CS6375 Assignment 1
https://github.com/caf3676/Homework-1

Carlos Figueroa-Diaz
Caf190007

1 Introduction and Data (5pt)

This project revolves around using a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN) to make predictions of yelp ratings based on a set of Yelp reviews. These Yelp reviews are split between training, testing, and validation sets and serve different functions. The training set is used to adjust weights through backpropagation, the validation set aids in hyperparameter tuning, and the test set evaluates final performance of the model. The reviews contain a lengthy description of a business followed by a numerical score between 1 and 5. The RNN also contains a pickle file containing word embeddings used to represent input vectors for the RNN. After running the models on their default settings with different numbers of hidden layers I determined that a 32 hidden layer model worked best for the FFNN with a training accuracy of 0.757625 and a validation accuracy of 0.6025. As for the RNN, the 32 hidden layer model worked best with a validation accuracy of 0.41 and a training accuracy of 0.43975. Upon reducing the learning rate to 0.001 the results for the FFNN improved on the 32 hidden layer model resulting in training and validation accuracies of 0.62 and 0.72625. However, this change did not reflect well on the RNN as it led to overfitting after one epoch.

2 Implementations (45pt)

2.1 FFNN (20pt)

My implementation of forward() FFNN.py is as follows:

```
def forward(self, input_vector):
    # [to fill] obtain first hidden layer representation
    hiddenLayer = self.activation(self.W1(input_vector))

    # [to fill] obtain output layer representation
    outputLayer = self.W2(hiddenLayer)

    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(outputLayer)

    return predicted_vector
```

The first hidden layer representation is computed by applying weight matrix W1 to input_vector and passing it through the ReLU() activation function. The output layer representation is computed by applying the W2 weight matrix to the hidden layer. Finally, the softmax function is applied to the output layer to obtain a probability distribution over the five possible output classes. The function then returns predicted_variable, which

contains log probabilities of each class from LogSoftmax(). PyTorch was used for defining and training the neural network, NumPy was used for general numerical computations tqdm provides progress bars for tracking training programs, json was used for reading and handling dataset files, and argparse was used for parsing command-line argument.

It is important to understand how each part of the code works in relation to forward(). The model contains an optimizer that uses stochastic gradient descent (SGD) with a learning rate of 0.01 and momentum of 0.9. The model parameters are updated using backpropagation via optimizer.step(). For the loss function, NLLLoss() (negative log-likelihood loss) is used since the model outputs log probabilities via the LogSoftmax function. The data processing revolves around vocabulary built from training data, with <UNK> handling unknown words. The model input vectors are formed by mapping each word to an index and converting them into a sparse vector. Training is composed of several steps, shuffling the data before training to improve generalization, mini-batch training small subsets of data is sizes of 16, loss averaging over batches to improve stability, and evaluating model accuracy after each epoch. The model trains for a fixed number of epochs as specified by the command line arguments.

2.2 RNN (25pt)

My implementation of forward() RNN.py is as follows:

```
def forward(self, inputs):
    # obtain first hidden layer representation
    _, hidden = self.rnn(inputs)

    # extract last hidden layer state
    hidden = hidden[-1]

    # obtain output layer representation
    output = self.W(hidden)

    # obtain probability dist.
    predicted_vector = self.softmax(output)

    return predicted_vector
```

The forward() function works by passing the inputs through the self.rnn module. The hidden state output of the RNN is extracted. Since there is only one hidden layer, the final hidden state is accessed using hidden[-1]. The hidden state is passed through a fully connected linear layer to transform it into the output dimension, or the number of classes, which is 5 in this case. The output of the linear layer is passed through a LogSoftmax(dim = 1) to produce a probability distribution over the classes. PyTorch was used for defining and training the neural network, NumPy was used for general numerical computations tqdm provides progress bars for tracking training programs, json was used for reading and handling dataset files, argparse was used for parsing command-line argument, and pickle was used to read a pretrained word embedding dictionary.

The RNN implementation has several differences from the FFNN implementation. Unlike FFNN's which process inputs in a single forward pass, RNNs maintain a hidden state across time steps, allowing them to process sequential data. In the FFNN, each layer has separate weights for each transformation, while RNNs use the same weights for each time step. By virtue of having a hidden state, RNNs capture past information, while FNNs do not retain information from previous inputs. Data vectorization differs with the RNN since it uses words mapped to embeddings via word_embedding.pkl. However, like the FFNN, unknown words are replaced with the <UNK> token. Training is done using stochastic gradient descent with an Adam optimizer, mini batches of size 16, a negative log-likelihood loss function. The training sequence stops when validation accuracy decreases while training accuracy increases, preventing overfitting. As such, if the stopping condition is met, training terminates early even if the total number of epochs has not been reached.

3 Experiments and Results (45pt)

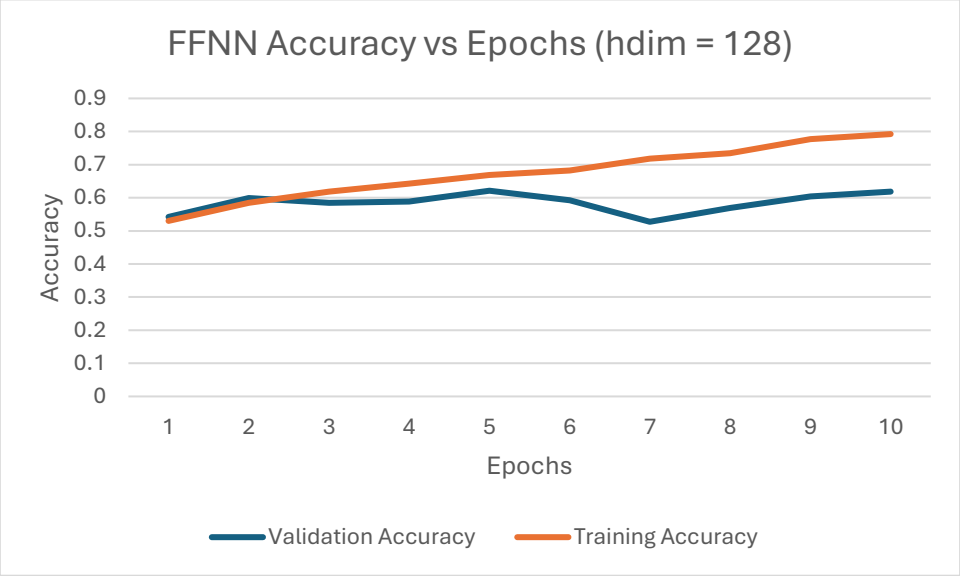
The models are evaluated using classification accuracy:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

For the FFNN, during both training and validation the model predicts a label by applying a softmax activation over the output layer. The predicted class is determined by using torch.argmax(predicted_vector), which selects the class with the highest probability. The number of correct predictions is counted iteratively through the dataset. The RNN differs with its hidden layer but still uses a softmax layer to convert outputs to probabilities. The predicted class is obtained using torch.argmax(output) and the accuracy is computed the same way as the FFNN.

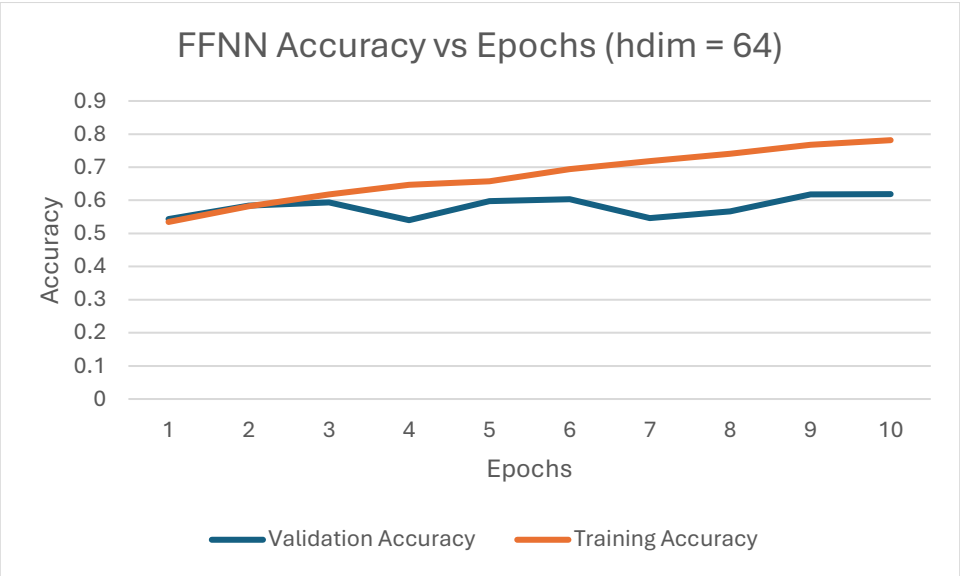
For the FFNN, I compared the results between 128 hidden layers, 64 layers, and 32 layers for 10 epochs with the default learning rate of 0.01. The training accuracy and validation accuracy by epoch have been tabulated and plotted below:

128 Hidden Layers



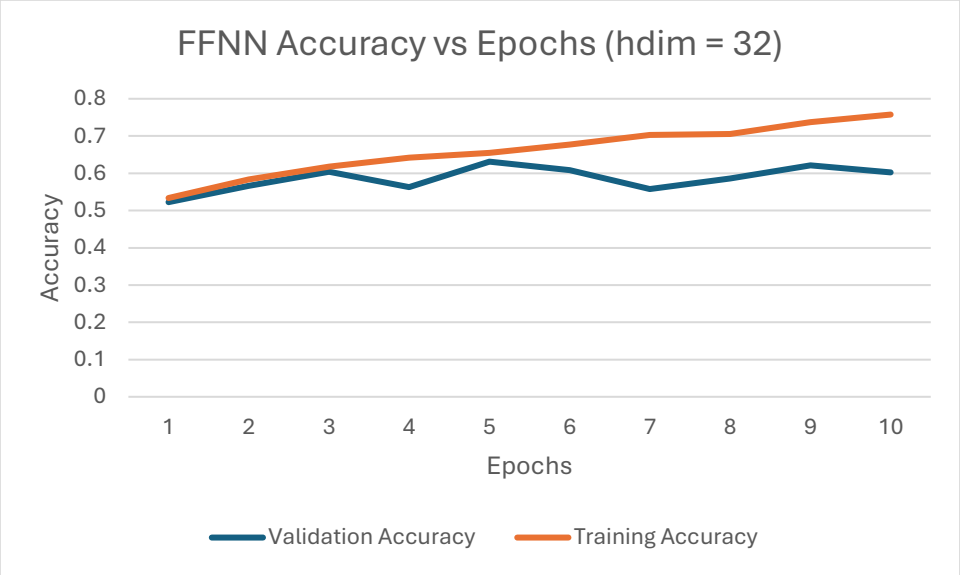
Epoch	Validation Accuracy	Training Accuracy
1	0.54375	0.534625
2	0.58375	0.581875
3	0.59375	0.618125
4	0.54	0.647125
5	0.5975	0.65775
6	0.60375	0.6945
7	0.54625	0.718875
8	0.56625	0.741
9	0.6175	0.767625
10	0.61875	0.781625

64 Hidden Layers



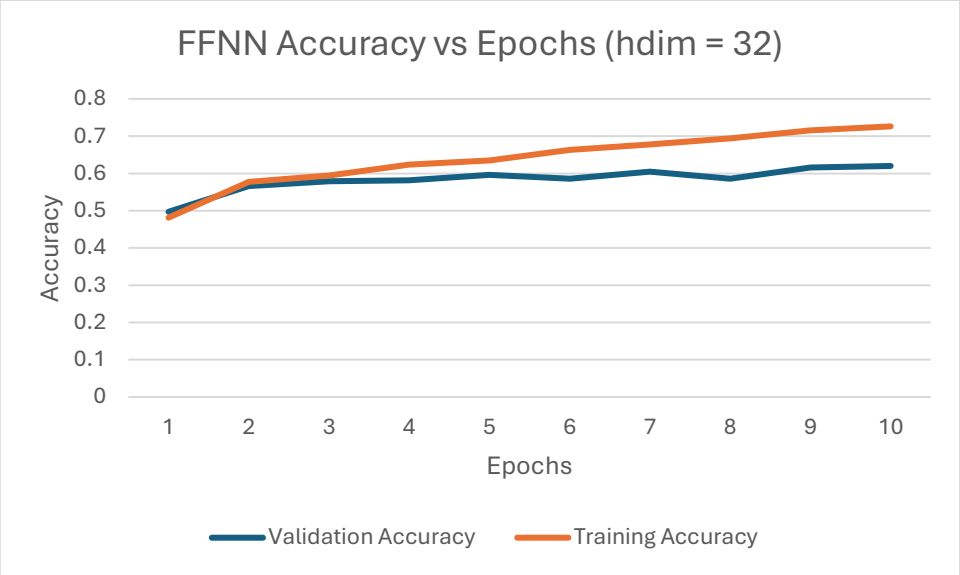
Epoch	Validation Accuracy	Training Accuracy
1	0.54375	0.534625
2	0.58375	0.581875
3	0.59375	0.618125
4	0.54	0.647125
5	0.5975	0.65775
6	0.60375	0.6945
7	0.54625	0.718875
8	0.56625	0.741
9	0.6175	0.767625
10	0.61875	0.781625

32 Hidden Layers



Epoch	Validation Accuracy	Training Accuracy
1	0.5225	0.533625
2	0.56625	0.58375
3	0.60375	0.61775
4	0.5625	0.642
5	0.63125	0.654875
6	0.60875	0.677125
7	0.5575	0.703
8	0.58625	0.705375
9	0.62125	0.737
10	0.6025	0.757625

Each model had steadily increasing training accuracies, but the gap between the validation accuracy and the training accuracy indicated signs of overfitting. To combat this, I tested reducing the learning rate to 0.001 on the most stable model, the 32 hidden layer model. The results are as follows:

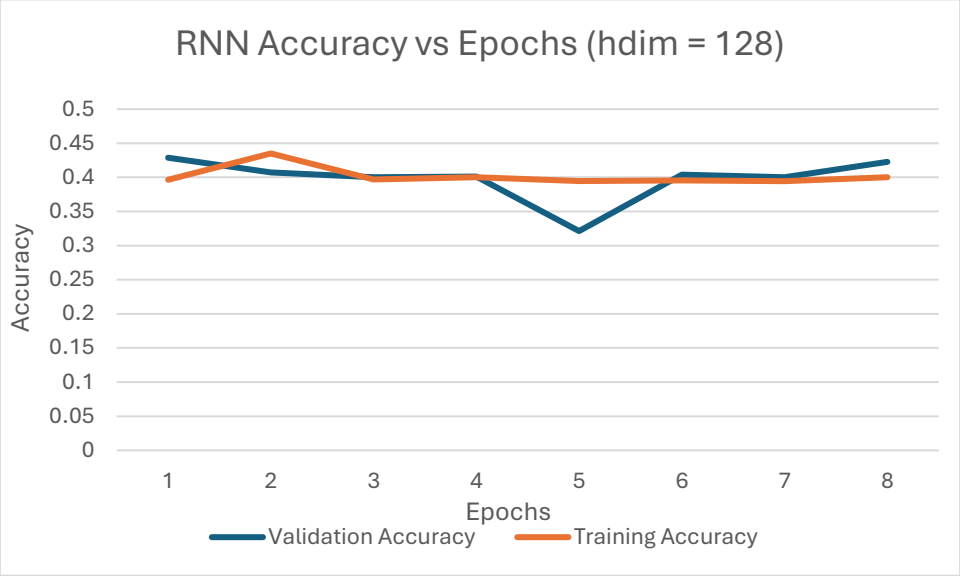


Epoch	Validation Accuracy	Training Accuracy
1	0.497	0.48125
2	0.56575	0.5775
3	0.57875	0.5945
4	0.58125	0.623625
5	0.59625	0.634875
6	0.58625	0.663
7	0.605	0.67775
8	0.58625	0.694125
9	0.61625	0.71575
10	0.62	0.72625

This FFNN model was more stable and improved validation accuracy compared to the previous overfitting cases. Both training and validation accuracy increased but the gap between the values was smaller, indicating gradual learning without excessive overfitting.

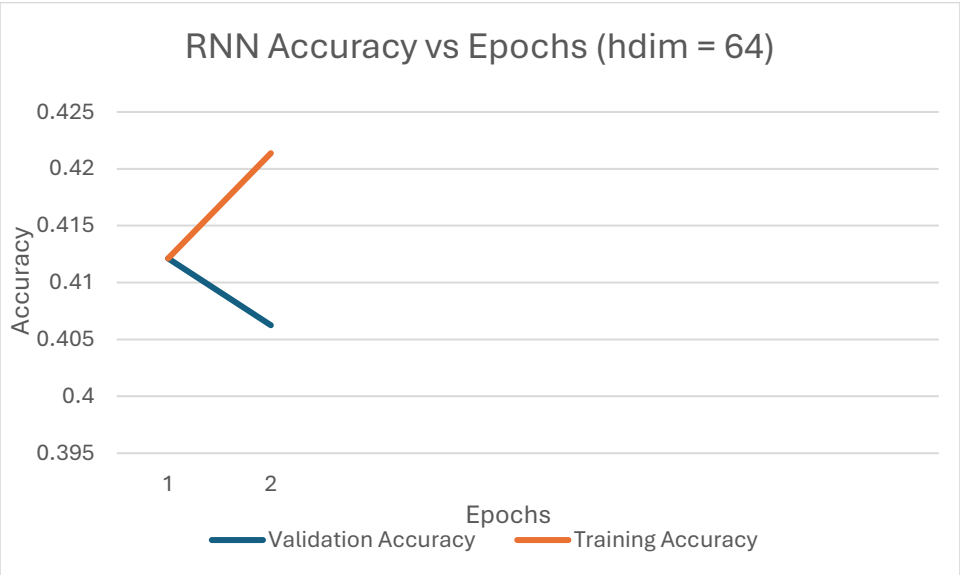
The RNN model was run with the default parameters and hidden dimension layers of 128, 64, and 32 respectively. The results are tabulated and plotted below:

128 Hidden Layers



Epoch	Validation Accuracy	Training Accuracy
1	0.42875	0.39625
2	0.407125	0.435
3	0.4	0.397
4	0.40125	0.400375
5	0.32125	0.394625
6	0.40375	0.395625
7	0.40025	0.394375
8	0.4225	0.4

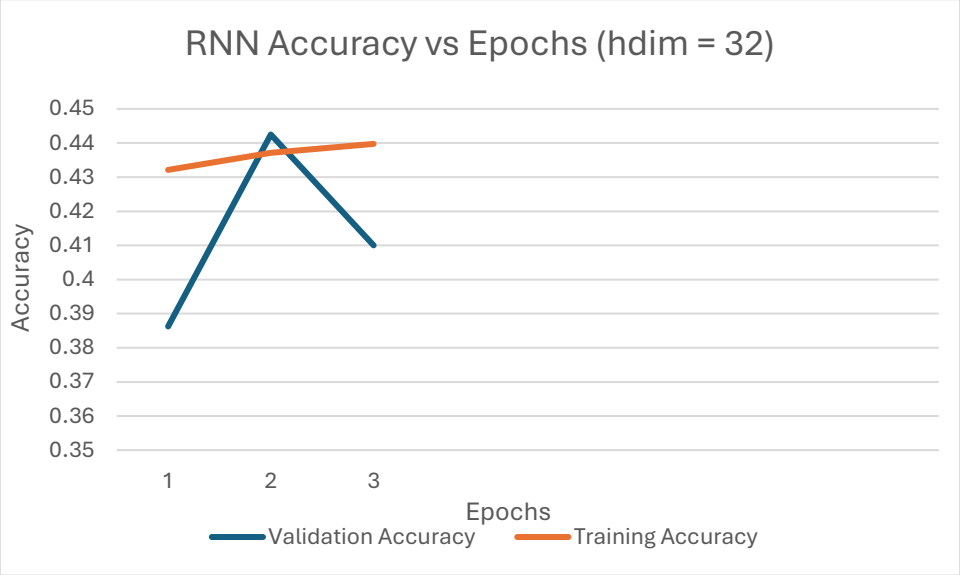
64 Hidden Layers



Epoch	Validation Accuracy	Training Accuracy

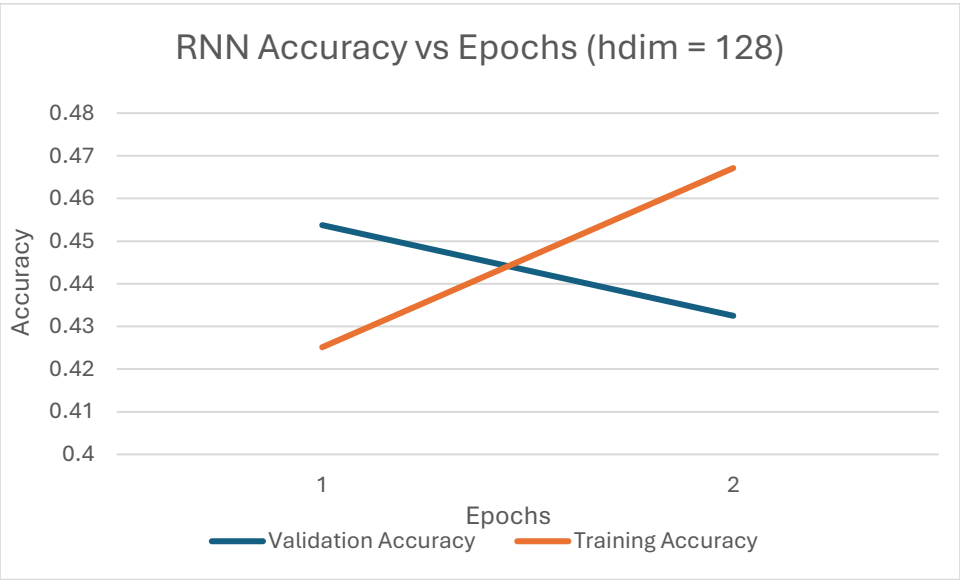
1	0.412125	0.412125
2	0.40625	0.421375

32 Hidden Layers



Epoch	Validation Accuracy	Training Accuracy
1	0.38625	0.432125
2	0.4425	0.437125
3	0.41	0.43975

Of all the models, the 128 hyperparameter model had the most epochs, but the validation and training accuracies oscillated suggesting overfitting. The 64- and 32-layer models had better training accuracies, but the model stopped much earlier to prevent overfitting. Out of curiosity, I tested the 128 hyperparameter model with a learning rate of 0.001 to see if it could fix the oscillation problems it had with the default settings. The data is as follows:



Epoch	Validation Accuracy	Training Accuracy
1	0.45375	0.425125
2	0.4325	0.467125

After one epoch the model detected overfitting. It seems that decreasing the learning rate made the model less effective. In the future I would probably change the implementation of my model or use more hyperparameters to see if it makes a noticeable difference.

4 Conclusion and Others (5pt)

I was responsible for the entire project. The project took about 6 hours of work spread over the course of 3 days. Most of the work was spent writing the report, running the models, and formatting the data. Implementing the model was not too difficult although my unfamiliarity with neural networks required some research on the topic so that I could make sense of what the provided code was doing. Also, I was confused about the provided assignment documentation as there are two sets of testing, training, and validation datasets, and I am worried I might have used to wrong one as I am not sure if the correct set was specified. If this was specified, my apologies, but if not, I would suggest not having files of the same name in different folders in the future to reduce confusion.