

CZ 4042

Neural Networks and Deep Learning
Assignment 1 Report

Tan Chuan Xin
U1821755B

Table of Contents

Prologue.....	3
Part A: Classification Problem	4
Introduction.....	4
Methods.....	4
Normalization	4
Mini-Batch Gradient Descent	5
Loss Function – SparseCategoricalCrossentropy (ce)	5
k-fold Cross-Validation	5
Three-Way Data Split.....	6
Regularization.....	7
Custom Callbacks	7
Saving Data	8
Incremental analysis	8
Question 0 – Normalization method	9
Question 1 – Convergence	11
Question 2 – Batch size	12
Question 3 – Number of neurons.....	14
Question 4 – Weight decay parameter	16
Question 5 – Comparison against 4-layer network	18
Summary.....	19
Part B: Regression Problem	20
Introduction.....	20
Methods.....	20
Loss Function – MeanSquareError (mse)	20
Recursive Feature Elimination	20
Dropout Layers	21
Question 0 – Normalization method	22
Question 1 – Convergence	24
Question 2 – Recursive Feature Elimination	26
Question 3 – Number of Layers, Dropout	28
Summary.....	30

Prologue

This assignment was done through Jupyter Notebook. The repository has been hosted on Github, and cloning it will be the best way to get a copy of the code to run. It can be found at the following link: https://github.com/tanchuanxin/cz4042_assignment_1











Each individual question has a notebook dedicated to it. Please refer to the relevant notebooks alongside this project report for a better understanding.

This report will be written as a companion to the notebook. Code blocks will be omitted as much as possible, as this report shall center on the rationale of various decisions made that led to the code, rather than the actual code itself.

Source code

The source codes can be found under the folders titled 1a and 1b

The relevant .ipynb files are as such

For 1a	For 1b
 assignment_1a_q0.ipynb	 assignment_1b_q0.ipynb
 assignment_1a_q1.ipynb	 assignment_1b_q1.ipynb
 assignment_1a_q2.ipynb	 assignment_1b_q2.ipynb
 assignment_1a_q3.ipynb	 assignment_1b_q3.ipynb
 assignment_1a_q4.ipynb	
 assignment_1a_q5.ipynb	

Part A: Classification Problem

Introduction

Part A is a classification problem on the Cardiotocography dataset containing measurements of fetal heart rate (FHR) and uterine contraction (UC) features on 2126 cardiotocograms classified by expert obstetrician. The end goal is to output a class label (N: Normal; S: Suspect; P: Pathologic) given a set of input values using a classification neural network.

Five tasks were given, and I have created a 0th task, which is to determine the optimal normalization method to apply on the data set. They are broadly to determine for a 3-layer neural network, the

0. Optimal normalization method
1. Number of epochs required for convergence
2. Optimal batch size, by speed and accuracy
3. Optimal number of neurons
4. Optimal weight decay parameter for regularization
5. Comparing optimal 3-layer network found from above, against a specified 4-layer network

The primary goal of this classification problem is to determine the best set of hyperparameters for the classification model to achieve the best prediction results for the class labels.

Methods

There are several primary techniques that are common throughout the six notebooks used for Part A. We shall address why these methods were necessary, and hence chosen.

Normalization

The goal of normalization is to adjust the data that we have, such that the different variables all share a similar range of values. Intuitively, if we have a variable ranging from [1, 10] versus another that ranges from [1, 10000], the latter might affect the result significantly, simply through magnitude of the range. Therefore, normalization seeks to utilize the relative range of variables rather than the absolute range to achieve a better prediction.

There are two main methods of normalization, namely scaling and standardizing.

Scaling – Min max method	Standardizing – Gaussian method
Scale the inputs such that $x_i \in [0, 1]$: $\tilde{x}_i = \frac{x_i - x_{i,min}}{x_{i,max} - x_{i,min}}$	Normalize the input to have standard normal distributions $x_i \sim N(0,1)$: $\tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$
More useful for distance based algorithms like KNN, SVM	More useful for gradient-descent based algorithms like neural networks

The notebook assignment_1a_q0.ipynb will test out normalization through scaling and standardizing, to determine which normalization method will suit the data more.

We also need to be careful when normalizing our data. What we should do is

1. Split train and test data
2. Normalize the train data first and obtain a normalization function based on train data distribution
3. Apply that normalization function on the test data

By doing so, we prevent the test data characteristics from leaking over into the train data. We also ensure that we are truly evaluating if the model we have trained is able to generalize its learnings from the train data onto the test data.

Mini-Batch Gradient Descent

This is typically used when building neural networks, instead of Gradient Descent, or Stochastic Gradient Descent. Mini-batch gradient descent balances the strengths of both GD and SGD, allowing us to reduce training time, improve accuracy of the model, and improve the stability of the model. This allows us to achieve a more well-rounded model. However, the best batch size can only be determined through empirical testing.

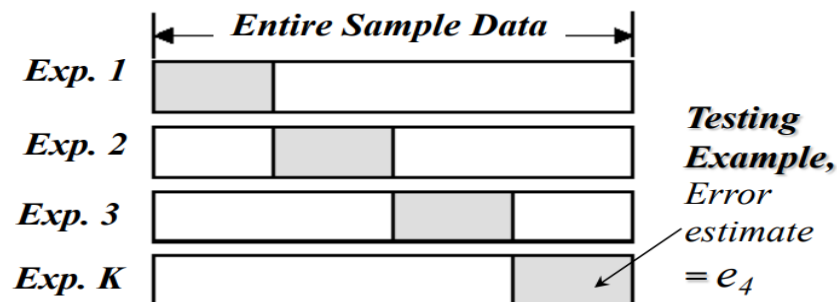
Mini-batch gradient descent is implemented through setting the `batch_size` parameter in a `model.fit()` function call.

Loss Function – SparseCategoricalCrossentropy (ce)

Loss functions are critical for a neural network to learn. Choosing the appropriate loss function can make or break the model. Given that this is a classification problem with more than two categories, and observing the distribution of the class labels, I have decided that SparseCategoricalCrossentropy will be an appropriate loss function to us. According to keras documentation, “Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided as integers”.

k-fold Cross-Validation

k-fold Cross-Validation is a form of cross-validation. It is a resampling procedure that breaks a dataset into a certain number of partitions with parameter k , and then using each in turn as a validation set to train the model. The use of cross-validation in general is to circumvent the problem of having very small data sets in which we cannot afford to employ the holdout method, or to avoid the problem of a unfortunate data split that holdout methods might cause.



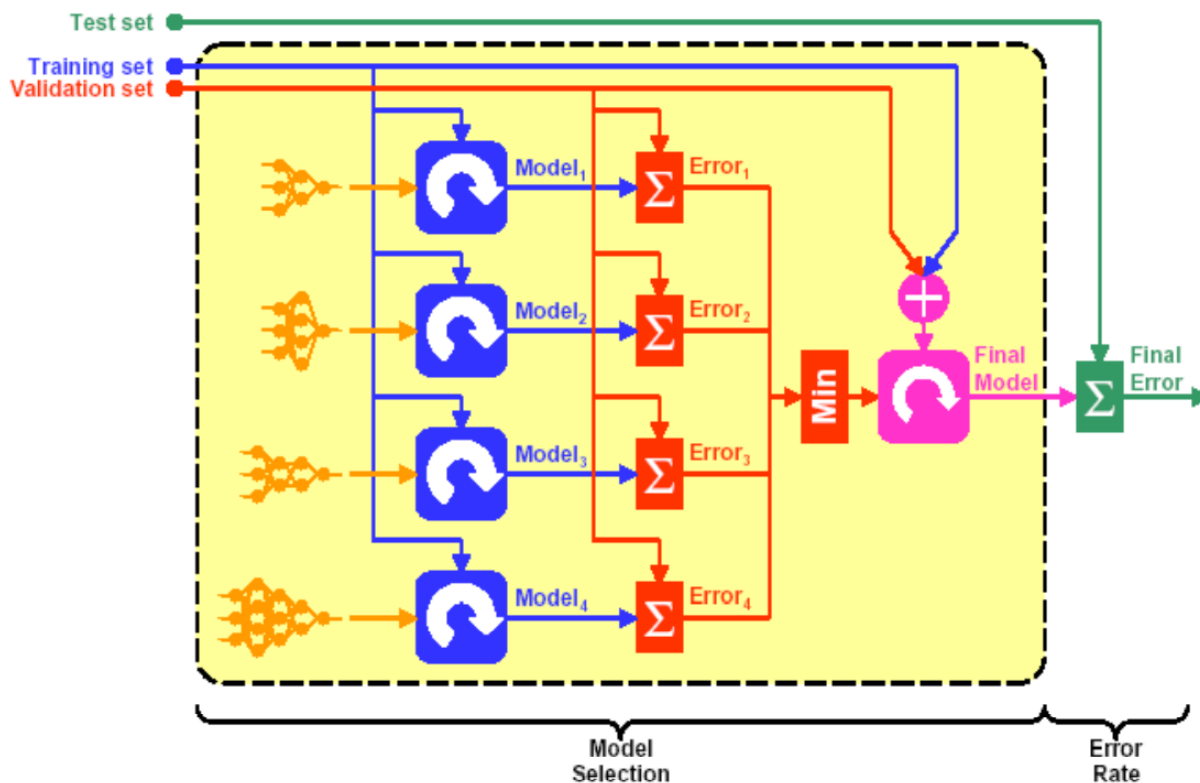
Given that the dataset for Part A only has 2126 entries, it is a small dataset and might benefit from k-fold cross-validation.

The greater the value of parameter k , the more partitions that dataset is split into. The drawback is computational cost – instead of one training cycle, we now must undergo k training cycles for the same dataset. Empirically, values of $k=5$, $k=10$ have been found to work well.

After obtaining the performance metrics for each fold, we will stack them onto each other and take the mean value across all the folds. This averaging will give us a better representation of the overall expected performance of the models.

Three-Way Data Split

This assignment requires us to generate different models, select a model, and evaluate the models' performance on a test set. This is exactly a situation that calls for a three-way data split. In essence, the data we have is broken into <training, validation, testing> sets. The validation set is used together with the training set to calculate the entropy of a model and refine the model. The test set is held out and exclusively reserved for use as a final evaluation of the models' ability to generalize to unseen cases. Therefore, a three-way data split has been employed.



The train-test split is 70-30 for this assignment.

The train-validation split will also be 70-30. The validation split can also be generated through k -fold cross-validation as well. If that is the case, the split will depend on k .

Regularization

During training with small datasets, some model weights may become very large. This is termed as overfitting. This jeopardizes the models' ability to generalize and learn, as it has become 'fixated' on several small and huge weights and is detrimental to the model. Regularization essentially imposes a penalty on larger weights in the loss function and discourages weights from attaining too large a value and causing overfitting.

This is important for our small dataset, as we do not want the model to perform extremely well during training due to overfitting, but perform very poorly during testing because it has become too specialized on the training data distributions and characteristics.

L2 regularization is to be used, as specified by the assignment. It is represented by the formula below

$$J_1 = J + \beta_2 \sum_{ij} (w_{ij})^2$$

It is implemented in keras as such

```
weight_decay_parameter = 10e-6  
regularization = keras.regularizers.l2(weight_decay_parameter)
```

Custom Callbacks

Keras allows for the use of a powerful feature called callback. Callbacks are actions that can be performed at specific stages of a training process and allows us to interrupt the training process and collect data, or modify it accordingly. Custom callbacks have been implemented to track key information as required by this assignment. Two such callbacks were created.

TestCallback()

```
# custom callback to evaluate the test set at each epoch  
class TestCallback(keras.callbacks.Callback):  
    def __init__(self, X_test, Y_test):  
        self.X_test = X_test  
        self.Y_test = Y_test  
  
    def on_epoch_end(self, epoch, logs={}):  
        loss, accuracy, sparse_categorical_crossentropy = self.model.evaluate(self.X_test, self.Y_test, verbose=0)  
  
        histories_test['test_values']['accuracy'].append(accuracy)  
        histories_test['test_values']['loss'].append(loss)  
        histories_test['test_values']['sparse_categorical_crossentropy'].append(sparse_categorical_crossentropy)
```

This is used to perform an evaluation of the model on the test data after every single epoch. Typically evaluating the model on the test data will only be done at the very end, but due to the assignment requirement, we will evaluate on every epoch.

TimeHistory()

```
class TimeHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.times = []

    def on_epoch_begin(self, batch, logs={}):
        self.epoch_time_start = time.time()

    def on_epoch_end(self, batch, logs={}):
        self.times.append(time.time() - self.epoch_time_start)
```

This is essentially used as a timer for each training epoch. Mostly relevant when tuning the batch size hyperparameter

Saving Data

The train-test data is generated once, saved, and reused subsequently. This is very useful for hyperparameter tuning, as we do not want to constantly recreate the same data whenever we try out a new hyperparameter.

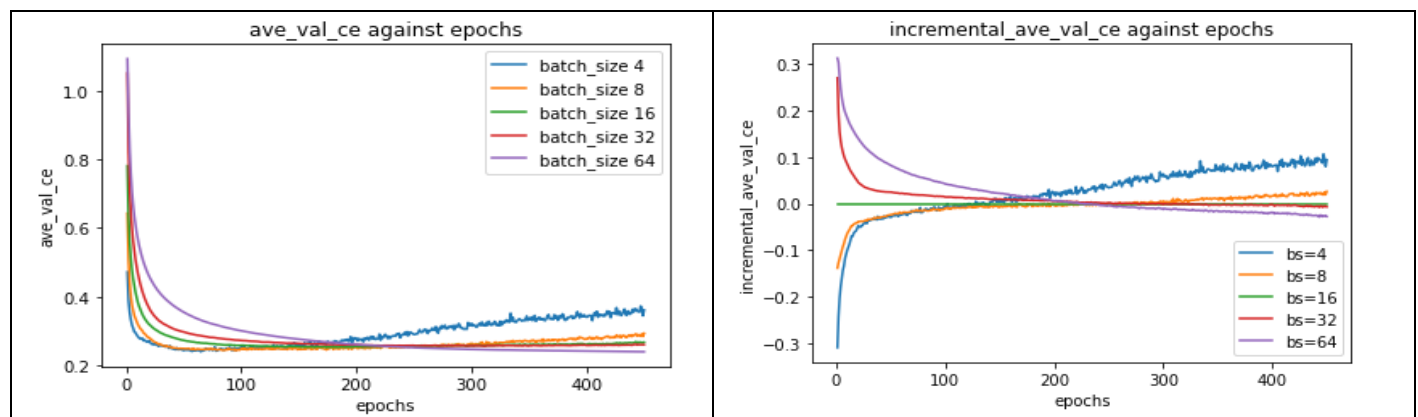
The history of each model generated by the model.fit() method will be saved. However because the custom callback **TestCallback()** does not generate a history, we have to manually populate a dictionary to mimic the structure of a history, and then save it. Logging these outputs is necessary, in order to analyze them afterwards without having to keep the jupyter notebook instance running constantly.

Two file formats are used - .json for history data, and .npy for the train-test data. Both are common file formats used.

Incremental analysis

For ease of comparison, and easier visualization of results, incremental analysis has also been conducted. Essentially, for a given set of data, one column will be subtracted from all the other columns to become a baseline zero on the graph. This will allow us to easier visualize models that are better or worse than it, when other models also have values that are very similar. This is nothing different from zooming in onto the graph, merely a different representation.

Take the following graph as an example. The incremental graph on the right simply shows all model's performance relative to the model with bs=16. Since it is a graph of crossentropy loss (ce), lines above the baseline green line are worse than it. The incremental analysis graphs might be used as well if they provide more clarity.



Question 0 – Normalization method

Experiments

This section performed one key experiment, which is to compare the normalization methods of scaling vs standardizing. The data was read in and passed through the starter_code provided for this question for fairness in comparison.

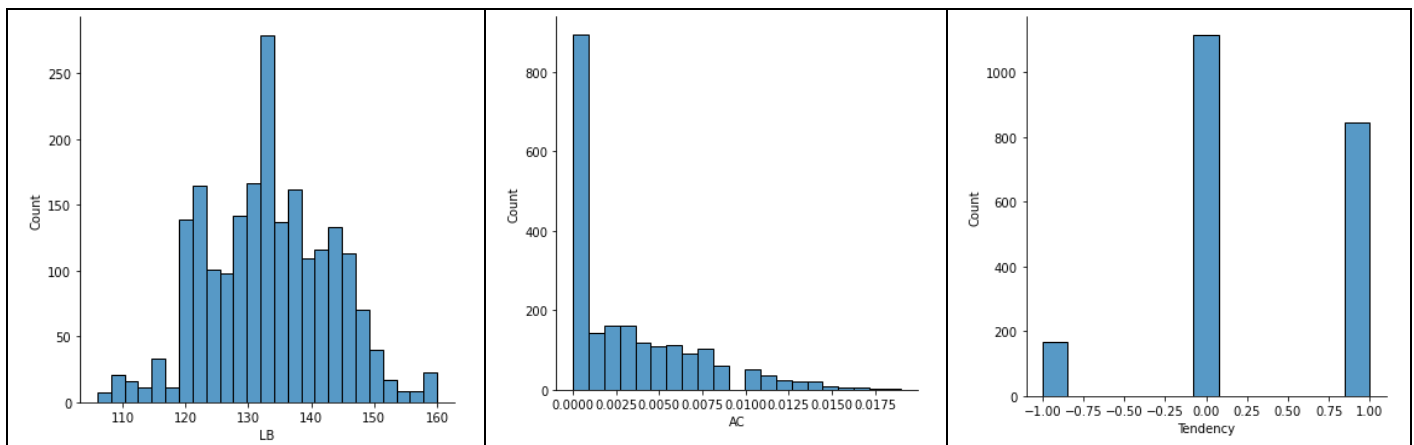
Pseudocode

```
# Load in data
  # visualize and understand data
# normalize data (scale, standard)
  # run through starter code
  # plot performance
  # compare normalization method
# save data
  # train-test-split
  # normalize
  # save for future use
```

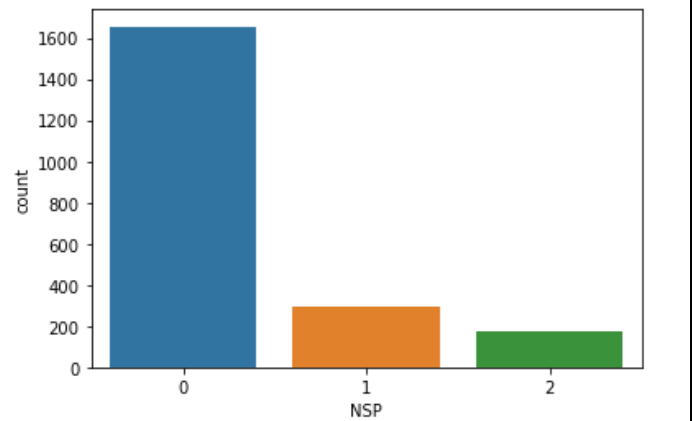
Results

Distributions

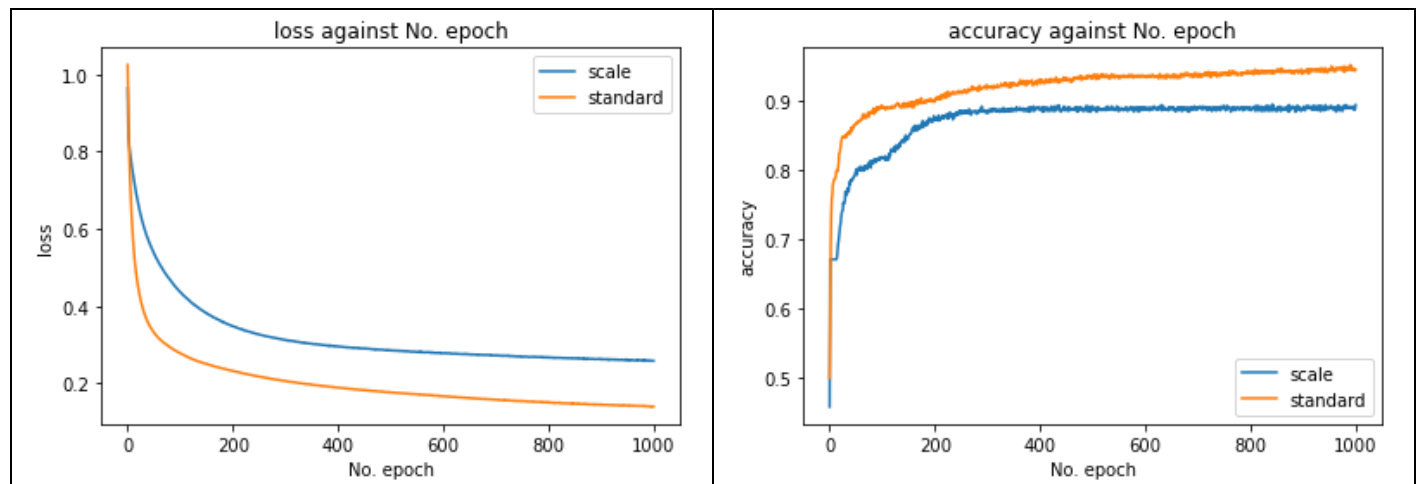
Plotting out the distributions of the variables showed that some variables resemble standard normal distributions, others exponential, and some are even discrete variables. These may have implications on the normalization method of choice. It is useful to understand the distributions of variables, because if they all appear to follow Gaussian distributions, we can very safely assume that standardizing will be a good option.



Plotting out the countplot of the target variable has a direct implication on the loss function that we will choose subsequently. For example, since this has three target variables, we should not use BinaryCrossentropy, but instead SparseCategoricalCrossentropy as the loss function.



Normalization performance



We can see quite clearly that standardizing the input variables yielded higher accuracy and lower losses versus scaling the input variables.

Conclusions

We shall utilize standardization as our normalization function of choice. This is different from the starter_code, which used simple scaling. Doing so will immediately improve our models' performance. It is important to get this first step right as a lot of performance gains can be gotten from simply choosing the right normalization method for the data.

The train-test set will be generated according to standardization.

```
# We shall use the same seed as per the starter code
X_train, X_test, Y_train, Y_test = train_test_split(df_x, df_y, test_size=0.3, random_state=seed)

X_train, X_train_normalizer = standard(X_train)
X_test = X_train_normalizer.transform(X_test) # applying X_train normalization params to X_test
X_test = pd.DataFrame(X_test, columns=X_train.columns)

print("train-test data normalized")
```

Question 1 – Convergence

Experiments

This section performed one key experiment, which is to determine the point whereby the model starts to converge. Convergence is when the model can perform no better even with more training, and represents the limit of the models' ability to learn, given the existing dataset. Several model parameters have been listed out for us

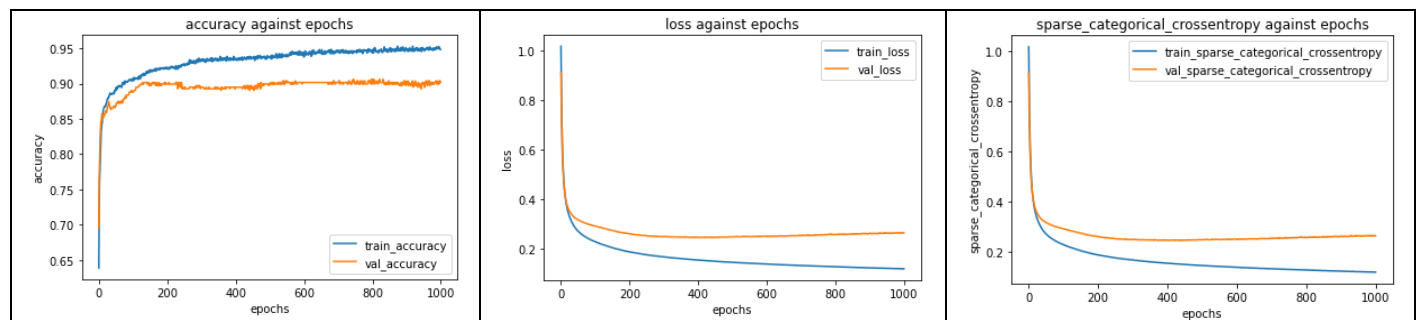
1. Design a feedforward neural network which consists of an input layer, one hidden layer of 10 neurons with ReLU activation function, and an output softmax layer. Assume a learning rate $\alpha = 0.01$, L2 regularization with weight decay parameter $\beta = 10^{-6}$, and batch size = 32. Use appropriate scaling of input features.
 - a. Use the training dataset to train the model and plot accuracies on training and testing data against training epochs.
 - b. State the approximate number of epochs where the test error begin to converge.

Pseudocode

```
# load in data
# define model parameters
  # set a high epoch count
# training model
  # create model
  # run the model
  # save results
# plot model performance
  # determine convergence point
# try on test set
```

Results

Training set history



We observe that the validation set has a minimum point, after which loss starts to increase, albeit slightly. This is a sign of overfitting but will be remedied as we continue tuning the hyperparameters.

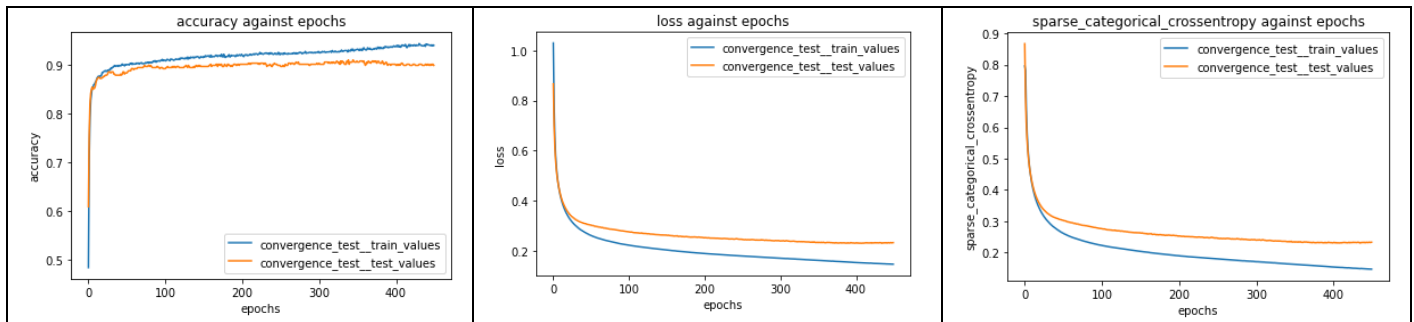
```
val_loss = histories['convergence_train']['val_loss']
print("val_loss convergence epoch: ", val_loss.index(min(val_loss)))

val_ce = histories['convergence_train']['val_sparse_categorical_crossentropy']
print("val_ce convergence epoch: ", val_ce.index(min(val_ce)))

val_loss convergence epoch: 405
val_ce convergence epoch: 405
```

We can see that around 405 epochs will be convergence for the model. We verify on the test set.

Test set history



We can see that the model has converged around 450 epochs, with stable values for the test data. Further training will likely lead to overfitting of the model.

Conclusion

Test error converges around 450 epochs. This will be used for subsequent parts.

Question 2 – Batch size

Experiments

This section performed one key experiment, which is to determine the optimal batch size for training this model. We are implementing mini-batch gradient descent for this assignment, the details of which have been mentioned under “Methods”.

5-fold cross validation is also implemented in this experiment, for every batch size that we are testing. Averaging out the performance metrics of the five folds for each batch size will give us a better representation of the overall expected performance of the model.

2. Find the optimal batch size by training the neural network and evaluating the performances for different batch sizes.
 - 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.
 - b. Select the optimal batch size and state reasons for your selection.
 - 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.

Pseudocode

```
# Load in data
# define model parameters
# set array of batch_sizes = [4,8,16,32,64]
# set the 5-fold indexes
# training model
# for each batch_size in batch_size:
#   for each fold
#     create model
#     run the model
```

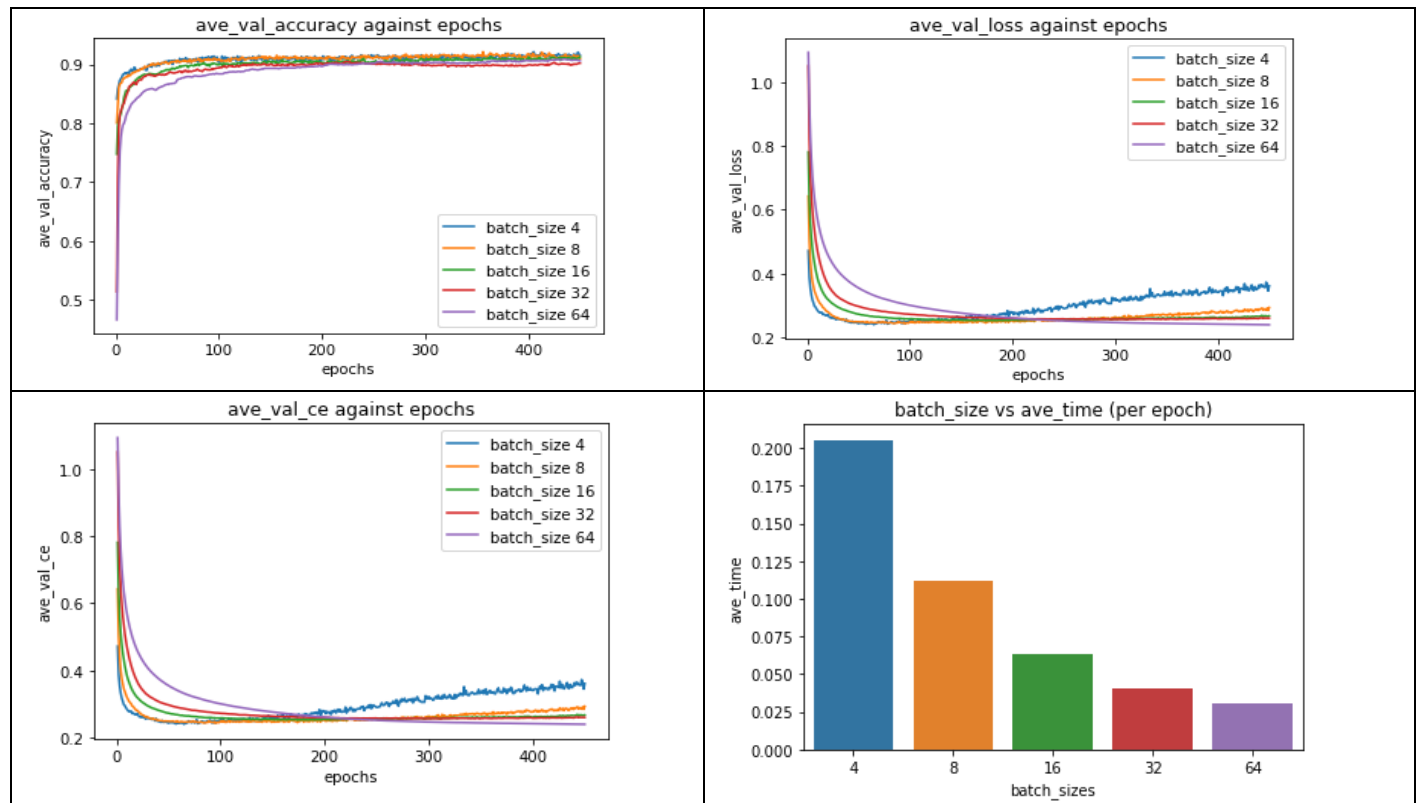
```

# save results
# average 5-fold metrics and time
# plot ave model performance
# determine best model
# try on test set

```

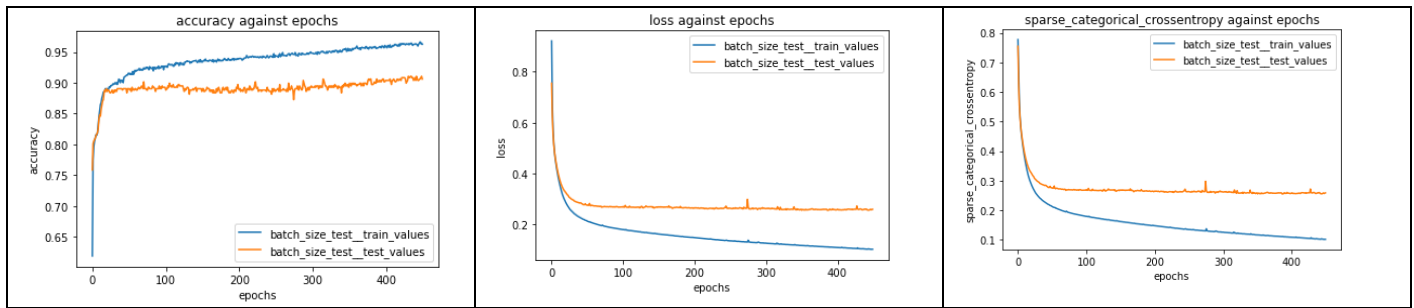
Results

Training set history



We can see that batch_size=16 provides significant time savings relative to the smaller batch sizes (only 1/3 the time of batch_size=4), while still providing good performance (accuracy is fully comparable to smaller batch size, and better than larger batch size) and stability (losses not increasing). We shall try with batch_size=16 and evaluate on the test set.

Test set history



We can see that using batch_size=16 on the test set, we have achieved very stable losses for the test set.

Conclusion

The optimal batch size is 16, on a combination of speed of performance and strength of performance.

Question 3 – Number of neurons

Experiments

This section performed one key experiment, which is to determine the optimal number of neurons when training this model. We are still using 5-fold cross validation.

The number of neurons affects the ability of the model to learn information. Too few neurons, and the network cannot store sufficient information about the data characteristics, leading to a poor prediction. Too many, and the network will overlearn and remember too many details about the training data, therefore generalizing poorly to the unseen cases.

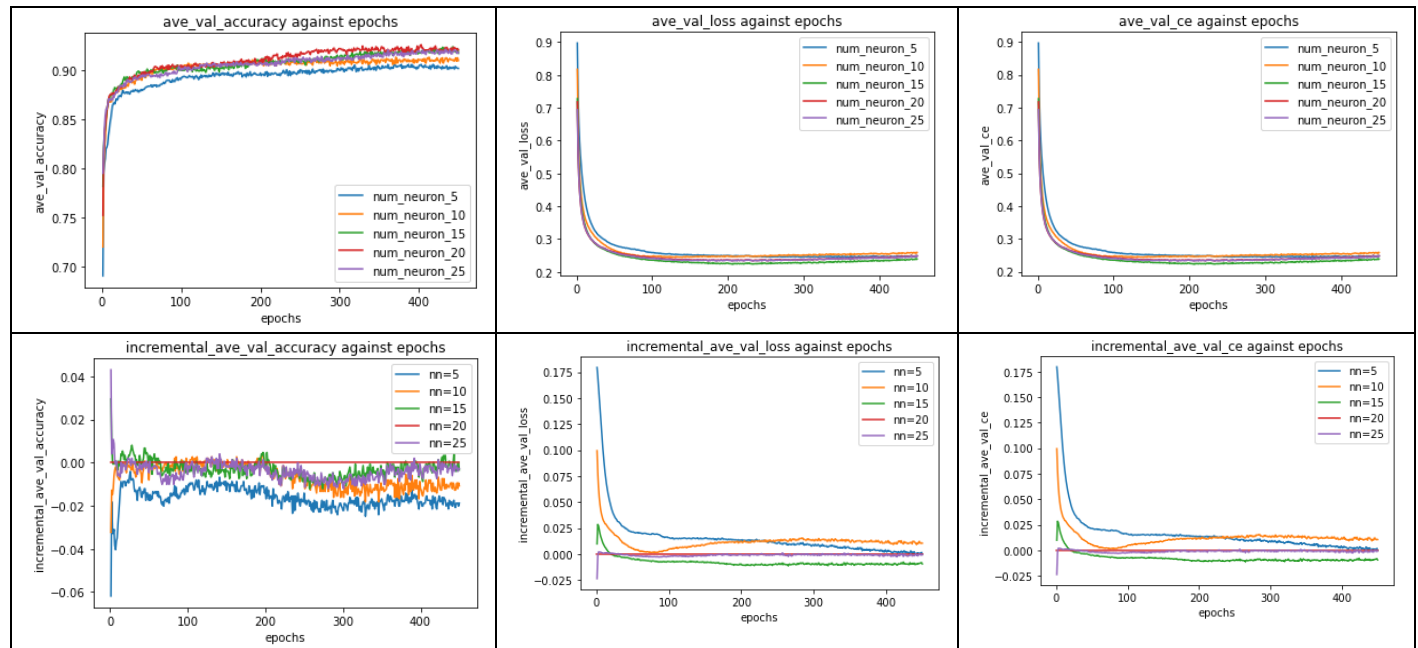
3. Find the optimal number of hidden neurons for the 3-layer network designed in part (2).
 - a. Plot the cross-validation accuracies against the number of epochs for different number of hidden-layer neurons. Limit the search space of number of neurons to {5,10,15,20,25}.
 - b. Select the optimal number of neurons for the hidden layer. State the rationale for your selection.
 - c. Plot the train and test accuracies against epochs with the optimal number of neurons.

Pseudocode

```
# Load in data
# define model parameters
# set array of num_neurons = [5,10,15,20,25]
# set the 5-fold indexes
# training model
# for each num_neuron in num_neurons:
#   for each fold
#     create model
#     run the model
#     save results
# average 5-fold metrics
# plot ave model performance
# determine best model
# try on test set
```

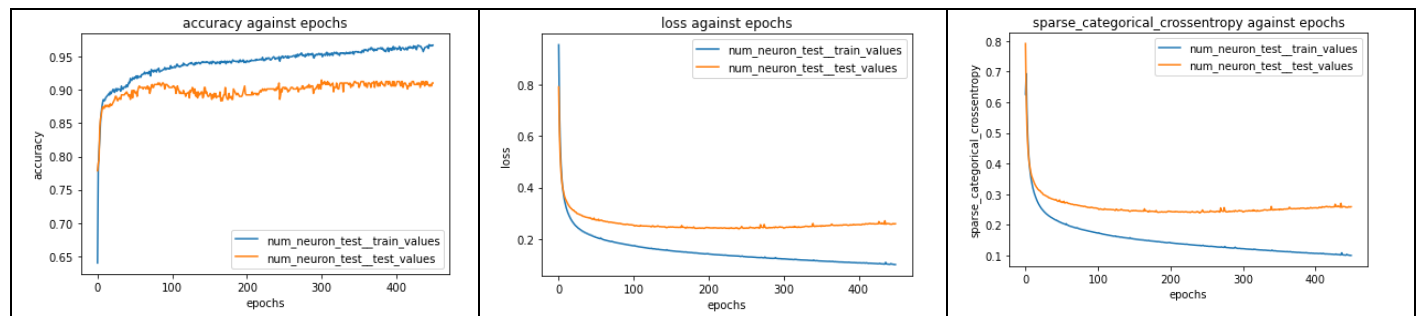
Results

Training set history



Observing the incremental analysis graphs might be easier. It shows that num_neurons=20 performs very well in losses against the best model num_neurons=15, but overall its num_neurons=20 has better accuracy versus every other model. We shall go with num_neurons=20 and evaluate on the test set.

Test set history



Once again, we can observe that our model with num_neurons=20 can achieve convergence for the test set

Conclusion

The optimal number of neurons will be 20. We shall use this going forward. A larger number of neurons will likely be able to learn more information about the data.

Question 4 – Weight decay parameter

Experiments

This section performed one key experiment, which is to determine the optimal weight decay parameter when training this model. We are still using 5-fold cross validation.

The weight decay parameter is provided to the regularization function as the regularization factor. The larger the weight decay parameter, greater the penalty on the larger weights. This is a common technique used to manage overfitting.

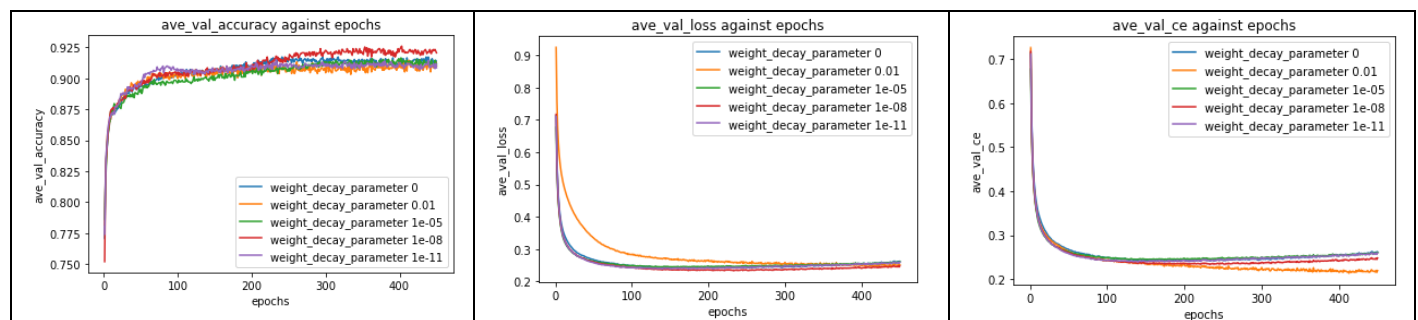
4. Find the optimal decay parameter for the 3-layer network designed with optimal hidden neurons in part (3).
 - a. Plot cross-validation accuracies against the number of epochs for the 3-layer network for different values of decay parameters. Limit the search space of decay parameters to $\{0, 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}\}$.
 - b. Select the optimal decay parameter. State the rationale for your selection.
 - c. Plot the train and test accuracies against epochs for the optimal decay parameter.

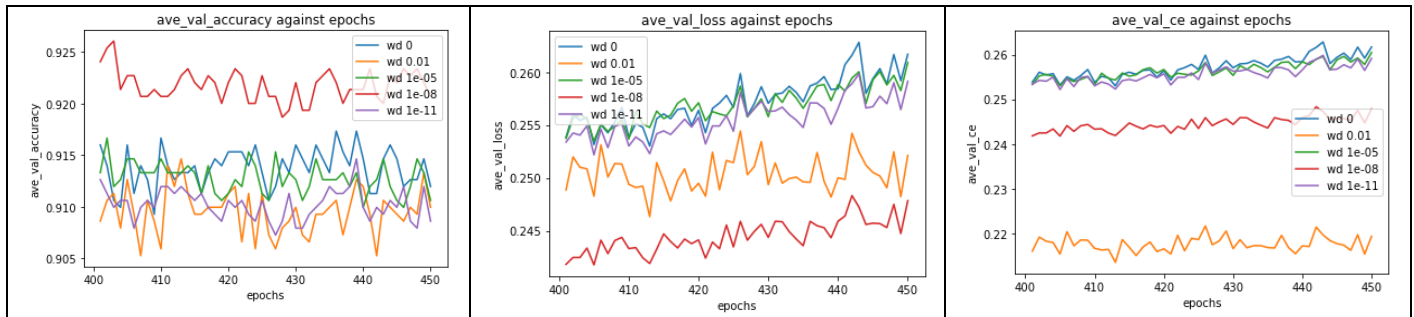
Pseudocode

```
# Load in data
# define model parameters
  # set array of weight_decay_parameters = [0, 10e-3, 10e-6, 10e-9, 10e-12]
  # set the 5-fold indexes
# training model
  # for each weight_decay_parameter in weight_decay_parameters:
    # for each fold
      # create model
      # run the model
      # save results
    # average 5-fold metrics
# plot ave model performance
  # determine best model
# try on test set
```

Result

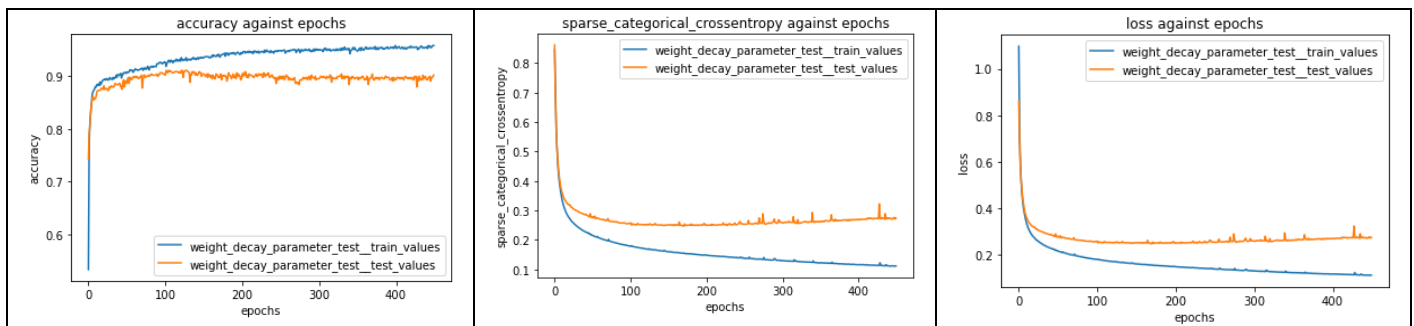
Train set histories





The second row of figures is a zoom in on the last 50 epochs. We can see that `weight_decay_parameter=10e-9` performs the best out of all the models in accuracy, as well as loss, and second best in crossentropy. It is quite a clear winner. We shall try `weight_decay_parameter=10e-9` on the test set

Test set histories



Once again, we can see that the values are stable

Conclusion

We should use `weight_decay_parameter=10e-9` going forward

Question 5 – Comparison against 4-layer network

Experiments

This section performed one key experiment, which is to compare the optimal 3-layer network found so far by answer q1-q4, against a specified 4-layer network.

Greater number of layers can learn more things about the training data. However, it might also learn too much, and overfit on the training data. This is an experiment to determine if an optimized 3-layer network can actually outperform a 4-layer network that may not have been optimized.

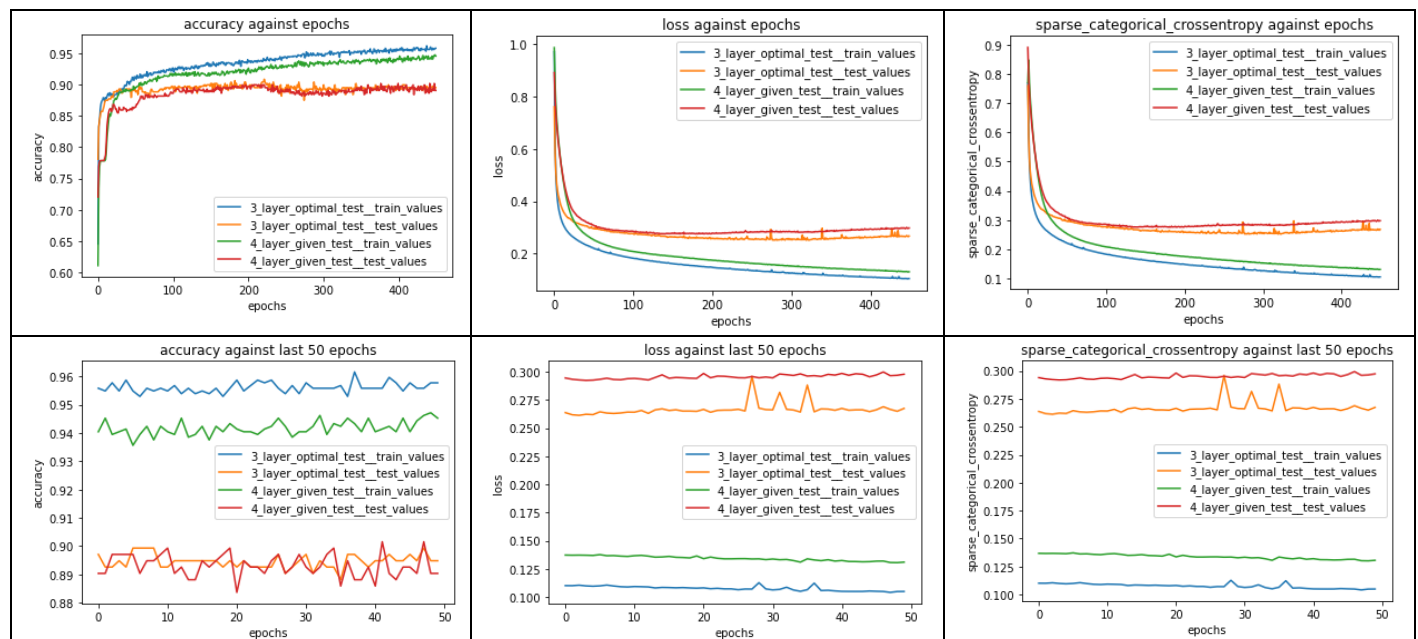
5. After you are done with the 3-layer network, design a 4-layer network with two hidden layers, each consisting 10 neurons, and train it with a batch size of 32 and decay parameter 10-6.
 - a. Plot the train and test accuracy of the 4-layer network.
 - b. Compare and comment on the performances of the optimal 3-layer and 4-layer networks.

Pseudocode

```
# Load in data
# define 3-layer parameters
# create, run, save model
# define 4-layer parameters
# create, run, save model
# compare model performance
# determine better model
# try on test set
```

Results

Test set histories



We can see that the 3-layer model has comparable accuracy values against the 4-layer model, but it has lower losses than the 4-layer model.

Conclusion

We can see that even though the neural network has fewer layers, it does not mean it cannot perform well. If we can finetune the hyperparameters for the neural network, it is able to output similar, if not better predictions than a neural network with more layers. However, this might also speak for the strength of deep neural networks. Without much finetuning done, it will be able to provide comparable performance to networks of lower depth.

Summary

The process of finetuning the hyperparameters of a model will eventually lead to us having a more robust model that can generalize to new and unseen test cases. We have experimented with several parameters in this assignment, namely the batch size, number of neurons in a hidden layer, the weight decay parameter of regularization, and lastly number of hidden layers.

While we may have arrived at several optimal features for our 3-layer neural network, these can only be discovered through empirical testing such as above. These hyperparameters depend greatly on the input dataset, and there is no golden rule for them.

Nonetheless, we can see the strength of hyperparameter tuning. A 3-layer network has comparable performance to a 4-layer network after much finetuning of the hyperparameters.

Part B: Regression Problem

Introduction

Part B is a regression problem on the Graduate Admissions Predication dataset containing parameters important in the application for a Masters Program. The end goal is to output a probability (Chance of Admit) for a set of input values using a regression neural network.

Five tasks were given, and I have created a 0th task, which is to determine the optimal normalization method to apply on the data set. They are broadly to determine for a 3-layer neural network, the

0. Optimal normalization method
1. Number of epochs required for convergence
2. Most redundant feature through Recursive Feature Elimination
3. Comparing number of layers, and effectiveness of Dropout layers

The primary goal of this regression problem is to determine the best set of features for the classification model to achieve the best prediction results for the admission probability, and if dropout will be useful.

Methods

There are several primary techniques that are common throughout the four notebooks used for Part B. We shall address why these methods were necessary, and hence chosen.

The following methods are common between Part A and Part B, and will not be elaborated on:

- Normalization, Mini-Batch Gradient Descent, Three-Way Data Split, Regularization, Custom Callbacks, Saving Data, Incremental analysis

Loss Function – MeanSquareError (mse)

The loss function for a regression problem is typically some form of square error. This loss function will effectively measure the “distance” away from the true value for a predicted value. It is one of the most common loss functions in regression problems.

MeanSquareError will be the measure of accuracy for a regression problem.

Recursive Feature Elimination

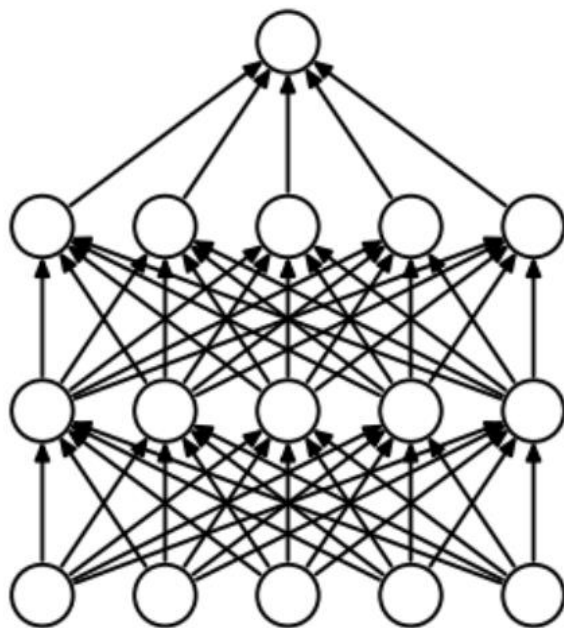
As described in the assignment handout, recursive feature elimination will allow us to eliminate the most redundant feature from the input feature space (variables). By removing some unnecessary features that do not contribute significantly to the model, or are detrimental to the model, we might be able to arrive at a better regression model than if we were to keep all the features in the training data. The idea is very simple, but might contribute immensely to the model performance.

Dropout Layers

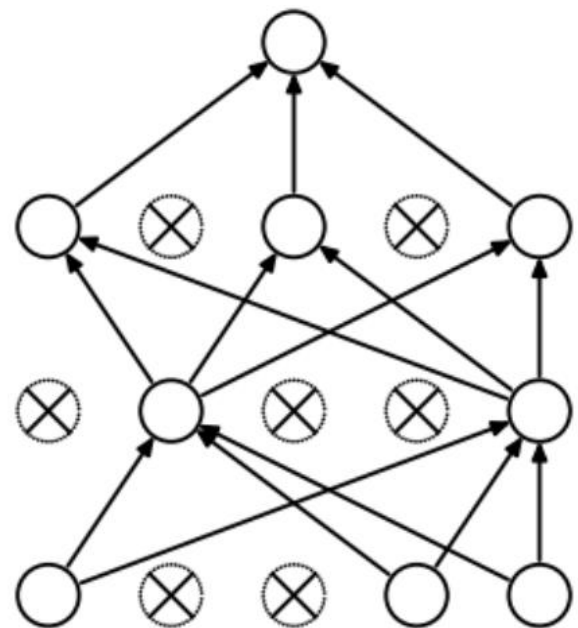
Dropout layers are layers that will randomly turn off some of the neurons in a neural network. The effect is that the weights learnt at those neurons become invalid, and no longer contribute to the model. The use of dropout has been found to help with the overfitting problem, whereby a model is unable to generalize.

Dropout is also a form of regularization, whereby we have a random chance to deactivate neurons, therefore the weights have a smaller chance of growing very large.

This may be an important addition to our network, since this dataset is only 400 rows and may be prone to overfitting.



(a) Standard Neural Net



(b) After applying dropout.

Typically, we would specify the number of features that we wish to keep and remove features up to the specified number. However, for this assignment, I have opted to remove all the way until the last feature remains, just for a more complete analysis.

Question 0 – Normalization method

Experiments

This section performed one key experiment, which is to compare the normalization methods of scaling vs standardizing. The data was read in and passed through the starter_code provided for this question for fairness in comparison.

We should realise first that this is a very small dataset – only 400 rows. Overfitting may become a big issue.

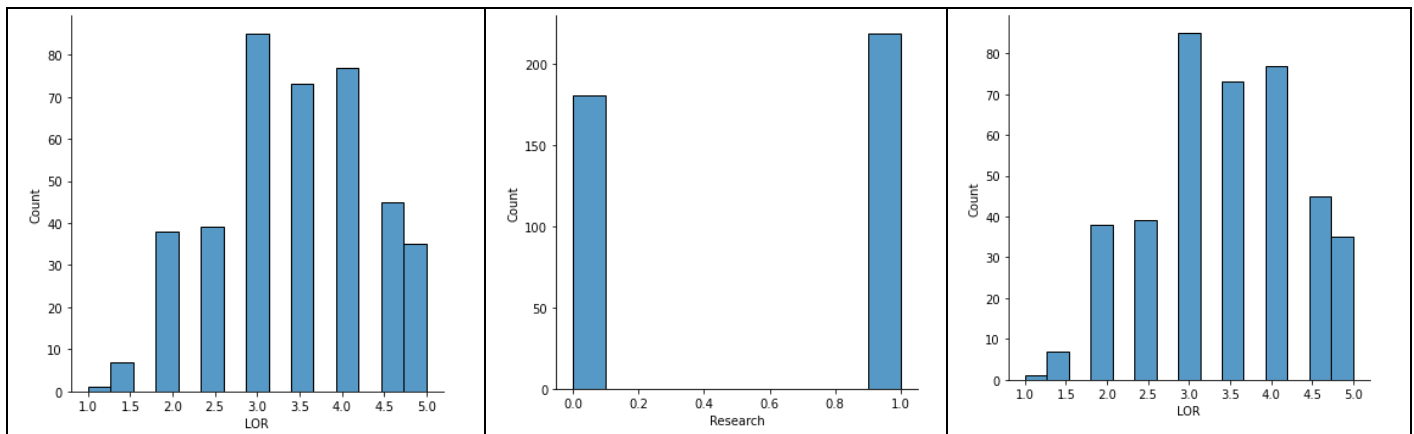
Pseudocode

```
# Load in data
# visualize and understand data
# normalize data (scale, standard)
# run through starter code
# plot performance
# compare normalization method
# save data
# train-test-split
# normalize
# save for future use
```

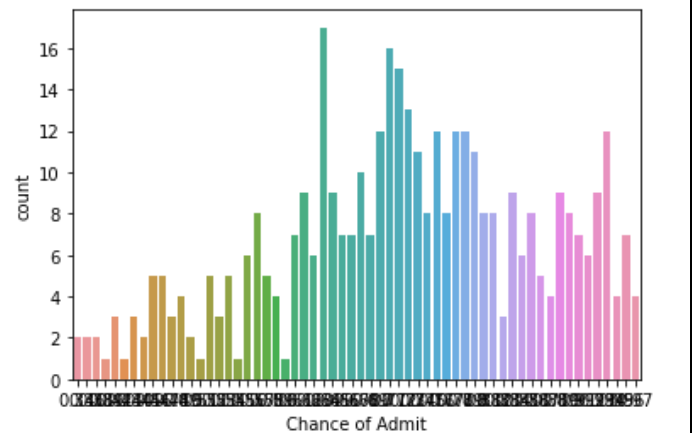
Results

Distributions

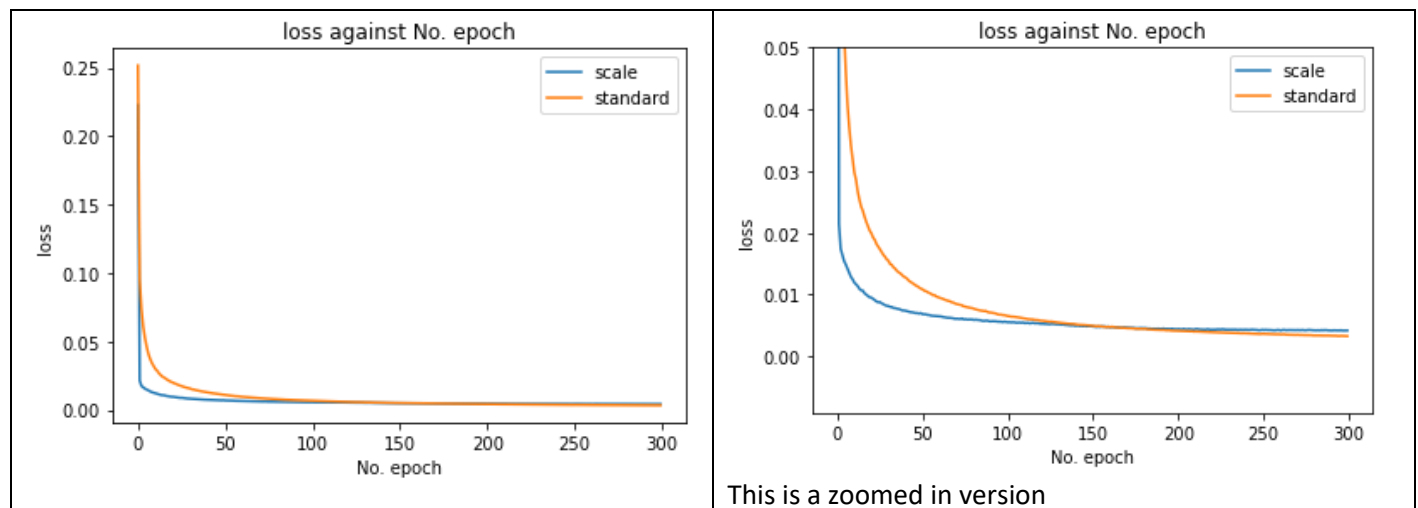
Like Part A, we can observe some discrete, and some standard normal distributed data



Plotting out the countplot of the target variable is not particularly useful in this case, since it is a regression problem.



Normalization performance



We can see that standardization eventually gives us better losses compared to scaling. It eventually has comparable loss after 100 epochs, and lower loss than scaling after around 170 epochs. However, both are very similar.

This is surprising because of the seven input features into the model, four of them (University Rating, SOP, LOR, Research) are discrete in nature. This is a situation where scaling could have proven useful instead of applying standardization. Nonetheless, the other three features (GRE Score, TOEFL Score, CGPA) might have swayed the decision towards standardization instead.

Conclusions

We shall utilize standardization as our normalization function of choice. We might also do well to use scaling, but we will stick with standardization for now.

The train-test set will be generated according to standardization.

Question 1 – Convergence

Experiments

This section performed one key experiment, which is to determine the point whereby the model starts to converge. Similar to Part A.

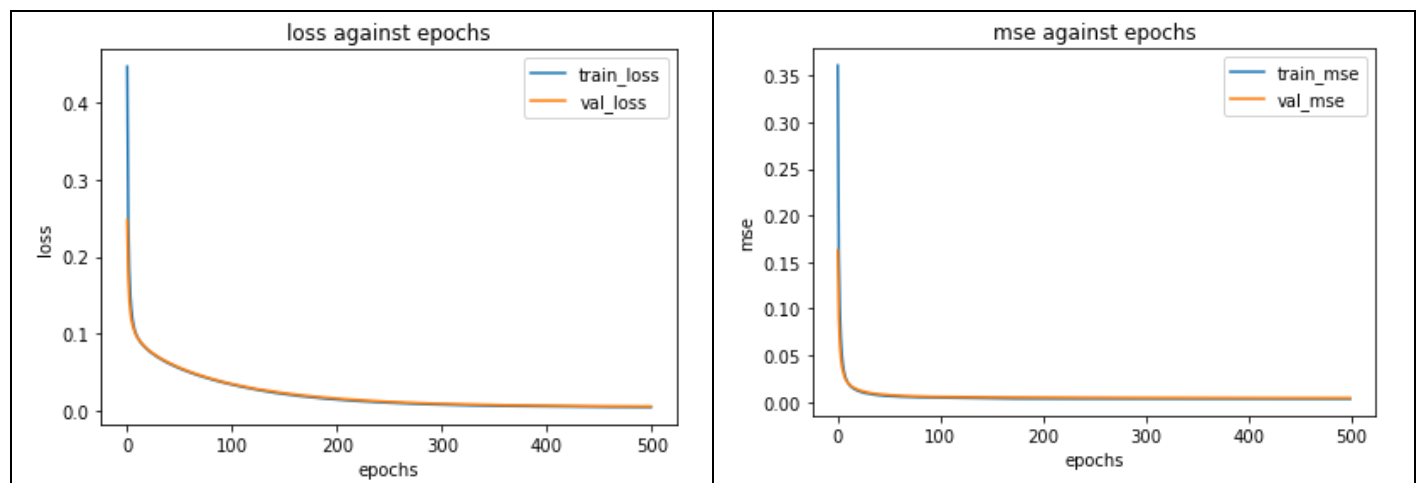
1. Design a 3-layer feedforward neural network consists of an input layer, a hidden-layer of 10 neurons having ReLU activation functions, and a linear output layer. Use mini-batch gradient descent with a batch size = 8, L_2 regularization at weight decay parameter $\beta = 10^{-3}$ and a learning rate $\alpha = 10^{-3}$ to train the network.
 - a. Use the train dataset to train the model and plot both the train and test errors against epochs.
 - b. State the approximate number of epochs where the test error is minimum and use it to stop training.
 - c. Plot the predicted values and target values for any 50 test samples.

Pseudocode

```
# Load in data
# define model parameters
    # set a high epoch count
# training model
    # create model
    # run the model
    # save results
# plot model performance
    # determine convergence point
# try on test set
# plot predicted and target value
```

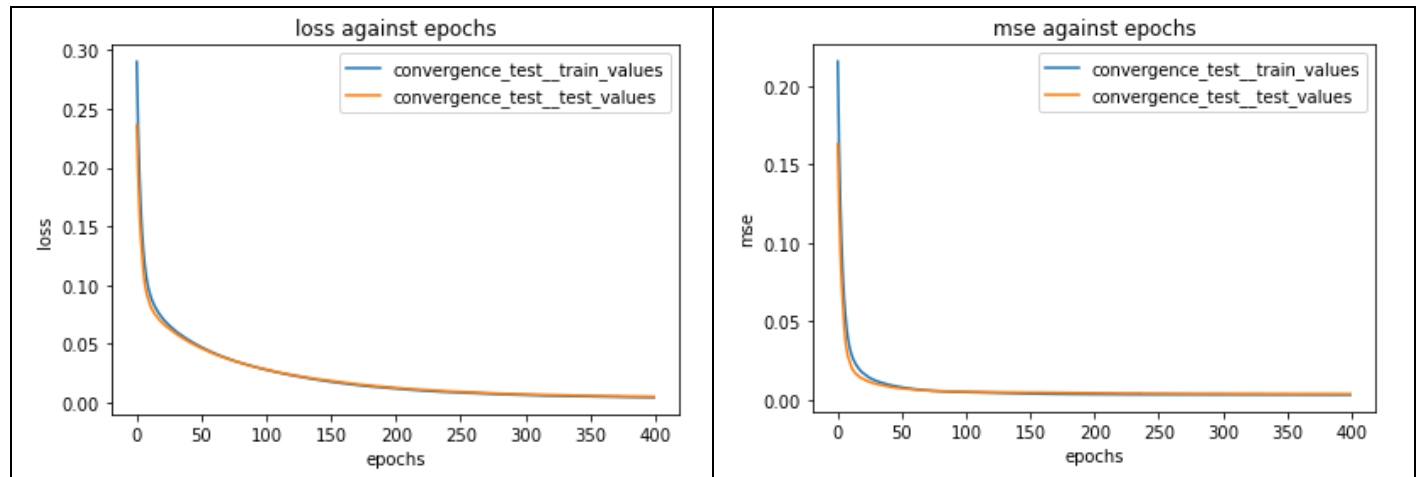
Results

Train set history



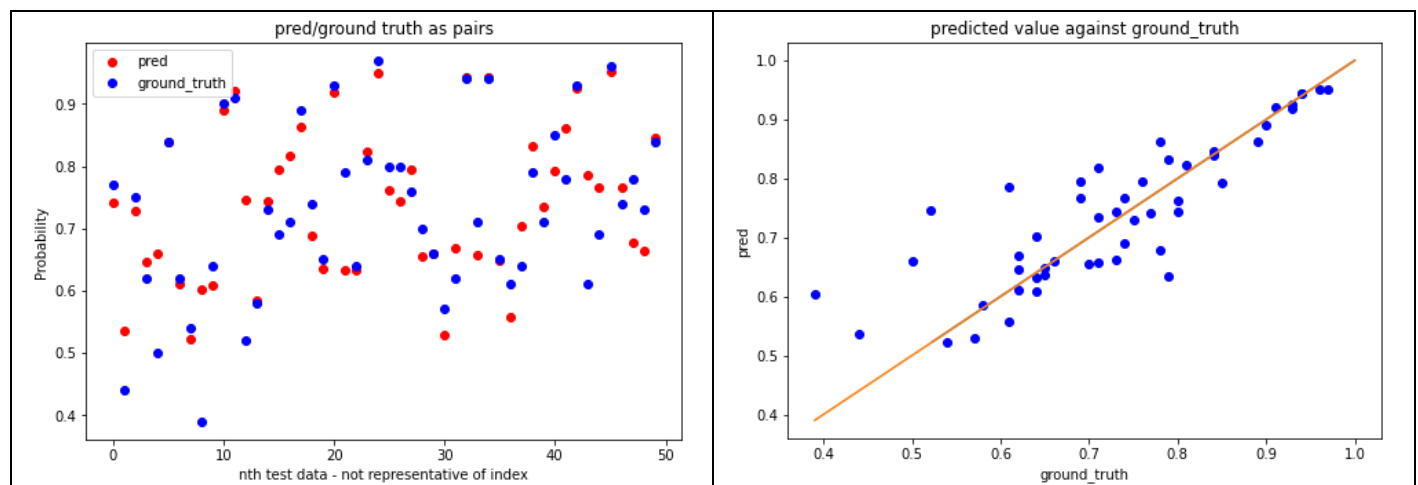
We can see that the losses will converge around 400 epochs. Let's try it on the test set.

Test set history



Similarly, we can see that we have gotten convergence at around 400 epochs for the test set. This appears to be a good enough epoch to train out model with.

Predicted values with test set



These are two representations of the prediction results from our model, versus ground truth (target values). 50 test points are plotted.

The graph on the left plots the prediction (red) + ground truth (blue) separately. The gap between two vertically collinear datapoints represents the prediction error.

The graph on the right plots a datapoint as a combination of prediction (y axis) and ground truth (x axis). A straight line of $y=x$ is drawn. Any point that lies on this line represents a perfect prediction. The further away from the line, the worse the prediction.

Conclusion

We can use 400 epochs to achieve convergence on the loss of our model.

Question 2 – Recursive Feature Elimination

Experiments

This section performed one key experiment, which is to implement RFE onto a given 3-layer model and identify the best performing feature space for this dataset. The RFE algorithm will start with a full-feature dataset, and iteratively drop every feature and test the performance of the model. The most redundant feature is established as the one that when dropped, provides the lowest losses or error to the model.

The original input feature space has seven features. These features will be labelled from f1 to f7. Hence for this first level, there will be seven combinations to test. The next level, a six-input feature space will have 5 combinations to test, and so on.

```
X_feature_list_mapping = ['GRE Score', 'TOEFL Score', 'University Rating', 'SOP', 'LOR', 'CGPA', 'Research']
X_feature_list = ['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7']
Y_feature_list = ['Chance of Admit']
```

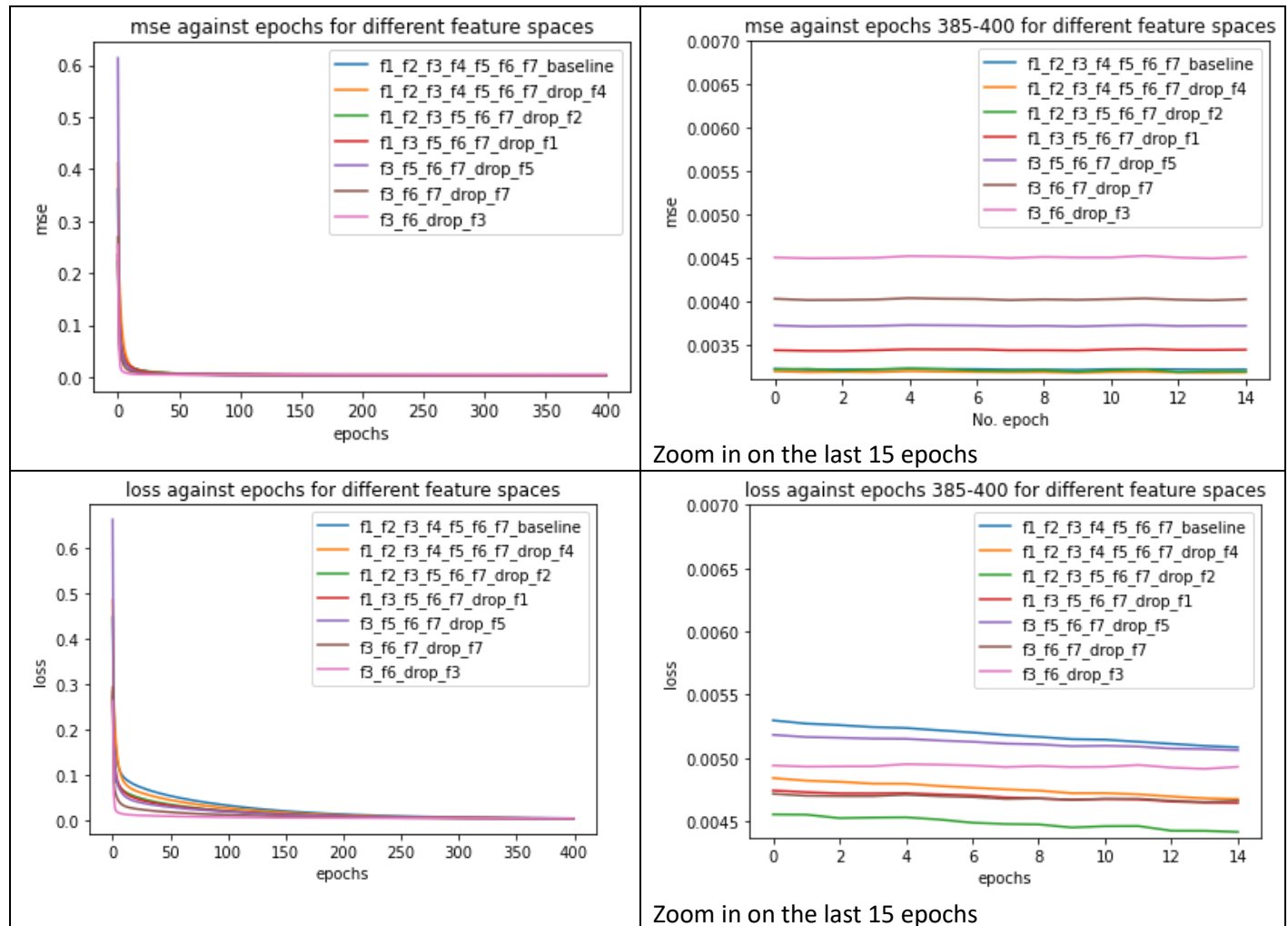
2. Design a 3-layer feedforward neural network consists of an input layer, a hidden-layer of 10 neurons having ReLU activation functions, and a linear output layer. Use mini-batch gradient descent with a batch size = 8, L_2 regularization at weight decay parameter $\beta = 10^{-3}$ and a learning rate $\alpha = 10^{-3}$ to train the network.
 - a. Use the train dataset to train the model and plot both the train and test errors against epochs.
 - b. State the approximate number of epochs where the test error is minimum and use it to stop training.
 - c. Plot the predicted values and target values for any 50 test samples.

Pseudocode

```
# Load in data
# define model parameters
# training model
  # best_mse = none
  # best_model = none
  # redundant_feature = none
  # call RFE():
    # if feature_length == 1
    #   return
    # if feature_length == 7
    #   evaluate model
    # for feature in feature_list:
    #   feature_list.drop(feature)
    #   evaluate model
    #     # if model_mse > best_mse
    #     #   best_mse = model_mse
    #     #   best_model = model
    #     #   redundant_feature = feature
    # RFE(feature_list.drop(redundant_feature))
# plot model performance
  # determine best model
# try on test set
```

Results

Test set history



How to read:

f1_f2_f3_f4_f5_f6_f7_baseline is the full-featured dataset.

f3_f6_f7_drop_f7 means that from a 3-input feature space that has f3, f6, f7, the feature f7 was dropped. Therefore it represents a 2-input feature space of f3_f6. Features f3 and f6 will be dropped in turn, evaluated, and the three results for this level will be compared

The best model at each level have been plotted on the graphs above. From above, we can see that the 6-input feature space (yellow) has marginally lower mse than the 5-input feature space (green).

```
{
  'f1_f2_f3_f4_f5_f6_f7_baseline':
    {'most_redundant_feature': 'None', 'lowest_mse': 0.0032243337482213974, 'corresponding_loss': 0.005084874574095011},
  'f1_f2_f3_f4_f5_f6_f7':
    {'most_redundant_feature': 'f4', 'lowest_mse': 0.003191093448549509, 'corresponding_loss': 0.005120051559060812},
  'f1_f2_f3_f5_f6_f7':
    {'most_redundant_feature': 'f2', 'lowest_mse': 0.0032013177406042814, 'corresponding_loss': 0.0048890975303947926},
  'f1_f3_f5_f6_f7':
    {'most_redundant_feature': 'f1', 'lowest_mse': 0.003451753407716751, 'corresponding_loss': 0.004798648413270712},
  'f3_f5_f6_f7':
    {'most_redundant_feature': 'f5', 'lowest_mse': 0.0037261336110532284, 'corresponding_loss': 0.005160422995686531},
  'f3_f6_f7':
    {'most_redundant_feature': 'f7', 'lowest_mse': 0.004031957592815161, 'corresponding_loss': 0.00466249929741025},
  'f3_f6':
    {'most_redundant_feature': 'f3', 'lowest_mse': 0.00451825003457069, 'corresponding_loss': 0.008966663852334023}
}
```

From here we can see the 6-input feature space just edges out the 5-input feature space in terms of mse, for a value of 0.00001 less, which is quite insignificant. However the losses for the 5-input feature space are visibly lower than that of the 6-input feature space at 0.0003 less.

Conclusion

We can now see that by applying RFE, we were able to get a better model and better prediction. It might be a worthwhile investment of our time to go through the RFE process before embarking on building our models, since it is more helpful to weed out non-contributing or detrimental features, rather than trying to build a model that can accommodate them.

The optimal feature space is found to be the 5-input feature space, that removed f2 and f4, corresponding to the features TOEFL Score, and SOP (statement of purpose) respectively.

Question 3 – Number of Layers, Dropout

Experiments

This section performed one key experiment, which is to compare 4-layer and 5-layer models, with and without dropout, against the 3-layer model found from the previous question. The input feature space will also be the truncated feature space found from question 2, which drops the features TOEFL Score, and SOP.

Dropouts are expected to help, since we have a very small amount of data to work with and the assumption is that the layers will tend to overfit.

The 3-layer model has 10 neurons in the hidden layer, the 4-layer and 5-layer models will all have 50 neurons in the hidden layers.

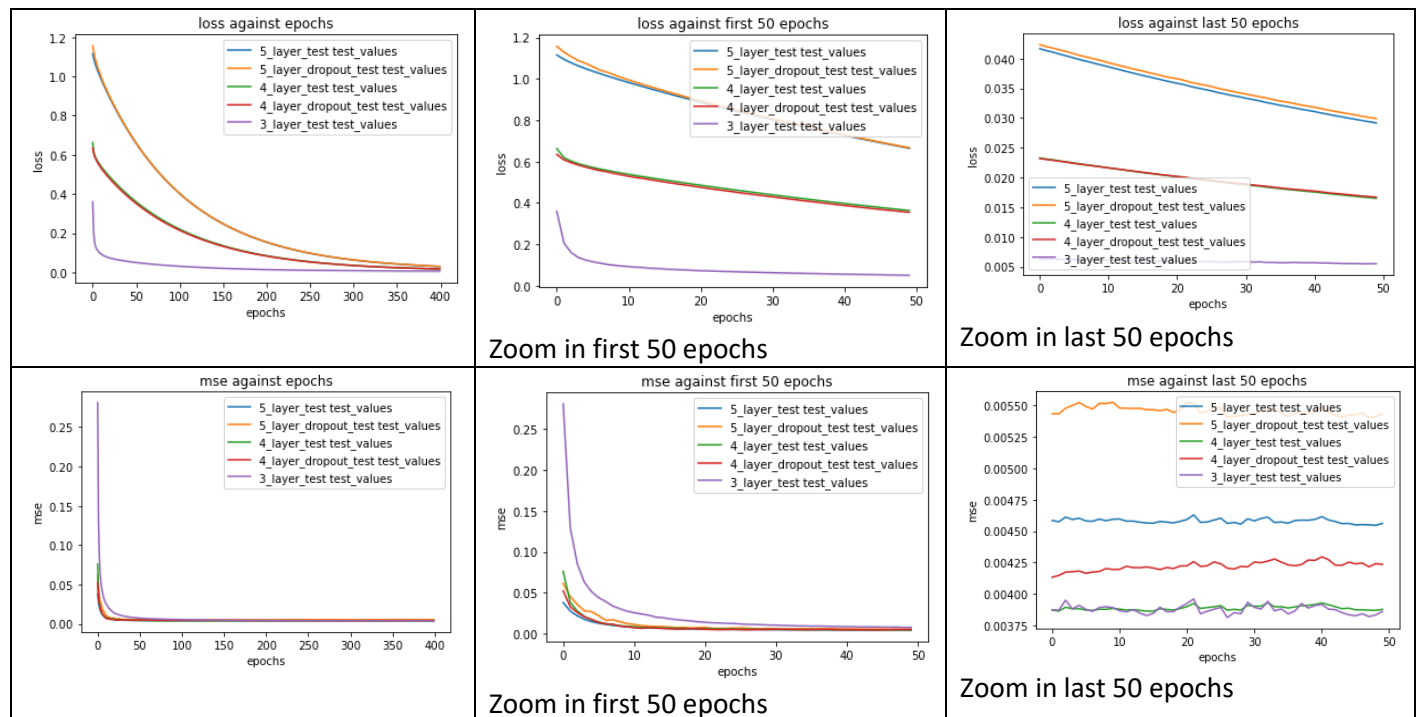
3. Design a four-layer neural network and a five-layer neural network, with the hidden layers having 50 neurons each. Use a learning rate of 10⁻³ for all layers and optimal feature set selected in part (3). Introduce dropouts (with a keep probability of 0.8) to the layers and report the accuracies. Compare the performances of all the networks (with and without dropouts) with each other and with the 3-layer network.

Pseudocode

```
# Load in data
# try on test set
  # for model in [3-layer, 4-layer, 4-layer-dropout, 5-layer, 5-layer-dropout]:
    # define model parameters
    # create model
    # run model
    # save results
# plot model performance
# determine best model
```

Results

Test set history



We can observe that the losses for the 3-layer model in the first 50 epochs is a lot lower than the other models, but the mse is the highest. This implies that the 3-layer model is the least accurate model in the first 50 epochs. However in the last 50 epochs, we can see that the 3-layer model now has both the lowest loss, and also is joint lowest in mse together with 4-layer (without dropout).

It appears that more layers, or more neurons/layer helps early on. We can see that in the first 50 epochs, the 4-layer and 5-layer networks had significantly lower mse than the 3-layer network. This is likely because the increased layer count and neuron count allowed the networks to learn much faster than a 3-layer network, therefore bringing down the mse much faster.

However more layers, or more neurons/layer did not help later. The greater the layer count, the worse it performed later into the training cycle. This might be a sign of overfitting over the data set since the entire data set is just 400 rows, and the training set is 280 rows.

What is surprising is that the dropout did not appear to help the models at all. For the 4-layer and 5-layer networks, those that implemented dropout fared worse than those that did not. It might be the case that either

- dropout is too much, which causes more information loss than it should have and therefore underfitting
- dropout not enough, thus the model is still overfitting

Conclusion

It appears that for dropout layers to help resolve the overfitting problem, the dropout rate must be carefully chosen. Choosing a wrong dropout rate might not be able to assist the model in learning at all.

The best model so far is a 3-layer model, with 10 neurons in the hidden layer, and not implementing dropout.

Summary

We have discovered that Recursive Feature Elimination can be a powerful tool in improving the performance of a neural network model. By eliminating features that are unimportant to the prediction, it will also speed up the training time while providing a boost in accuracy.

Furthermore, we realize that dropout rates need to be finetuned as a hyperparameter as well, in order to allow it to be effective in preventing overfitting of a model.

<<< END >>>