

**CZ 4042**

**Neural Networks and Deep Learning**

**Assignment 2 Report**

**Tan Chuan Xin**

**U1821755B**

Table of Contents

[Prologue 3](#_Toc53866285)

[Part A: Classification Problem 4](#_Toc53866286)

[Introduction 4](#_Toc53866287)

[Methods 4](#_Toc53866288)

[Normalization 4](#_Toc53866289)

[Mini-Batch Gradient Descent 5](#_Toc53866290)

[Loss Function – SparseCategoricalCrossentropy (ce) 5](#_Toc53866291)

[k-fold Cross-Validation 5](#_Toc53866292)

[Three-Way Data Split 6](#_Toc53866293)

[Regularization 7](#_Toc53866294)

[Custom Callbacks 7](#_Toc53866295)

[Saving Data 8](#_Toc53866296)

[Incremental analysis 8](#_Toc53866297)

[Question 0 – Normalization method 9](#_Toc53866298)

[Question 1 – Convergence 11](#_Toc53866299)

[Question 2 – Batch size 12](#_Toc53866300)

[Question 3 – Number of neurons 14](#_Toc53866301)

[Question 4 – Weight decay parameter 16](#_Toc53866302)

[Question 5 – Comparison against 4-layer network 18](#_Toc53866303)

[Summary 19](#_Toc53866304)

[Part B: Regression Problem 20](#_Toc53866305)

[Introduction 20](#_Toc53866306)

[Methods 20](#_Toc53866307)

[Loss Function – MeanSquareError (mse) 20](#_Toc53866308)

[Recursive Feature Elimination 20](#_Toc53866309)

[Dropout Layers 21](#_Toc53866310)

[Question 0 – Normalization method 22](#_Toc53866311)

[Question 1 – Convergence 24](#_Toc53866312)

[Question 2 – Recursive Feature Elimination 26](#_Toc53866313)

[Question 3 – Number of Layers, Dropout 28](#_Toc53866314)

[Summary 30](#_Toc53866315)

Prologue

This assignment was done through Google Colaboratory Notebooks. 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\_](https://github.com/tanchuanxin/cz4042_assignment_1)2

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 |

Part A: Object Recognition

# Introduction

Part A is an object recognition problem on the CIFAR-10 dataset, which contains images in 10 classes. This is also a classification problem. The end goal is to perform a multi-class classification and output a class label corresponding to the input image.

There are four tasks given, namely

1. Implement a Convolutional Neural Network (CNN) based on parameters that were provided
2. Perform a grid search to find optimal channels for the CNN
3. Experiment with various optimization techniques to improve performance
4. Compare every model created and discuss their performance

The primary goal of this problem is to discover a set of hyperparameters that are optimal to classify the images in the CIFAR-10 dataset when using CNNs.

# Methods

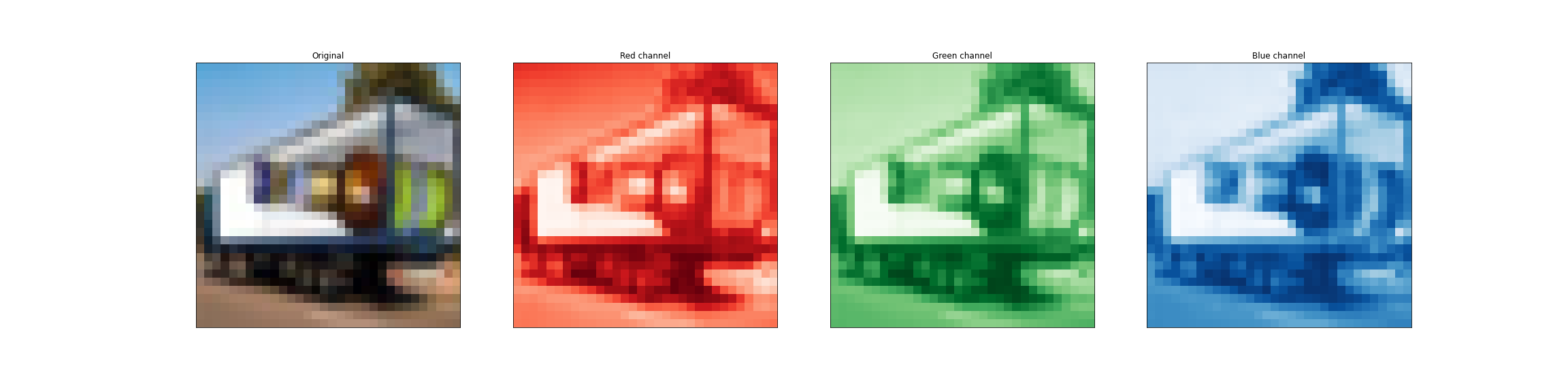
There are several primary techniques that are common throughout the notebooks used for Part A. We shall address why these methods were necessary, and hence chosen. We will also explain the methods used by the starter code, when deemed important.

## Data Preparation

Some changes were made to the load\_data(file) function in the starter code. When trying to plot out the source image for inspection, random noise was plotted out instead. It was discovered that there was a problem with the reshaping of the source data into proper images. Therefore, this line was added to ensure proper reshaping into a 32x32x3 dimension image.

  data = data.reshape(len(data), 3, 32, 32).transpose(0, 2, 3, 1)

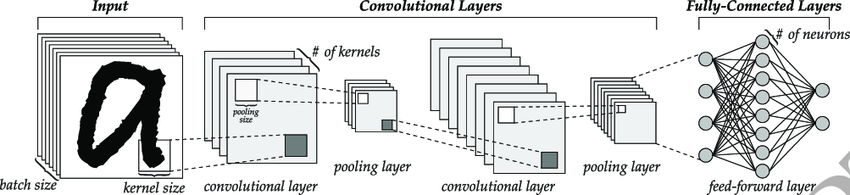
Before, the numpy array representations of the images were incorrectly prepared. Feeding these into our model provided incoherent predictions and very random intermediate layer activations. Therefore, it was important that the data was prepared correctly. Now, we are able to output the images as they were meant to be.



## Convolutional Neural Networks (CNN)

CNNs are a powerful form of feedforward neural networks, especially when it comes to tasks related to computer vision. CNNs are defined in a large part with convolutional layers and pooling layers.

Convolutional layers perform convolutional operations that involve taking a kernel (essentially a set of weights) and passing it across the entire input matrix to produce a feature map. Multiple kernels of varying sizes can be passed through the input matrix to produce various feature maps, that may correspond to certain characteristics of the input matrix. Feature maps are then passed through pooling layers, which perform local interpolation on feature maps to select just a few values from them, such as maxpooling, which takes the maximum value from a window of values.



The number of channels in a convolutional layer represent the number of kernels that are applied on the input matrix, which produces an equal number of feature maps for the image. Given that ‘feature maps’ are nothing but characteristics of a given input matrix (commonly images) represented as a matrix/vector, we expect that earlier convolutional layers should have more channels to capture various simple characteristics of the input, and later convolutional layers will have fewer channels that capture overarching themes of the earlier feature maps produced by earlier layers.

We can capture the feature maps at each convolutional layer and pooling layer by simply getting the outputs of each layer in a fully trained model, instead of just the terminal output at the softmax layer. This allows us to visualize what the convolutional and pooling layers are doing.

## Optimization

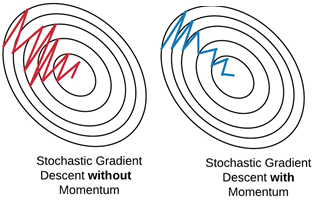
To enhance the performance of our model, various optimization techniques were used. The following were utilized in this question.

1. Stochastic Gradient Descent (SGD) / Mini-Batch Gradient Descent

SGD is one of the simplest learning algorithms that we can use. In essence, we compute the gradient of the cost function with respect to the parameter that we are optimizing (usually the weights of the model) to discover the optimal set of values for the weights that minimizes our cost function. Stochastic Gradient Descent (formally) will update the weights with every training input. In our case, we implemented mini-batch gradient descent.

Mini-Batch Gradient Descent updates the weights after a specified number of inputs have been processed, termed as the batch size. This speeds up the model, as SGD is very slow due to weights updating for every single training input.

1. Mini-Batch Gradient Descent with momentum term

Gradient Descent aims to find an optimal set of weights that allows us to achieve the global/local minima for the cost function. However, these minimum points are often associated with steep gradient drops. Vanilla SGD tends to oscillate across the slopes of these ‘ravines’ and take much longer to reach the minima. Therefore, the model will take a longer time to approach convergence at minima.

Momentum terms nudge the gradient towards the ravine. Essentially, the momentum term accelerates the gradient in the direction that previous gradients also imply. If the history of the pastgradients has vectors that point in a certain direction, then there is momentum in that direction, and the momentum terms accounts for this direction, adding the momentum to the current gradient. Thus, if the direction of past gradients points toward the minima, the gradient will be accelerated towards the minima instead of taking small steps and bouncing across the ravine.

More formally,

|  |  |  |  |
| --- | --- | --- | --- |
| γ momentum term | θ optimization parameter | η learning rate | vt update vector at time t |
| vt=γvt−1+η∇θJ(θ) the momentum from update vector at previous time (t-1) carries into the update vector at time t, and joins with actual gradient of loss function with respect to optimization parameter to form new update vector | | | |

We expect that momentum terms will help navigate gradients better and converge faster and more accurately.

1. RMSProp algorithm

Complex neural networks like that pass gradients through multiple layers in back propagation might face vanishing gradients if the gradients are small and undergo repeated matrix multiplications with other small gradients, effectively causing them to go to zero. The inverse is true for gradient explosions, where large gradients keep getting bigger.

RMSProp normalizes gradients by using a running average of the magnitudes of recent gradients. Therefore, it places restrictions on how big and how small the gradients can end up being, which helps avoid exploding gradients and vanishing gradients. This keeps the model weights under control, and training at a reasonable pace.

1. Adam optimizer

The Adam optimizer differs from SGD where learning rates are concerned. In SGD, a single learning rate is applied to the model. Adam uses adaptive moments to calculate a suitable learning rate for individual parameters, thus learning rates do not remain the same throughout training, and it is adapted by calculating exponential moving average of the gradient and squared gradients (controlled by two parameters β1 and β2). Adam optimizer shows good performance in many tasks.

1. Dropout

Neural networks function using layers of neurons. Dropout is a regularization technique that will randomly turn off some of the neurons in a neural network. Turning off a neuron simply renders its synaptic output invalid, and therefore the weights that are learned by neurons that have been turned off do not propagate through the model. Dropout is usually used to prevent overfitting by randomly turning off certain weights, which might include very large weights learned in overfitting. It is expected that dropout will help models generalize better in cases of overfitting.



## Saving Data

The history of each model generated by the model.fit( ) method will be saved. Logging these outputs is necessary, to analyze them afterwards without having to keep the notebook instance running constantly. It will allow us to easily plot any graph that we want without having to run the model and go through the model training every time.

We save the history of the model, which includes loss and accuracy information, as a .json file, which we can easily retrieve through helper functions that have been defined.

# Question 1 – Create a specified CNN, visualize feature maps

### Experiments

We were given specifications for a CNN, and tasked to build this CNN, train it, and visualize the feature maps at each convolution and pooling layer. Also, the loss and accuracy should be plotted.

Notebook title: 2a/2a1.ipynb

### Results

**Q1a – cost and accuracy plots**

|  |  |
| --- | --- |
|  |  |

We can see that the model has achieved convergence for the test set.

**Q1b – feature maps**

1. Convolutional layer C1, 50 channels
2. Maxpool layer S1
3. Convolutional layer C2, 60 channels
4. Maxpool layer S2

Therefore, we expect 50 feature maps for the first pair of layers, and 60 feature maps for the second pair of layers. We will plot the original image, and the feature maps for the first two test images. C1, S1 will be a grid of 5x10 images, each grid square representing a feature map, and C2, S2 will be a grid of 6x10 images, each grid square representing a feature map as well.

We can observe some edge detection from the C1 and S1, but nothing too significant or obvious. C2 and S2 simply look like noise, or rather the features are not discernable to us.

|  |  |  |
| --- | --- | --- |
| Original | Conv1 | Pool1 |

**x\_test image 0**

|  |  |
| --- | --- |
|  | |
|  |  |
|  |  |

**x\_test image 1**

|  |  |
| --- | --- |
|  | |
|  |  |
|  |  |

# Question 2 – Hyperparameter sweep

### Experiments

Question 1 involved us using 50 channels for C1, S1 and 60 channels for C2, S2. Now, we will do a hyperparameter sweep over the following: C1, S1 – 10, 30, 50, 70, 90 C2, S2 – 20, 40, 60, 80 ,100

We will use test accuracy to determine the optimal combination.

Notebook title: 2a/2a2.ipynb

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 𝛼 = 0.01, L2 regularization with weight decay parameter 𝛽 = 10−6 , and batch size = 32. Use appropriate scaling of input features.* 
   1. *Use the training dataset to train the model and plot accuracies on training and testing data against training epochs.*
   2. *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*

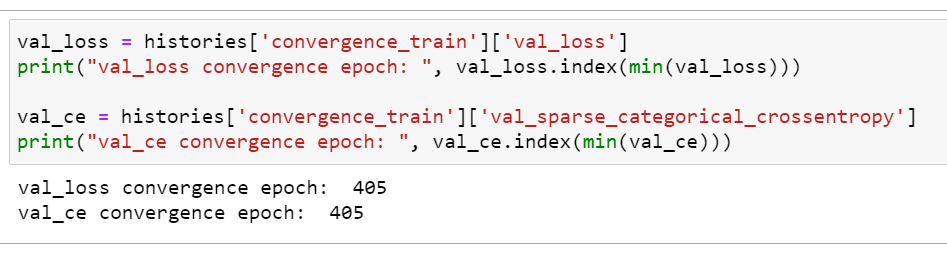
*# try on test set*

### 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.



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.

1. *Find the optimal batch size by training the neural network and evaluating the performances for different batch sizes.* 
   1. *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.*
   2. *Select the optimal batch size and state reasons for your selection.*
   3. *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.

1. *Find the optimal number of hidden neurons for the 3-layer network designed in part (2).* 
   1. *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}.*
   2. *Select the optimal number of neurons for the hidden layer. State the rationale for your selection.*
   3. *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.

1. *Find the optimal decay parameter for the 3-layer network designed with optimal hidden neurons in part (3).* 
   1. *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}.*
   2. *Select the optimal decay parameter. State the rationale for your selection.*
   3. *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.

1. *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 .* 
   1. *Plot the train and test accuracy of the 4-layer network.*
   2. *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

1. Optimal normalization method
2. Number of epochs required for convergence
3. Most redundant feature through Recursive Feature Elimination
4. 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 than 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.



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**

|  |  |
| --- | --- |
|  | This is a zoomed in version |

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, 𝐿2 regularization at weight decay parameter 𝛽 = 10−3 and a learning rate 𝛼 = 10−3 to train the network.* 
   1. *Use the train dataset to train the model and plot both the train and test errors against epochs.*
   2. *State the approximate number of epochs where the test error is minimum and use it to stop training.*
   3. *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

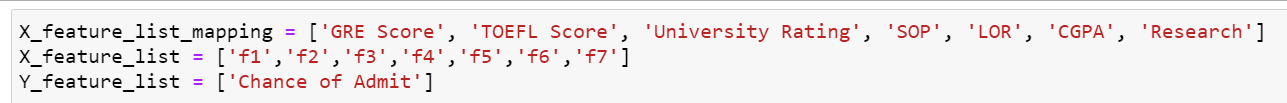
We can use 400 epochs to achieve convergence on the loss of our model.

# 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.



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, 𝐿2 regularization at weight decay parameter 𝛽 = 10−3 and a learning rate 𝛼 = 10−3 to train the network.* 
   1. *Use the train dataset to train the model and plot both the train and test errors against epochs.*
   2. *State the approximate number of epochs where the test error is minimum and use it to stop training.*
   3. *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**

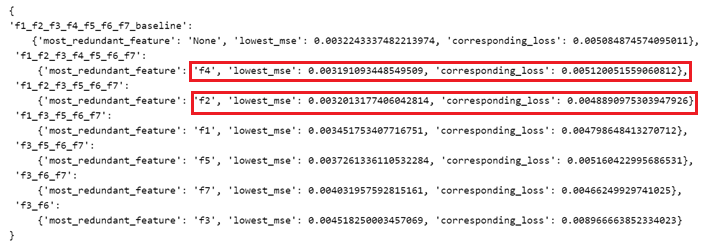
|  |  |
| --- | --- |
|  | Zoom in on the last 15 epochs |
|  | Zoom in on the last 15 epochs |

How to read:

f1\_f2\_f3\_f4\_f5\_f6\_f6\_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).



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.

1. *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**

|  |  |  |
| --- | --- | --- |
|  | Zoom in first 50 epochs | Zoom in last 50 epochs |
|  | Zoom in first 50 epochs | Zoom in last 50 epochs |

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 >>>**