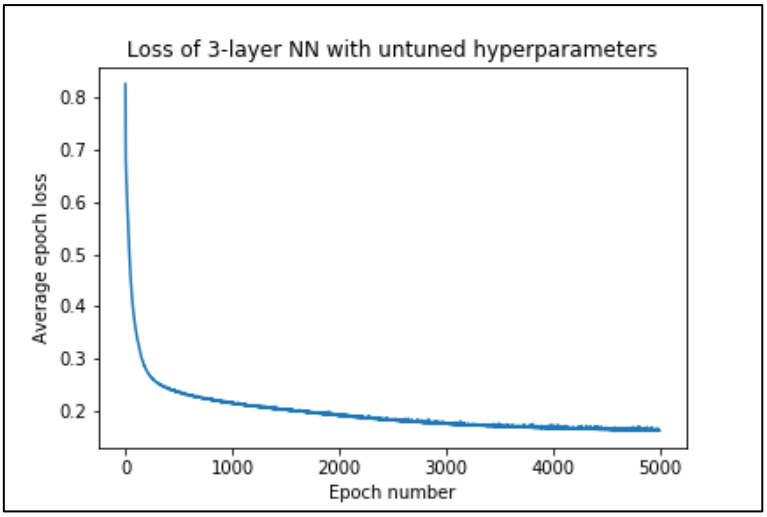
```
    # Reseed so shuffled data for mini-batch gradient descent are consistent with each training
    np.random.seed(seed)

# Initialize weights and biases in the model chosen
sess.run(tf.global_variables_initializer())

for i in range(epochs):
    if use_small_dataset:
        # Debugging
        model.run(feed_dict={x_: trainX, y_: trainY})
        train_acc.append(accuracy.eval(feed_dict={x: trainX, y_: trainY}))

        if i == 0 or i == epochs - 1:
            x_, y__, logits_, W1_, b1_, H1_, W2_, b2_, regularization_term_, Y_ = sess.run([x, y_, logits, W1, b1, H1, W
            print('iteration %d:' % (i))
            print('X: ', x_)
            print('Y: ', y__)
            print('W1: ', W1_)
            print('b1:', b1_)
            print('H1: ', H1_)
            print('W2: ', W2_)
            print('b2: ', b2_)
            print('Logits: ', logits_)
            print('Y: ', Y_)
            print('Regularization term: ', regularization_term_)

        if i % 100 == 0:
            print('iter %d: accuracy %g'%(i, train_acc[i]))
    else:
        # Shuffle the dataset before grouping them into minibatches for gradient updates
        dataset_size = trainX.shape[0]
        idx = np.arange(dataset_size)
        np.random.shuffle(idx)
        trainX, trainY = trainX[idx], trainY[idx]
```

Experiments and Results

1. Initial Training

Initial training of the 3-layer model using mini-batch stochastic gradient descent is performed using the hyperparameters illustrated above. The learning curve for the 3-layer neural network across 5000 epochs is shown below. Also shown below is the value of the loss function across 5000 epochs:



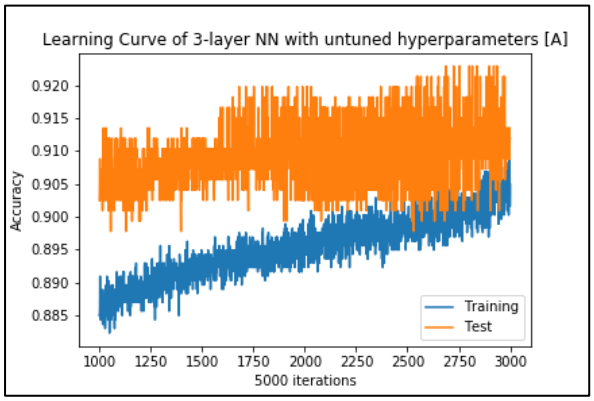


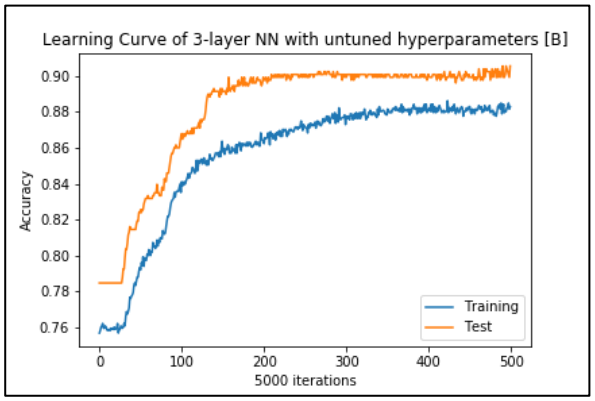
Final training accuracy	0.916
Final test accuracy	0.909
Final loss	0.162

Quite interestingly, test accuracy was higher than training accuracy at the end of most epochs. Though in the end, training accuracy was higher than test accuracy, their minute differences

indicate that the 3-layer neural network approximates the function quite accurately and does not significantly overfit or underfit the function.

After closer inspection of the model learning curve,



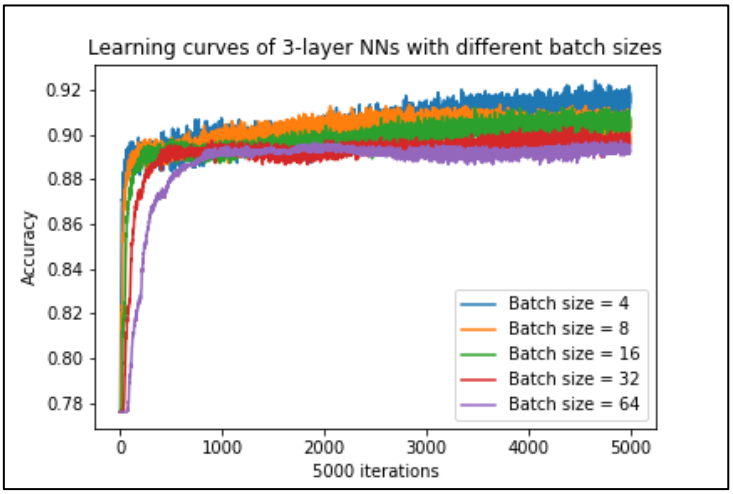


the test accuracy seems to converge at around the 0.90+ value after about **210 epochs**.

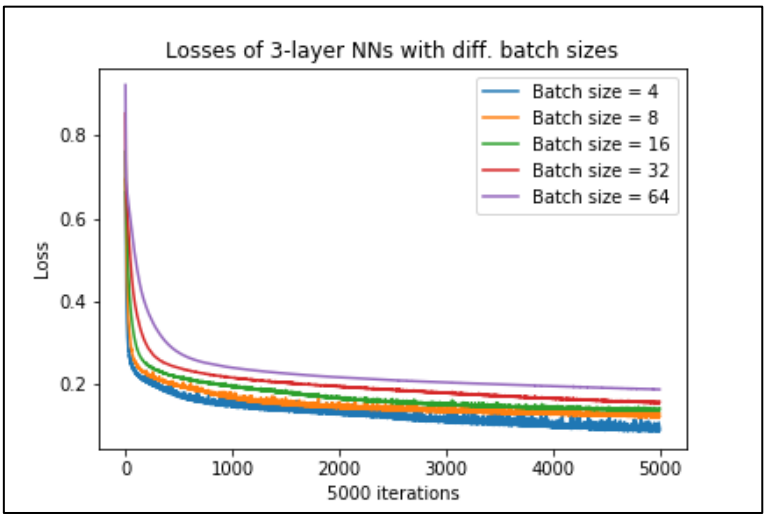
2. Tuning Batch Size

Batch size will be the first hyperparameter to tune. The five candidate values for batch size are $[4, 8, 16, 32, 64]$. In theory, making the batch size smaller will make training resemble more to **stochastic gradient descent**, considering more of the 'random noise' in the data, therefore making the final model more accurate at the expense of a very long training time. Conversely, larger batch sizes consider less of this 'noise', making the model less accurate but the training time shorter.

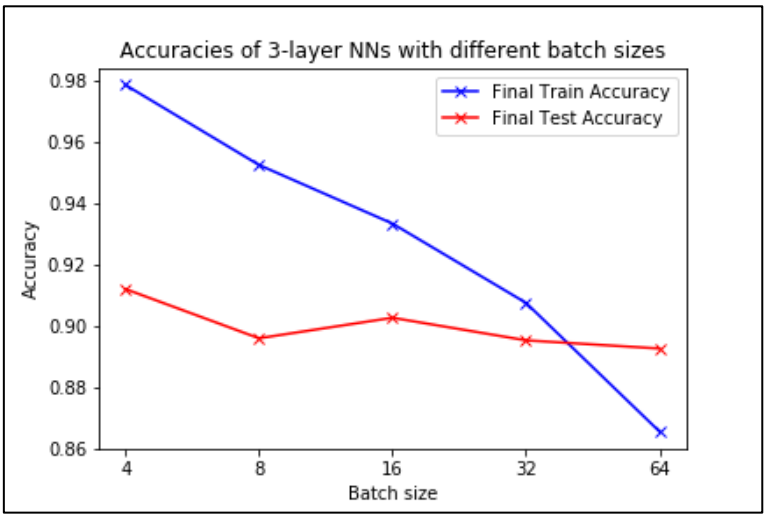
Training under 5-fold cross-validation is applied for different batch sizes, keeping other hyperparameters the same. The learning curves (test accuracy vs epoch) of each model are shown below:



The losses over time for different batch sizes are also shown below. Notice how smaller batch sizes tend to give out a smaller final value of loss, which **might** mean a more accurate model.



The final average train accuracies, test accuracies and losses of each batch size are visualized below:

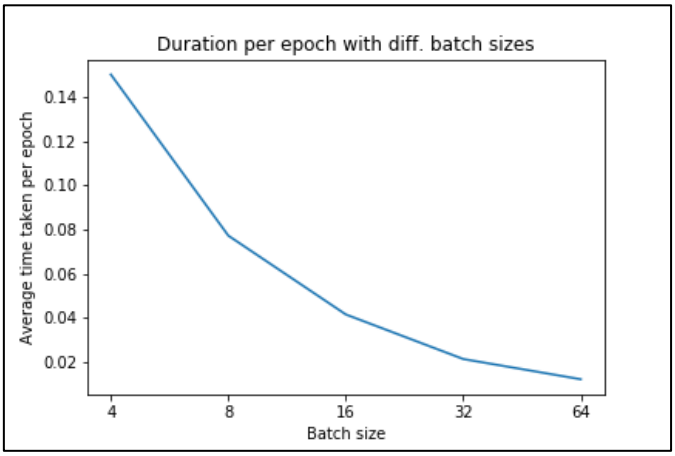


	Batch size				
	4	8	16	32	64
Final train accuracy	0.979	0.953	0.933	0.908	0.865
Final test accuracy	0.912	0.896	0.903	0.895	0.893
Final loss	0.098	0.130	0.143	0.158	0.188

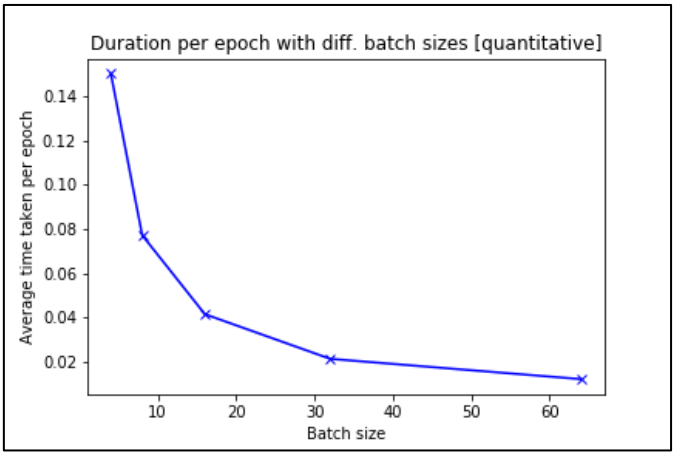
As the batch size increases, the train accuracy decreases. This makes sense as more SGD-like training will result in higher training accuracies. The same argument can be said for the final

loss increasing as batch size increases. However, test accuracy remains fairly the same at about 0.90 for different batch sizes. This may suggest that the function approximated by the model in different batch sizes are very similar after 5000 epochs and that the final parameters of the model are not far apart. After 5000 epochs, models of different batch sizes would converge to similar functions; hence giving similar test accuracies. Do note that batch size 16 gives out the highest test accuracy at 0.903.

In addition to this, each time training is performed, the process is timed and divided by the total number of epochs (5000) to get the average time per epoch of training for different batch sizes. The duration per epoch vs batch size graph is visualized below:

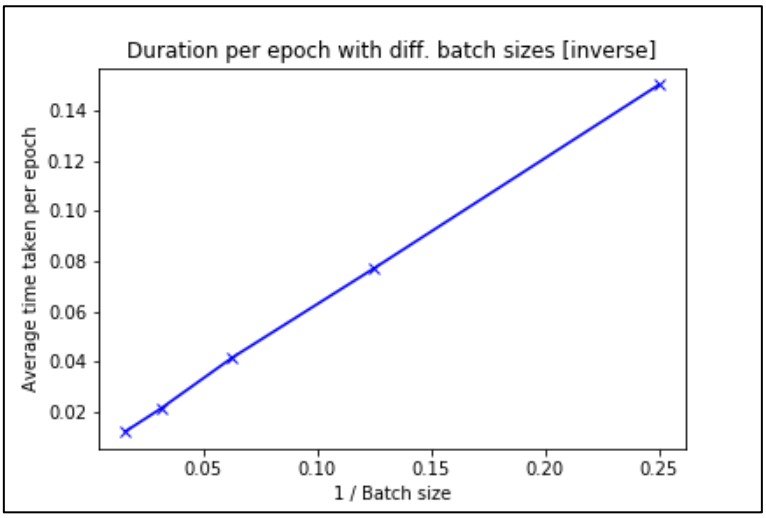


The quantitative visualization for this duration, as well as the table of duration per epoch vs different batch sizes is also shown below:



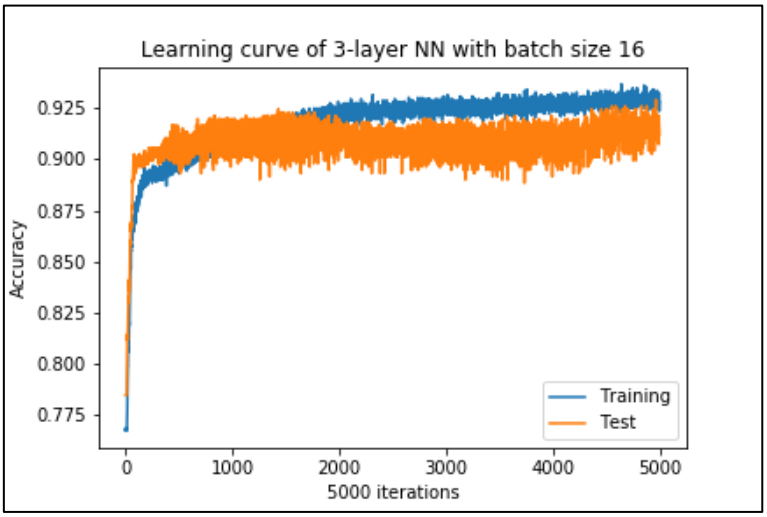
	Batch size				
	4	8	16	32	64
Duration per epoch / seconds	0.1543	0.0788	0.0405	0.0213	0.0111

Notice the inverse relationship between batch size and the duration per epoch. As the batch size doubles, the duration per epoch approximately halves. This is in line with theory. This next graph visually confirms the inverse relationship (average duration vs (batch size)$^{-1}$):



According to the data above, the test accuracy is relatively high when the batch size is 16 (0.903) compared to other batch sizes other than 4 (0.912). In addition to this, the duration per epoch for batch size 16 allows training to run for 202.5 seconds (3 minutes and 22.5 seconds), which is a relatively short and convenient time. Hence, a batch size of 16 is chosen to train the model due to its high model test accuracy and because it offers a nice compromise of time spent for training (training with batch size 4 is too long even though it provides a higher test accuracy under 5-fold cross-validation).

Training is then performed with batch size 16. Below is the learning curve of the model and comparisons between the model of batch size 16 and the initial model of batch size 32:



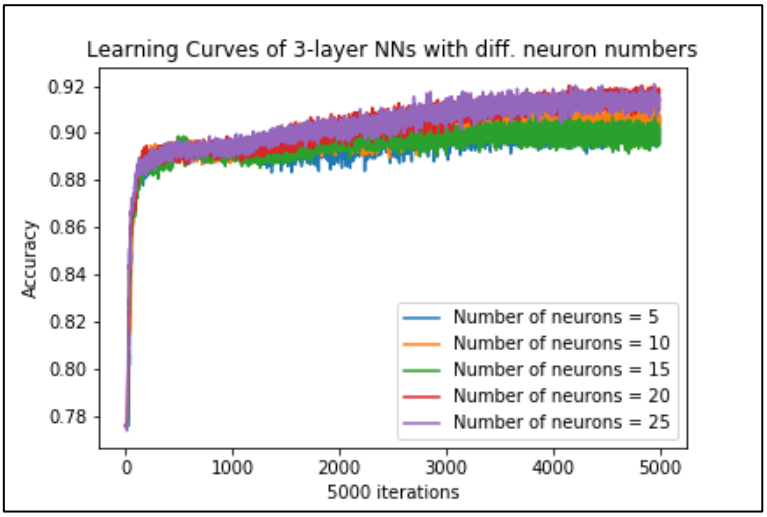
	Batch size 32	Batch size 16
Final training accuracy	0.916	0.928
Final test accuracy	0.909	0.912
Final loss	0.162	0.152

While the model trains for a shorter time, a higher test accuracy of 0.912 is obtained with batch size 16 compared to 0.909 with batch size 32. A fair compromise is also struck between test accuracy and training time.

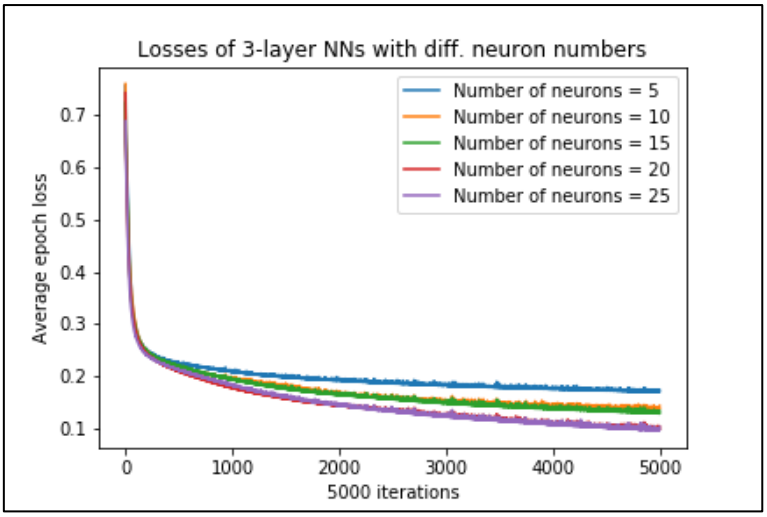
3. Tuning number of neurons in hidden layer

The number of neurons in the hidden layer will be the next hyperparameter to tune. The five candidate values for the number of neurons are $[5, 10, 15, 20, 25]$. In theory, making the number of neurons higher should increase the training accuracy of the model, but having too many hidden-layer neurons may cause the model to 'overfit' the function to classify fetal status, causing the test accuracy to drop. Here, the model with the highest test accuracy (the one that has the highest number of hidden-layer neurons without overfitting) will be chosen.

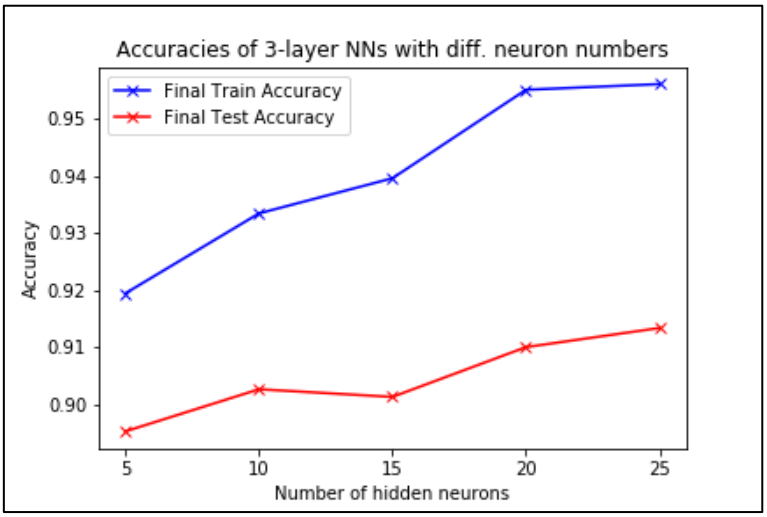
Training under 5-fold cross-validation is applied for different number of neurons, keeping other hyperparameters the same. The learning curves (test accuracy vs epoch) of each model are shown below:



The losses over time for different number of hidden-layer neurons are also shown below. Notice how, in general (25 hidden-layer neurons being the exception), more hidden-layer neurons tend to give out a smaller final loss, which **might** mean a more accurate model.



The final average train accuracies, test accuracies and losses of different number of hidden-layer neurons are visualized below:

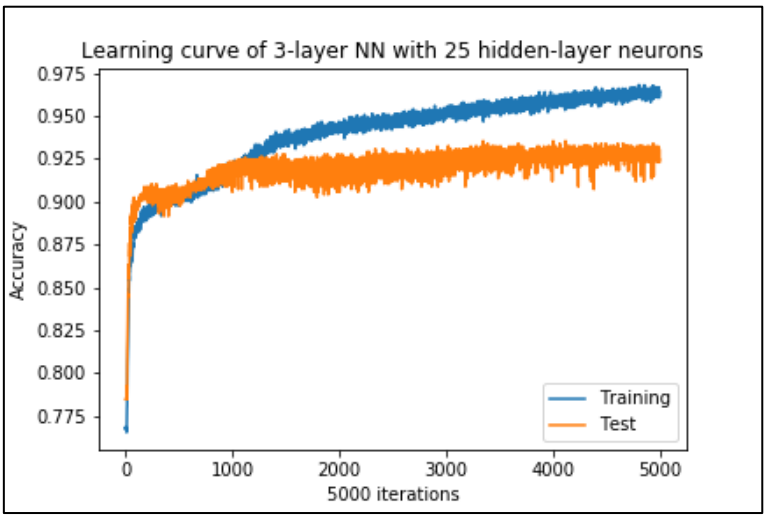


	Number of hidden-layer neurons				
	5	10	15	20	25
Final train accuracy	0.919	0.933	0.940	0.955	0.956
Final test accuracy	0.895	0.903	0.901	0.910	0.913
Final loss	0.173	0.143	0.134	0.103	0.102

As the number of hidden-layer neurons increases, the final train accuracy increases. This makes sense because more hidden-layer neurons would allow more complex functions to be approximated for the dataset, meaning it can 'fit' the dataset better. The same argument can be made for why the final loss decreases as the number of hidden-layer neurons increases.

For test accuracy, in general (except for 15 hidden-layer neurons, which is an anomaly), the final test accuracy overall increases as the number of hidden-layer neurons increases. This is because the model has not 'overfit' the data yet (it would overfit the data at some number of hidden-layer neurons above 25). Due to this, a 3-layer neural network model with **25 hidden-layer neurons** is chosen for subsequent training.

Training is then performed with batch size 16 and 25 hidden-layer neurons. On the next page is the learning curve of the model and comparisons between the model with 25 hidden-layer neurons and the previous model:



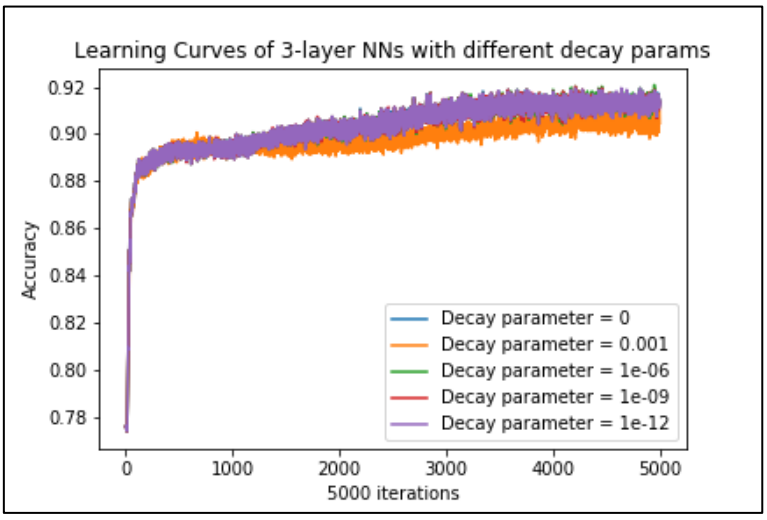
	Batch size 16 10 hidden neurons	Batch size 16 25 hidden neurons
Final training accuracy	0.928	0.964
Final test accuracy	0.912	0.925
Final loss	0.152	0.092

A significantly better model is obtained with 25 hidden-layer neurons with a higher test accuracy of 0.925 compared to the previous model's test accuracy of 0.912.

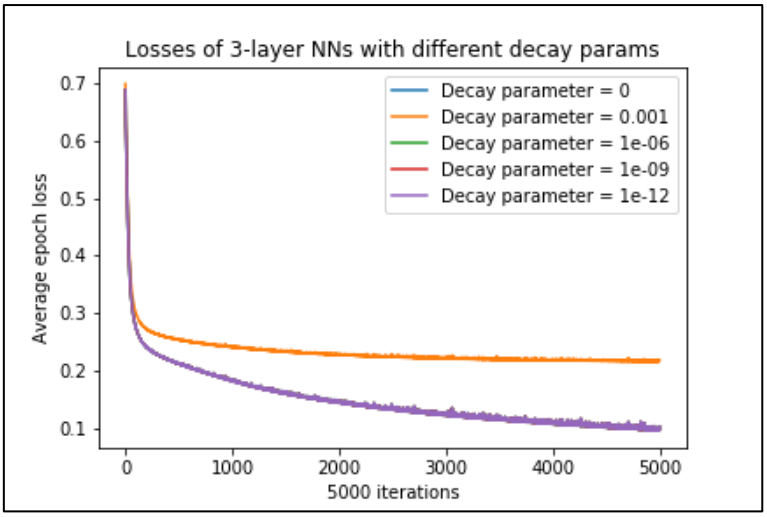
4. Tuning L2 regularization decay rate

The L2 regularization decay rate will be the last hyperparameter to tune. The five candidate values for the number of neurons are $[0, 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}]$. A decay rate of 0 means no L2 regularization, meaning that the loss function is not added with the sum of the model's weight. A higher decay rate means more regularization is applied, meaning the loss function is added with the sum of the model's weights times a factor of the decay rate. More regularization means that it is less likely for the model to overfit the data during training, albeit too much regularization could cause the training and test accuracy to drop significantly and make the model underfit the data. A good regularization balance is needed such that there is low bias and low variance.

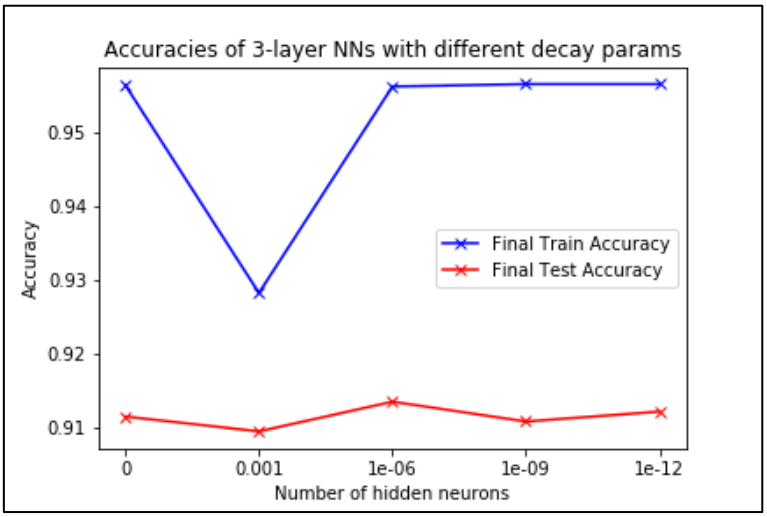
Training under 5-fold cross-validation is applied for different decay rates, keeping other hyperparameters the same. The learning curves (test accuracy vs epoch) of each model are shown below. Notice how the accuracy is almost the same for smaller decay parameters ($< 10^{-6}$). The test accuracy of the model with decay parameter 10^{-3} is significantly less accurate than that of other models; hence its orange learning curve being below the others.



The losses over time for different number of hidden-layer neurons are shown in the next page. The loss is much higher when most regularization (decay rate = 10^{-3}) is applied. This makes sense as the loss function with decay rate = 10^{-3} has the highest regularization 'term' added to the loss. While for much lower values of decay rate (exponentially), the loss is almost the same (hence the clashing learning curves / losses over time); the losses over time for decay rates 0, 10^{-6} and 10^{-9} are covered by that of 10^{-12}.



The final average train accuracies, test accuracies and losses of different decay rates are visualized below:

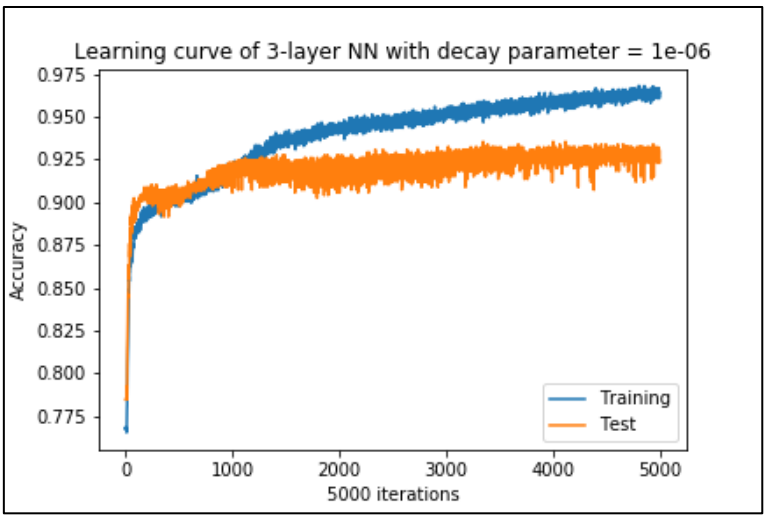


	L2 regularization decay rate				
	0	10^{-3}	10^{-6}	10^{-9}	10^{-12}
Final train accuracy	0.9563	0.9282	0.9562	0.9565	0.9565
Final test accuracy	0.9114	0.9094	0.9134	0.9107	0.9121
Final loss	0.1026	0.2181	0.1024	0.1024	0.1026

Notice how for a decay rate of 10^{-3}, train and test accuracy are far lower than that of lower decay rates. Decay rates below 10^{-6}, however do not affect the accuracy and loss by very much as it does not contribute significantly to the loss function. This is presumably why the train and test accuracies and losses for these decay rates are almost the same. However, notice that the model with decay rate 10^{-6} has the highest test accuracy of 0.9134 compared

to other models. Hence, even though it is a small gain in test accuracy, a 3-layer neural network model with **decay rate** of 10^{-6} is chosen for subsequent training. This is the same with the previous model.

Training is then performed with batch size 16, 25 hidden-layer neurons and decay rate of 10^{-6}. The results, unsurprisingly, are the same as the previous model, because the current model **is** the previous model.



	Batch size 16 25 hidden neurons Decay rate = 10^{-6}	Batch size 16 25 hidden neurons Decay rate = 10^{-6}
Final training accuracy	0.964	0.964
Final test accuracy	0.925	0.925
Final loss	0.092	0.092

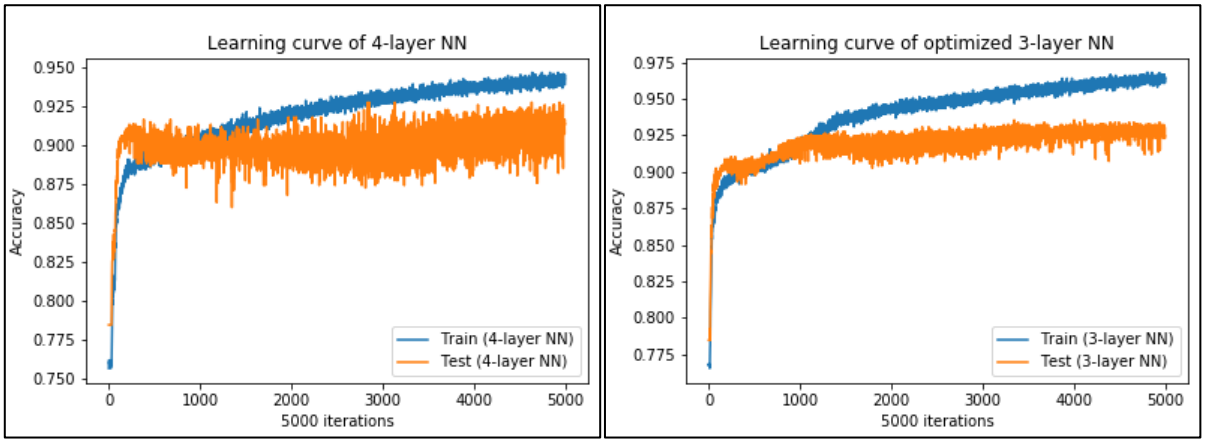
Hence an optimized 3-layer neural network is obtained with batch size 16, 25 hidden-layer neurons and a decay rate of 10^{-6}. Of course, if we could have more hidden-layer neurons, it could be possible to obtain an even better 3-layer neural network architecture with better test accuracy. This is because the model has not overfit yet in terms of the number of hidden-layer neurons used. Moreover, a better model might be obtained if we choose a decay rate between 10^{-6} and 10^{-3} that contributes just right to the model's loss function.

5. Comparing optimized 3-layer neural network and 4-layer neural network

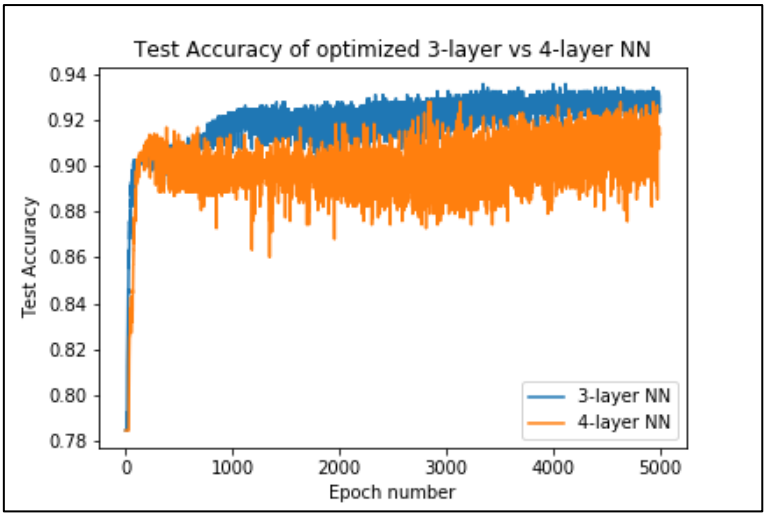
A 4-layer neural network is proposed to compete with the optimized 3-layer network. Its architecture is shown below:

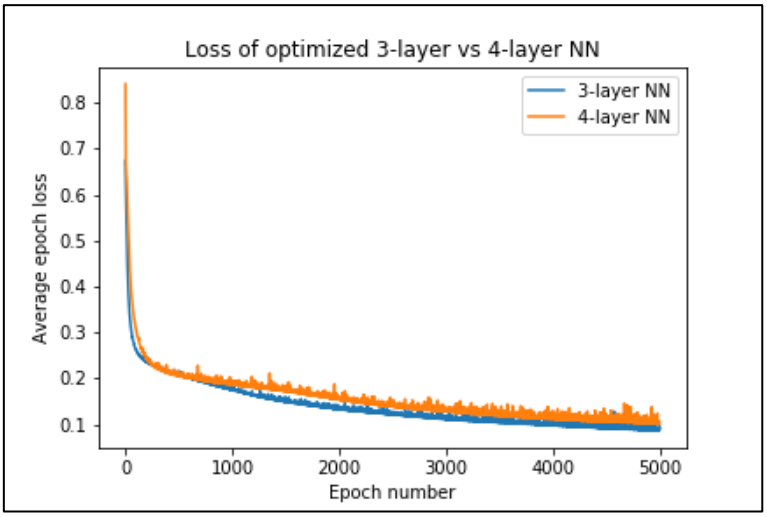
4-layer model
Input neurons = 21 No. of neurons in 1st hidden layer = 10 No. of neurons in 2nd hidden layer = 10 Output neurons = 3 Both hidden layers are ReLU layers. Output layer is a softmax layer. Batch size = 32 L2 regularization decay parameter = 10^{-6} No. of epochs = 5000 Learning rate (α) = 0.01

The 4-layer neural network model is trained using the training data and test data as a result of the dataset's 70:30 split. Here is the learning curve for the 4-layer neural network:



In the next page are the test accuracy and loss comparisons across 5000 epochs for the optimized 3-layer and 4-layer network. Notice how the final test accuracy and loss for the optimized 3-layer network is higher and lower than that of the 4-layer network, respectively.





	Optimized 3-layer neural network	4-layer neural network
Final training accuracy	0.964	0.939
Final test accuracy	0.925	0.914
Final loss	0.092	0.102

The optimized 3-layer neural network has a better train accuracy (0.964 > 0.939), test accuracy (0.925 > 0.914) and lower loss (0.092 < 0.102) than that of the 4-layer neural network. This shows that tuning hyperparameters such as batch size, decay rate and number of hidden-layer neurons could result in much better improvements than simply adding the number of layers. Yes, the unoptimized 4-layer neural network has a better test accuracy than the unoptimized 3-layer neural network, but the improvement is not as significant (0.914 > 0.909).

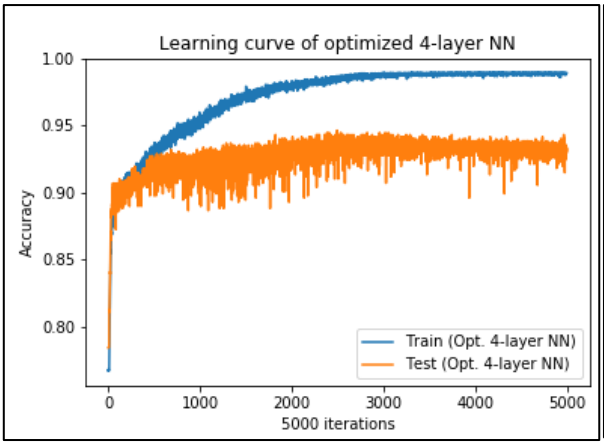
Because the dataset is simple, only having 21 inputs, it may be better to increase the number of features within those 21 inputs (higher no of hidden-layer neurons) instead of adding more abstract complex features within the features of the 21 inputs (more hidden layers).

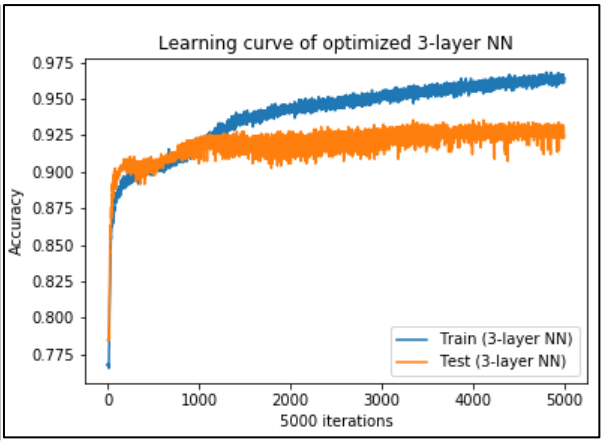
The 4-layer neural network may also perform better if the dataset was larger. Because it is inherently a more complex network than the 3-layer neural network, more data is needed to train and optimize it for test accuracy.

We will try another extra experiment. Instead of training an unoptimized 4-layer neural network, we will train an **optimized 4-layer neural network**. If the network does not overfit the data, it should have a higher accuracy than an optimized 3-layer neural network. The network architecture is shown as follows (note the change in hyperparameters):

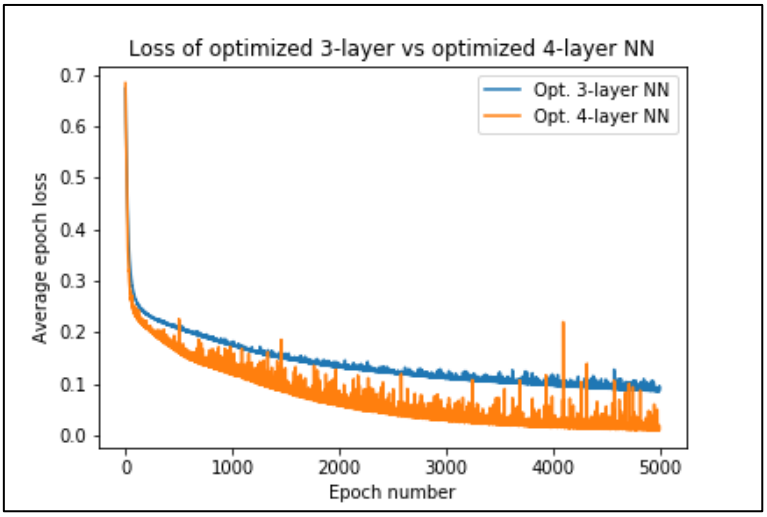
Optimized 4-layer model
Input neurons = 21 No. of neurons in 1st hidden layer = 25 No. of neurons in 2nd hidden layer = 25 Output neurons = 3 Both hidden layers are ReLU layers. Output layer is a softmax layer. Batch size = 16 L2 regularization decay parameter = 10^{-6} No. of epochs = 5000 Learning rate (α) = 0.01

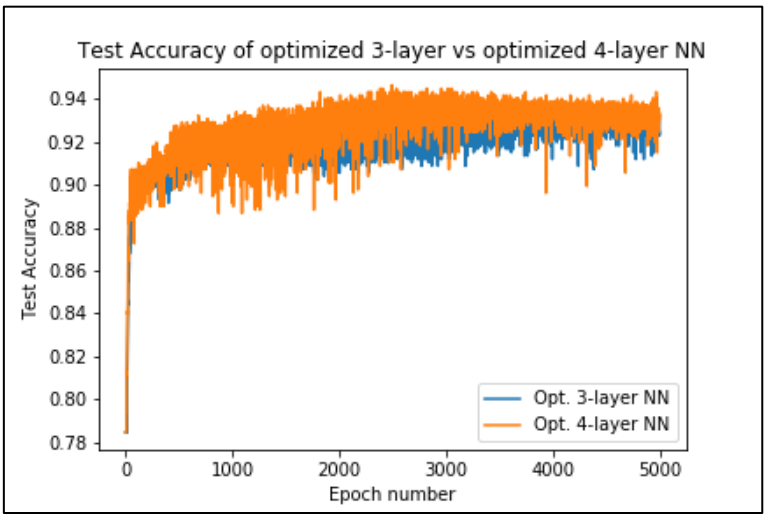
The optimized 4-layer neural network model is then trained the same way as the unoptimized 4-layer neural network. The next page shows the learning curve for the optimized 4-layer neural network.





Below are the test accuracy and loss comparisons across 5000 epochs for the optimized 3-layer and optimized 4-layer network. Notice how the final test accuracy and loss for the optimized 4-layer network is now higher and lower than that of the optimized 3-layer network, respectively.





	Optimized 3-layer neural network	Optimized 4-layer neural network
Final training accuracy	0.964	0.989
Final test accuracy	0.925	0.932
Final loss	0.092	0.017

The optimized 4-layer neural network has a better train accuracy (0.989 > 0.964), test accuracy (0.932 > 0.925) and lower loss (0.017 < 0.092) than that of the 3-layer neural network. This shows that, with the same suboptimal hyperparameters, the optimized 4-layer neural network performs better than the optimized 3-layer neural network because it can approximate more complex functions to 'fit' the data more. Of course, training the 4-layer neural network takes slightly more time due to more gradient descents being applied per epoch.

Conclusions

In this part-project, we have learned how to use 5-fold cross-validation in tuning hyperparameters to optimize a model. For batch size, a compromise between accuracy and training time needs to be struck. For regularization decay rate and the number of hidden-layer neurons, a compromise value should be chosen so that it does not overfit or underfit the data. Finally, increasing the number of hidden layers with other hyperparameters being kept the same has the potential to improve the accuracy of a model should it not overfit the training data.

One striking conclusion that is made from this experiment is the rapid fluctuation of accuracy even in later epochs of training that attributes to 'noise' in the learning curves. This may be due to the sample space for estimating a model function to classify fetal status to have lots of local minima, causing gradient descent to fluctuate very much around these local minima. One way to prevent this from happening is to decrease the learning rate of 0.01 during training. This may result in lesser noise in the learning curve, even though it would take more epochs to convergence during training.

Project 1B: Regression Problem

Introduction

We are given a dataset containing Graduate Admissions Prediction found here: https://www.kaggle.com/mohansacharya/graduate-admissions. The dataset contains 400 entries; each entry consists of numerical parameters considered for graduate admission: GRE score, TOEFL score, university rating, strength of statement of purpose, strength of recommendation letter, undergraduate GPA, and research experience. The target feature to predict is the **chance of admit** to the graduate university.

Information regarding the dataset is given below:

admission_predict.csv
No of entries in dataset = 400 **No of input attributes per row** = 7 (8 including Serial No. but it is not used) **Range of output** = [0, 1] All 7 input used attributes are floating point numbers.

The first 25 rows of the dataset are visualized below:

Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
1	337	118	4	4.5	4.5	9.65	1	0.92
2	324	107	4	4	4.5	8.87	1	0.76
3	316	104	3	3	3.5	8	1	0.72
4	322	110	3	3.5	2.5	8.67	1	0.8
5	314	103	2	2	3	8.21	0	0.65
6	330	115	5	4.5	3	9.34	1	0.9
7	321	109	3	3	4	8.2	1	0.75
8	308	101	2	3	4	7.9	0	0.68
9	302	102	1	2	1.5	8	0	0.5
10	323	108	3	3.5	3	8.6	0	0.45
11	325	106	3	3.5	4	8.4	1	0.52
12	327	111	4	4	4.5	9	1	0.84
13	328	112	4	4	4.5	9.1	1	0.78
14	307	109	3	4	3	8	1	0.62
15	311	104	3	3.5	2	8.2	1	0.61
16	314	105	3	3.5	2.5	8.3	0	0.54
17	317	107	3	4	3	8.7	0	0.66
18	319	106	3	4	3	8	1	0.65
19	318	110	3	4	3	8.8	0	0.63
20	303	102	3	3.5	3	8.5	0	0.62
21	312	107	3	3	2	7.9	1	0.64
22	325	114	4	3	2	8.4	0	0.7
23	328	116	5	5	5	9.5	1	0.94
24	334	119	5	5	4.5	9.7	1	0.95
25	336	119	5	4	3.5	9.8	1	0.97

In the second part-project, a multi-layer neural network will be generated to predict the **chance of admit** to the graduate university based on the 7 input parameters considered for graduate admission. We will compare three different networks and apply **recursive feature elimination** to one network, finding the optimal set of features that give out the least error (test loss) for the model.

Methods

Train and Test Split

In training and testing the neural network, the dataset is split to a 70:30 ratio; 70% of the data is for training model parameters, while 30% of the data is for testing the final model accuracy.

Normalization of features

Before the dataset is fed for training, each input feature is normalized such that they are under the same scale and 'contribute' equally to fitting the model function. Shown below is the function to scale the inputs. Unlike Project 1a, we normalize each feature by subtracting the feature instance value by the mean feature value, then dividing it with the feature standard deviation. Shown below is the line to normalize input features:

```
testX = (testX - np.mean(trainX, axis=0)) / np.std(trainX, axis=0)
trainX = (trainX - np.mean(trainX, axis=0)) / np.std(trainX, axis=0)
```

Model Architecture and Training

The three model architectures that will be trained are outlined below and in the next page.

3-layer model
Input neurons = 7 [will be decremented for RFE] No. of neurons in hidden layer = 10 Output neuron = 1 Hidden layer is a ReLU layer. Output neuron has a linear activation function. Batch size = 8 L2 regularization decay parameter = 10^{-3} Learning rate (α) = 0.001

4-layer model	5-layer model
Input neurons depends on RFE result. No. of neurons in 1st hidden layer = 50 No. of neurons in 2nd hidden layer = 50 Output neuron = 1 All hidden layers are ReLU layers. Output neuron has a linear activation function. Batch size = 8 Dropout keep probability = 0.8 Learning rate (α) = 0.001	Input neurons depends on RFE result. No. of neurons in 1st hidden layer = 50 No. of neurons in 2nd hidden layer = 50 No. of neurons in 3rd hidden layer = 50 Output neuron = 1 All hidden layers are ReLU layers. Output neuron has a linear activation function. Batch size = 8 Dropout keep probability = 0.8 Learning rate (α) = 0.001

For the latter two networks outlined above, **dropout** will be applied to them. This means, during training, the hidden-layer neurons of each layer have a 1 - 0.8 = 0.2 probability of being dropped, and during testing, none of these neurons are dropped (to compute test loss).

Training is performed using **mini-batch stochastic gradient descent** with L2 regularization for the first model and dropout for subsequent models. Training and test errors (i.e. losses) are obtained at the end of each epoch.

The model is implemented using the TensorFlow library in Python.

In the next section, training will be performed. **Recursive feature elimination** will be performed on the first model, and subsequent model architectures will be trained with the optimal set of features obtained by RFE.

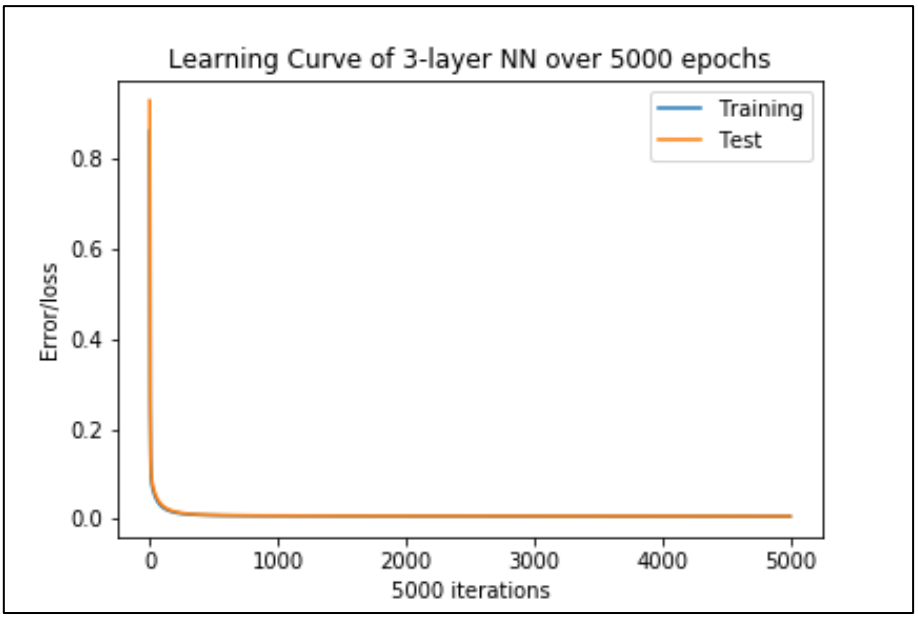
Seed initialization for predictable pseudo-randomness

The same seed initialization for predictable pseudo-randomness is applied in the same places as we did in Project 1a for the same reasons (refer to this same section but in Project 1a).

Experiments and Results

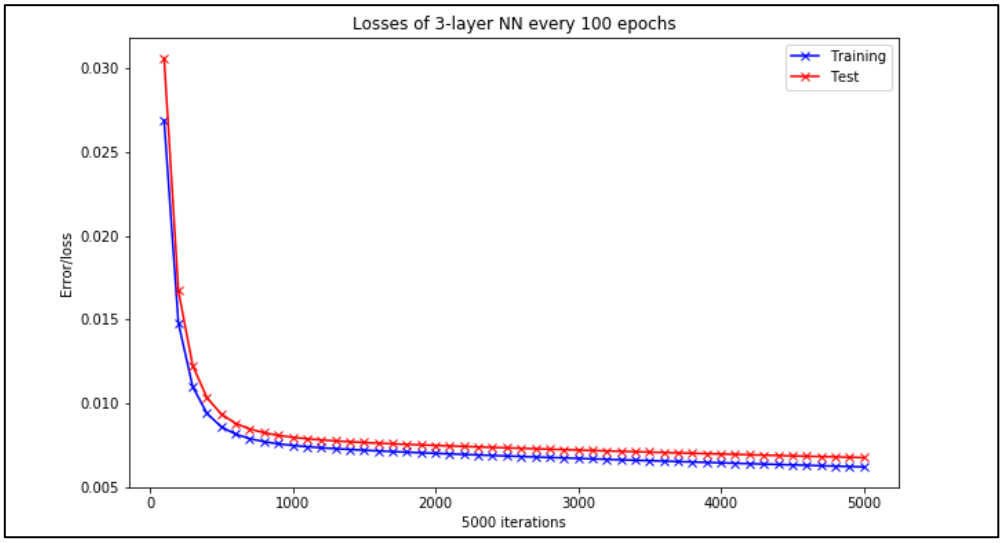
1. Initial Training

The 3-layer network outlined above will be trained with 5000 epochs. The training and test losses over 5000 epochs are shown and visualized below:



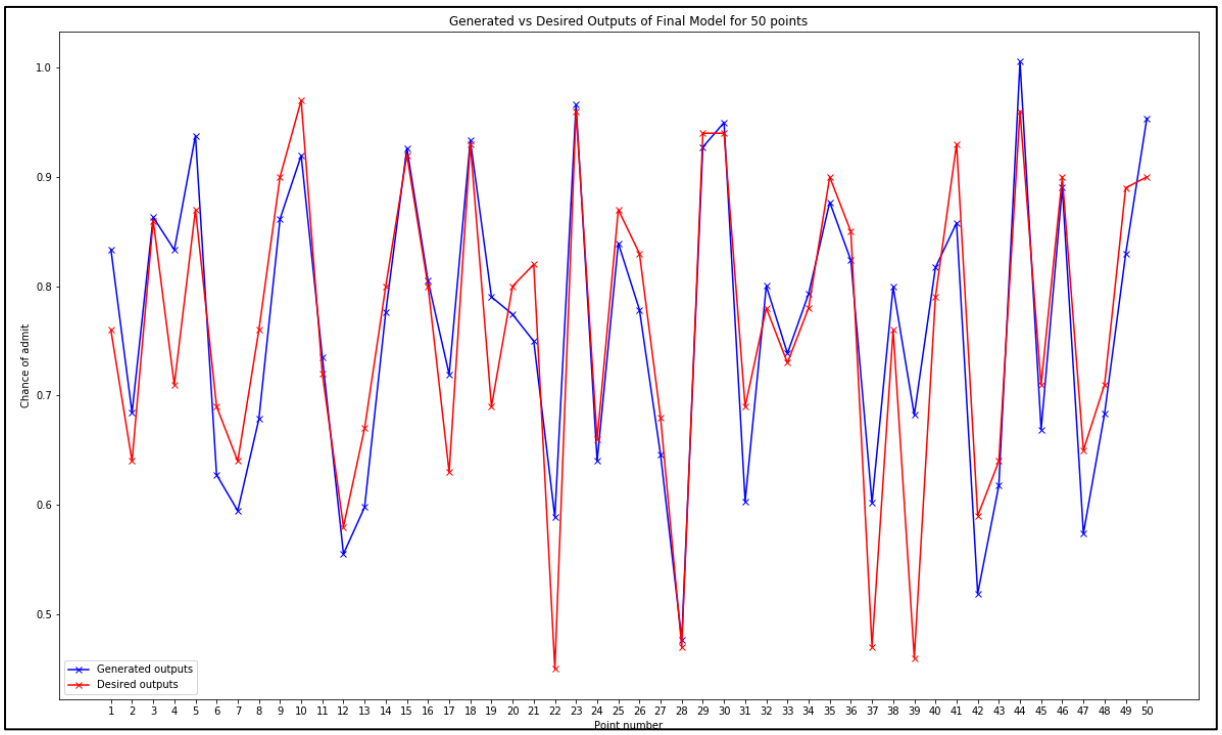
	No of epochs					
	0	1000	2000	3000	4000	5000
Training loss	0.863	0.00746	0.00699	0.00669	0.00643	0.00618
Test loss	0.931	0.00795	0.00746	0.00719	0.00695	0.00674

Notice that the loss becomes minimum very early on. We further plot the losses every 100 epochs (excluding the first loss which is very high) as such for a better read:



According to the figure above, although the test loss continues to decrease, its decrease is very little after about 1000 epochs. Because of this, training will be done with 1000 epochs from now on, as it is the approximate number of epochs whereby the test loss is considerably minimum (theoretically, the test loss is minimum at a very high number of epochs; training the model with 20000 epochs and beyond will still decrease test loss, albeit negligibly. This is presumably because the dataset does not have many rows in the first place, taking a very long time before overfitting starts to occur).

Afterwards, 50 random inputs in the test data are selected and are run through the model trained for 1000 epochs. The actual vs predicted values of the **chance of admit** are plotted and visualized below:

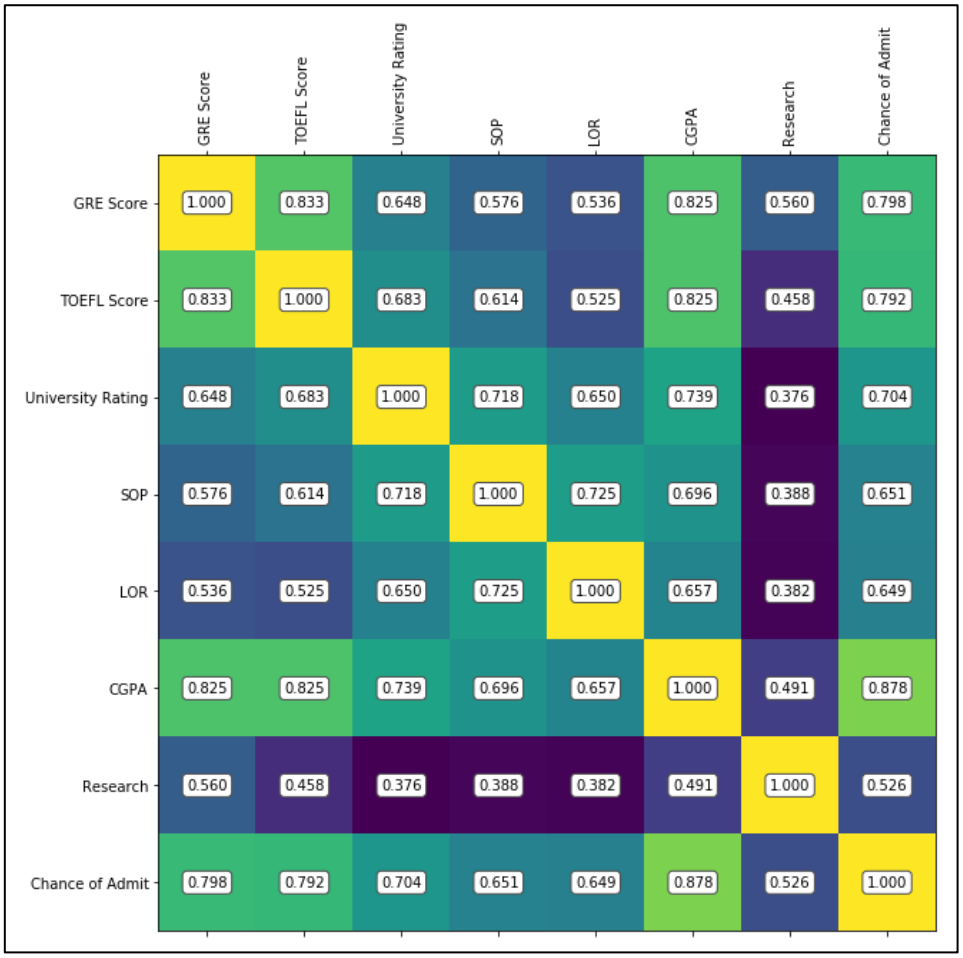


Notice that the differences between the 50 chosen values differ by slightly less than 0.1 on average, which makes sense since the **average** test loss after 100 epochs is 0.00799, which is less than 0.01 which is 0.1^2, 0.1 being the average **absolute** difference of the actual and desired values as visualized above.

2. Feature/target correlation and correlation matrix

Next, a 8x8 correlation matrix is plotted between the 7 input features of the dataset and the target feature (chance of admit). This is done using Pandas and visualized using Matplotlib.

Below is the correlation matrix:



Between input features, the feature pair with the highest correlation is **TOEFL Score** and **GRE Score**, with a correlation of 0.833. This makes sense as these figures solely depend on performance scores examinations. TOEFL and GRE are examination-based and test similar skillsets, hence such scores are highly correlated. Other feature pairs with high correlations (>0.8) are **CGPA** and **GRE Score**, as well as **CGPA** and **TOEFL Score**, both having a correlation of 0.825.

The correlation matrix overall indicates that there is a redundancy of information in input features, namely **CGPA**, **TOEFL Score**, and **GRE Score**.

The features having high correlation with the **Chance of admit** are **CPGA**, with a correlation of 0.878. Not far behind are **GRE Score** and **TOEFL Score**, with a correlation of 0.798 and 0.792 respectively.

3. Recursive feature elimination

Next, recursive feature elimination will be applied to the three-layer model to find the optimal set of features that give out the best test loss. In recursive feature elimination, features from the dataset are removed one-by-one until the test loss no longer decreases. The code snippet to implement RFE is shown as follows:

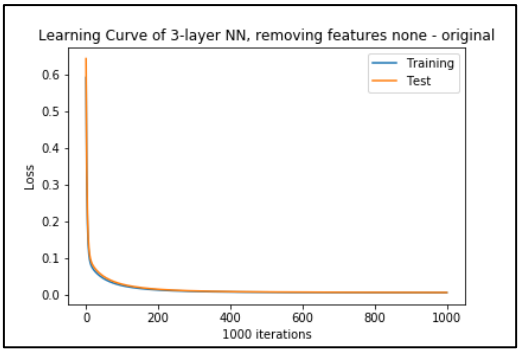
```
initial_data = original_df.to_numpy()
initial_shuffled_data = initial_data[idx]
original_trainX, original_trainY, original_testX, original_testY = split_data_to_train_and_test(initial_shuffled_data, len(features) + 1)
original_training_losses, original_test_losses = train_and_save_figures(original_trainX, original_trainY, original_testX, original_testY,
                                                                        features_removed=[], original=True)
original_best_final_test_loss = original_test_losses[-1]

def rfe(df, feature_list, prev_best_final_test_loss, removed_features=[]):
  final_test_losses = []

  for feature in feature_list:
    rfe_df = df.drop(feature, 1)
    admit_data = rfe_df.to_numpy()
    admit_shuffled_data = admit_data[idx]
    trainX, trainY, testX, testY = split_data_to_train_and_test(admit_shuffled_data, len(feature_list) + 1)

    # train model and keep the loss
    training_losses, test_losses = train_and_save_figures(trainX, trainY, testX, testY, removed_features + [feature])

    final_test_losses.append(test_losses[-1])

  best_test_loss = min(final_test_losses)
  best_feature = feature_list[final_test_losses.index(best_test_loss)]

  # If better than previous model, recurse; otherwise return the previous model
  if best_test_loss < prev_best_final_test_loss:
    removed_features.append(best_feature)
    feature_list.remove(best_feature)
    return rfe(df.drop(best_feature, 1), feature_list, best_test_loss, removed_features=removed_features)
  else:
    return prev_best_final_test_loss, removed_features
```

The steps for RFE are implemented as follows:

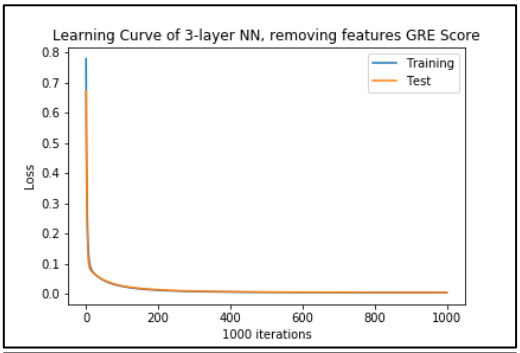
1. Train the network with no features removed. Keep the test loss.
2. Remove one feature from the input and train the network. Find the one feature removed that minimizes test loss.
3. If this test loss is lower than the test loss before the feature is removed, recurse and repeat Step 2. Otherwise, return the features removed and the better/lower test loss.

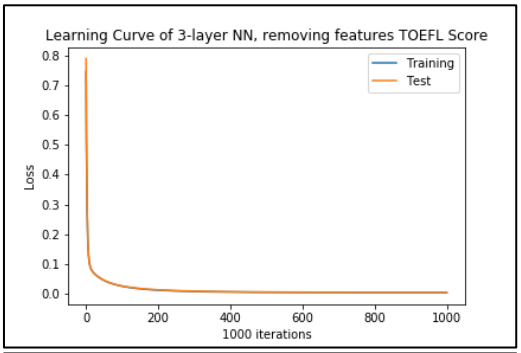
Models with 6 input features

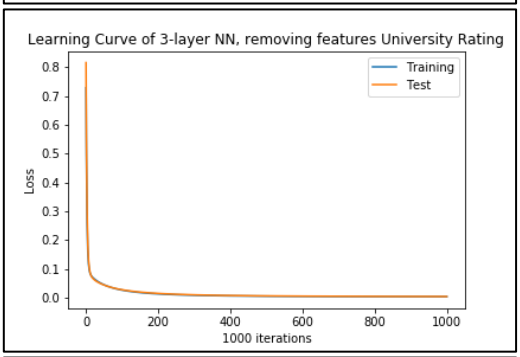
RFE is applied with one feature removed (selecting 6 input features) and then two features removed (selecting 5 input features). The results (i.e. learning curves and final training and test losses) are shown in the next pages:

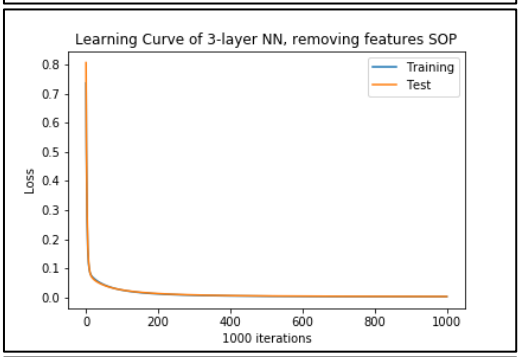


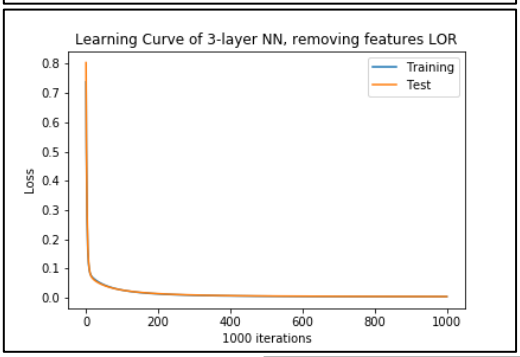
VS

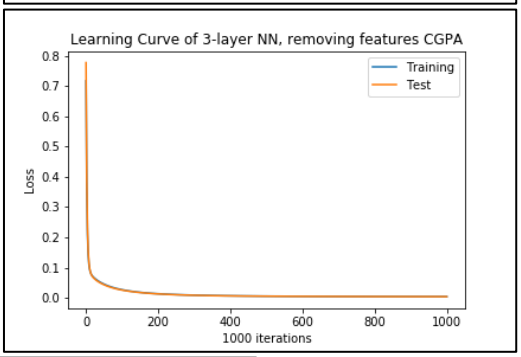


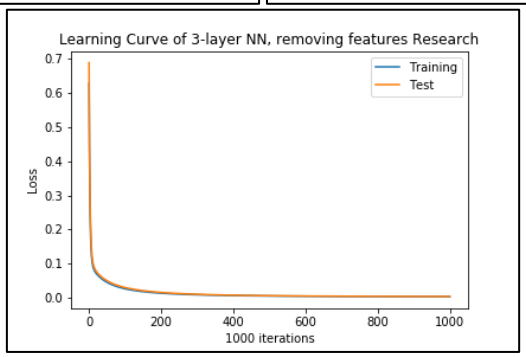












Learning curves of 7 input features vs 6 input features

		Training loss	Test loss
Feature removed	None	0.00746	0.00795
	GRE Score	0.00381	0.00438
	TOEFL Score	0.00379	0.00434
	University Rating	0.00387	0.00440
	SOP	0.00385	0.00440
	LOR	0.00372	0.00412
	CGPA	0.00382	0.00388
	Research	0.00379	0.00421

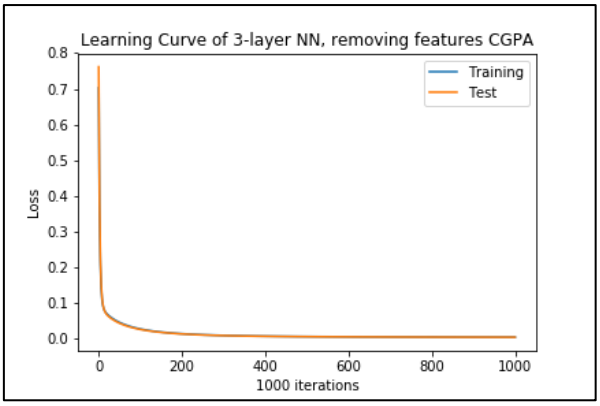
6 input features, 1 feature removed.

According to the figures, the test loss is lowest when the CGPA feature is removed. This result makes sense because even though CGPA is highly correlated with the chance of admit (meaning removing it may make it harder to predict chance of admit), its correlation with GRE score and TOEFL score makes CGPA a roughly redundant feature (i.e. removing CGPA simplifies or reduces the number of input dimensions but does not compromise the information conveyed by the dataset or reduce its variance).

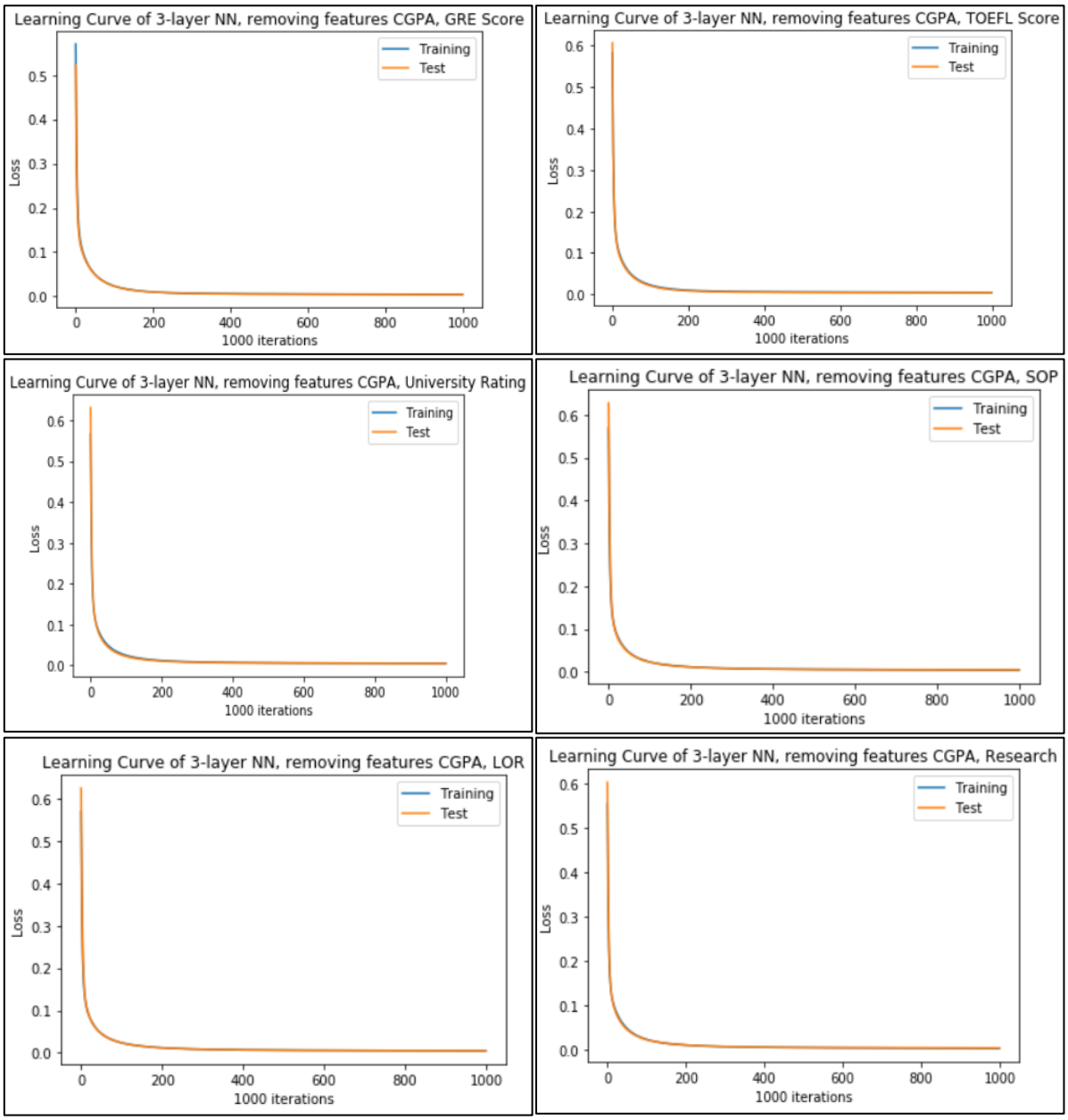
Moreover, since the test loss when CGPA is removed from the input is lower than that when no features are removed, RFE continues, trying to remove a second feature from the input to see if it will make the model more accurate.

Models with 5 input features

The results of trying to remove a second feature from the input dataset during RFE (i.e. learning curves and final training and test loss) are shown in the next pages:



VS



Learning curves of 6 input features vs 5 input features

		Training loss	Test loss
Features removed	CGPA	0.00382	0.00388
	CGPA and GRE Score	0.00438	0.00428
	CGPA and TOEFL Score	0.00463	0.00420
	CGPA and University Rating	0.00461	0.00446
	CGPA and SOP	0.00465	0.00460
	CGPA and LOR	0.00466	0.00474
	CGPA and Research	0.00469	0.00519

6 input features, 2 features removed.

According to the figures, when two features are removed, the test loss is lowest when CGPA and TOEFL score are removed from the input data. This makes sense as CGPA and TOEFL score have high correlations with GRE score, meaning that removing them will not alter too much the variance of the dataset because these two features are 'redundant' (i.e. CGPA and TOEFL score does not add much variance/information gain when GRE score is already present). However, this test loss is higher than that when only CGPA is removed from the input. This is presumably because only GRE score would be the only feature to have a moderately high correlation with the chance of admit (0.798). Hence this information loss is significant enough that the model fits less with the reduced input compared to when only CGPA is removed.

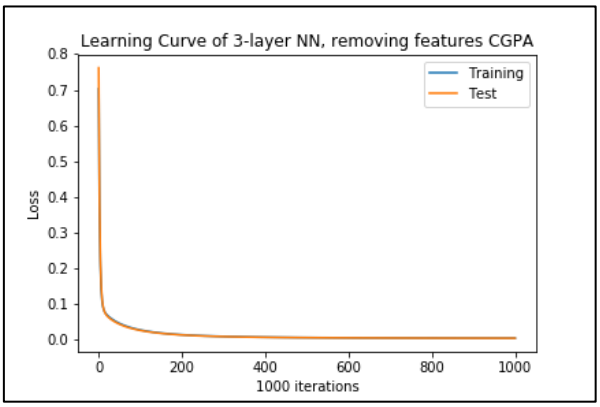
Due to this, RFE stops there and by this experimental result, the optimal feature set chosen for subsequent training of models is:

{GRE score, TOEFL score, University Rating, SOP, LOR, Research}

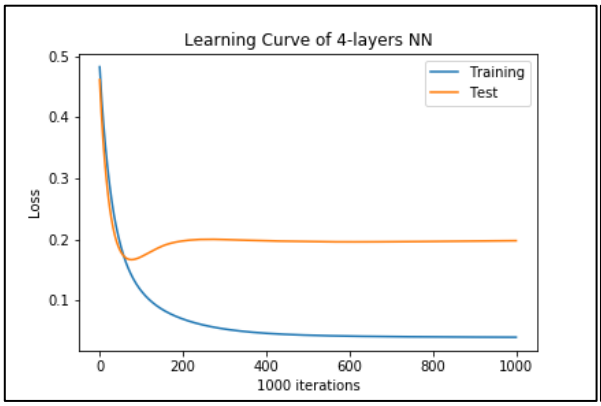
4. 4-layer and 5-layer neural network using optimal feature set with and without dropout

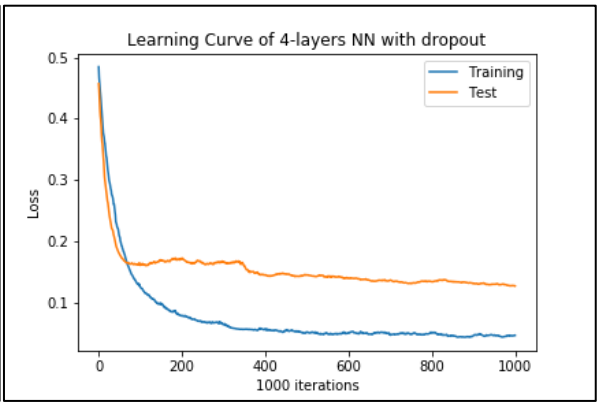
The 4-layer and 5-layer neural network mentioned in the **Model Architecture and Training** section will be trained once with dropout and once without dropout. The optimal feature set obtained in RFE of the 3-layer neural network will be used here.

The results of 4-layer and 5-layer neural networks with and without dropout, as well as with the 3-layer network are shown below:

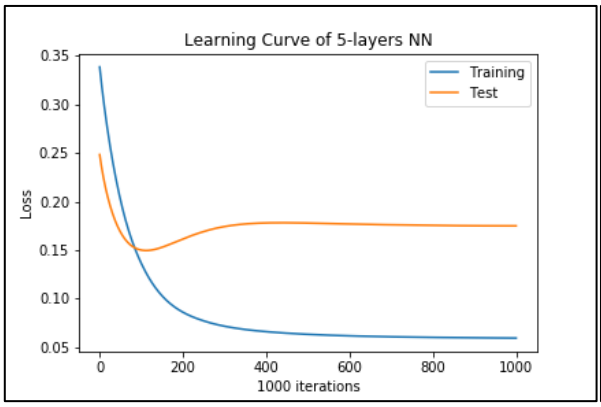


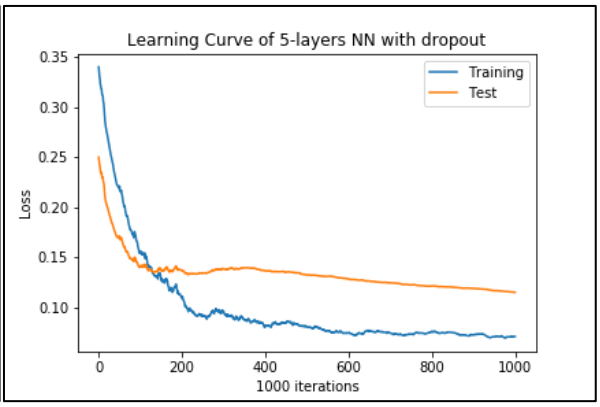
VS





VS





Train vs test accuracies of different model networks

		Training loss	Test loss
Network architecture	3-layer NN - (10 hidden neurons)	0.00382	0.00388
	4-layer NN	0.0400	0.198
	4-layer NN with dropout	0.0458	0.127
	5-layer NN	0.0596	0.175
	5-layer NN with dropout	0.0711	0.115

Comparisons between the 4-layer and 5-layer neural networks and the 3-layer neural network

The 4-layer neural networks and 5-layer neural networks (both with and without dropout) performed far worse than the 3-layer neural network. This is because of two reasons:

1) Because the 4-layer and 5-layer neural networks both have hidden layers of 50 neurons each (compared to the 3-layer neural network with one hidden layer of 10 neurons), the **regularization term** for these networks would be very high because of the much higher number of weights in these networks.
    a. The 3-layer neural network would have 6 * 10 + 10 = 26 weights to train.
    b. The 4-layer neural network would have 6 * 50 + 50 * 50 + 50 = 2850 weights to train.
    c. The 5-layer neural network would have 6 * 50 + 50 * 50 * 2 + 50 = 5350 weights to train.
2) Because of the sheer increase in the number of hidden-layer neurons and number of hidden layers for a simple dataset with only 6 inputs and 1 output, the 4-layer and 5-layer neural networks **overfit** the training dataset. Notice how, the test loss to training loss difference between the 3-layer neural network is nearly zero; but for the 4-layer and 5-layer neural network, the test loss could be up to 40 times as much as the training loss (showing that overfitting is happening).

Comparisons between the neural networks with and without dropout

When comparing the networks **with and without dropout**, the 4-layer and 5-layer neural networks performed far better with dropout than without. For the 4-layer neural network, the test loss for the network with dropout is much lower at 0.127 than without dropout at 0.198. While for the 5-layer neural network, the test loss for the network with dropout is

much lower at 0.115 than without dropout at 0.175. Notice that in the learning curve in the 4-layer and 5-layer network without dropout, the test loss dropped to a minimum before 100 epochs before increasing, signalling some form of overfitting. This increase is not present in the learning curves of the 4-layer and 5-layer networks with dropout (i.e. test loss continues to decrease, albeit it fluctuates regularly). This shows how well dropout is at reducing the extent at which the network overfits, since it reduces variance, for both network architectures.

Comparisons between the 4-layer network and the 5-layer network

When comparing **the 4-layer and 5-layer neural network**, the 5-layer neural network (with and without dropout) overall performed better than the 4-layer neural network. This is a kind of anomaly, showing that the 5-layer neural network was able to find a more general estimate for the function to approximate **chance of admit** better than the 4-layer neural network did.

Conclusions

In this project, we have learned how to tune hyperparameters for a neural network architecture, as well as experimented if more hidden layers could make a neural network perform better. We have also tried recursive feature elimination to find the optimal feature set for model training, as well as tried the effects of dropout on a neural network as compared to without.