CS 6375 Assignment 1

Austin Polk, arp210003

March 16, 2025

Github link: github.com/AustinPolk/CS6375-Assignment1

1 Introduction and Data

This project centers around the task of sentiment analysis. Specifically, two different models are used to predict the number of stars left on a review given the text left with the review. A Feed Forward model and a Recurrent model were used and compared with each other based on the metrics of average loss and accuracy. Multiple variations of each model were also trained and evaluated to see how they scale with an increasing number of parameters.

The training, validation, and test dataset consist of pairs of review text and stars left with the review. The number of pairs for each dataset are listed below:

Dataset	Pairs
Training	8000
Validation	800
Test	800

2 Implementations

2.1 Feed Forward Neural Network

```
class FFNN(nn.Module):
    def __init__(self, input_dim, h):
        super(FFNN, self).__init__()
        self.h = h
        self.W1 = nn.Linear(input_dim, h)
        self.activation = nn.ReLU()
        self.output_dim = 5
        self.W2 = nn.Linear(h, self.output_dim)

        self.softmax = nn.LogSoftmax(dim=0)
        self.loss = nn.NLLLoss()

    def compute_Loss(self, predicted_vector, gold_label):
        return self.loss(predicted_vector, gold_label)

    def forward(self, input_vector):
        x = self.W1(input_vector)
        x = self.activation(x)
        x = self.W2(x)
        x = self.activation(x)
        return self.softmax(x)
```

Above is the source code for my implementation of the **Feed Forward Neural Network**. This is a simple fully-connected neural network with one hidden layer, using the **ReLU** activation function for nonlinearity.

The input to the network is a combination of one-hot encodings for each of the words in the review text. As such, the dimension of the input layer is the size of the vocabulary. The next layer is the hidden layer, which has a variable length, specified using command line arguments to the **Python** script. The final layer is the output layer with 5 outputs, each corresponding to a rating in stars for that review.

The source code itself is fairly straightforward, thanks to the **PyTorch** library. In the **forward()** function, the input tensor is taken as the input to the first linear layer, and then the activation function is applied to get the result for that layer. This is then given to the next layer where the same thing happens. Then, the output of this layer is given to the **Softmax** function to convert it into a (logarithmic) probability distribution. This is analogous to a probability that the given input belongs to each of the output classes.

2.2 Recurrent Neural Network

```
class RNN(nn.Module):
    def __init__(self, input_dim, h):
        super(RNN, self).__init__()
        self.h = h
        self.numOfLayer = 1
        self.rnn = nn.LSTM(input_dim, h, self.numOfLayer)
        self.softmax = nn.LogSoftmax(dim=0)
        self.W = FFNN(h, 32)
        self.loss = nn.NLLLoss()

    def compute_Loss(self, predicted_vector, gold_label):
        return self.loss(predicted_vector, gold_label)

    def forward(self, inputs):
        output, _ = self.rnn(inputs)
        x = output[:, -1, :]
        return self.W(x[-1])
```

Above is the source code for my implementation of the **Recurrent Neural Network**. It consists of two components, the **LSTM** and fully-connected layers. The **RNN** used in the initial implementation was replaced with an **LSTM** to leverage a lower training time and ease of use. I also took advantage of the fact that I had already developed the **FFNN** and used it as the network that interpreted the output of the recurrent units, with a fixed hidden layer size of 32.

The input is a sequence of vector embeddings representing each word in the review text. These are fed to the **LSTM**, which processes each vector one at a time and updates its hidden states accordingly to produce a new encoded sequence. The final element in this sequence is the most important, as it has been encoded with information from the entire sequence. Taking this final tensor and passing it to the fully-connected layers, the final output is a logarithmic probability distribution of how likely the text is to belong to each class.

The source code is again fairly straightforward, especially since it leverages the existing **FFNN**. The input sequence of vectors is passed to the **LSTM**, which automatically computes all of its hidden states and returns them in the *output* variable. Only the last of these states is needed, so we trim it to only that one before passing it to the **FFNN** and returning that as the result.

2.3 Training

Both models were trained using **Stochastic Gradient Descent**, which I found to provide a good rate of convergence and ran slightly faster than the **Adam** optimizer for my purposes. Each model was fed training data in batches and updated upon the completion of each batch. After all batches had been run, the current state of the model was checked against the validation dataset.

I modified the training loops to continuously train for a set number of epochs, as opposed to training until some other condition was met. This allowed me to see longer-term trends in the training process, and it also allowed me to see how well the models could learn the datasets when given a larger amount of time.

3 Experiments and Results

3.1 Evaluations

The models were each evaluated according to:

1. **Average Accuracy**

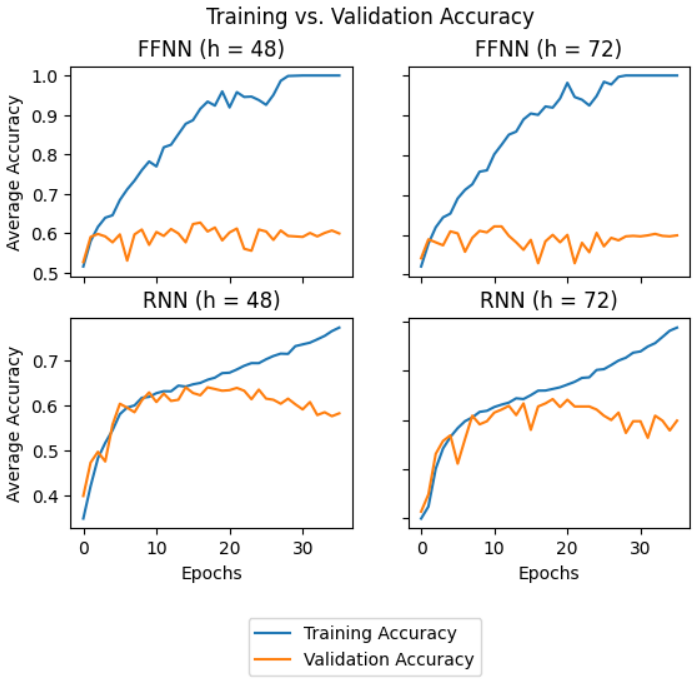
   The number of correct classifications made by the model during a given epoch vs the total number of predictions made. This is expected to increase as the model trains further, however it can fluctuate during training and decrease. It should get closer to 1 as training progresses, but it must be monitored to prevent overfitting.

2. **Average Loss**

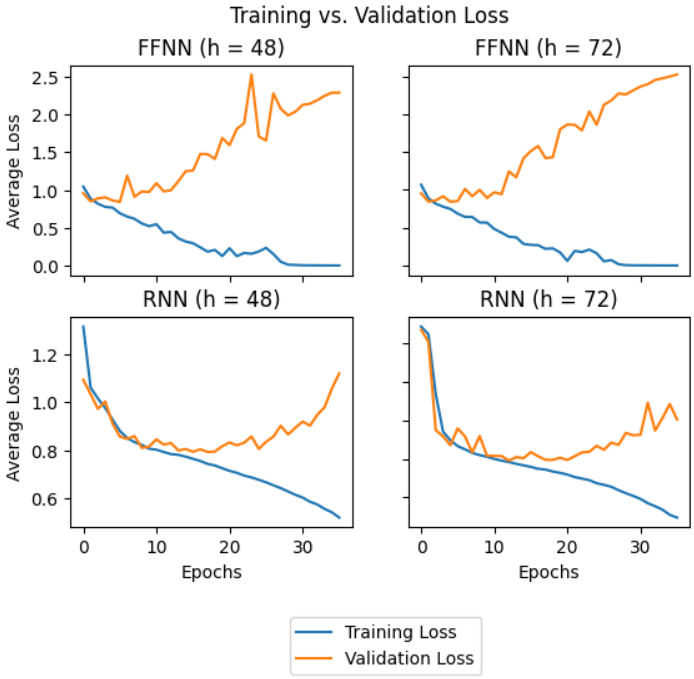
   The average loss of the model during a given epoch, as computed using the negative log likelihood loss function. This should decrease from epoch to epoch, as this is the metric used by the optimizer to train the model.

3.2 Results

The average accuracy during the training process can be seen in the following image:



The average loss during the training process can also be seen in the following image:



From the above graphics, a few observations can be made.

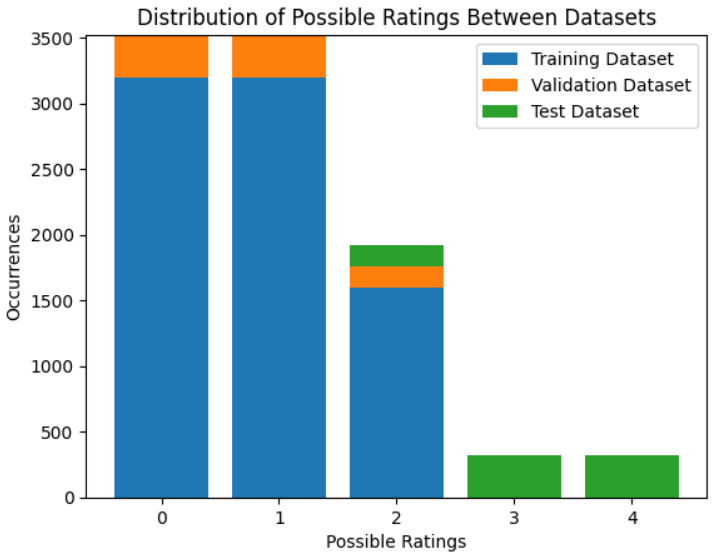
- While the training accuracy increases for the **FFNN** over time, its validation accuracy hovers around 60% for the duration of the training. This is in contrast to the **RNN**, whose training and validation accuracies rise in unison until it peaks at just above 60%.

- The validation loss for the **FFNN** actually increases steadily over time, even as the training loss decreases. This could be due to overfitting, or a poorly chosen architecture for the task. The **RNN**, on the other hand, has them both decreasing until a point at which it likely begins overfitting the training data.
- Although the hidden dimension size of each network may impact the overall accuracy or loss characteristics, the shape of the curves produced by them over time has more to do with the architecture of the network.
- Neither network could achieve much better than 60% accuracy during the training process, but that is not necessarily a hard limit. More training data for each output class can likely improve the final performance.

3.3 Performance on Test Dataset

I had some trouble running either model on the test dataset, which is meant to show how the model responds to out of sample data. The accuracy kept hovering near zero, and the loss was very high. However, upon analyzing the distribution of output classes in each dataset, I found something interesting.

The following graphic shows the distribution of training and testing pairs within all three of the datasets used in this assignment:



The training and validation data were heavily skewed towards representing the output classes 0, 1, and 2, while the test data only represented classes 2, 3, and 4. I believe this skewness is what led to such poor performance of both models on this dataset. Their training data did not equip them with the skills to accurately classify this data. For this reason, I have chosen not to include graphics relating to performance on this dataset.

4 Conclusion

This assignment made me more familiar with popular machine learning libraries and allowed me to expand my knowledge of machine learning. I had not experimented with **PyTorch** much before this, but it's good to gain experience with tools that are in heavy use in the field.

If I had to give any feedback on the assignment, it would say that it was nice to have a framework to start with, but it did lock me into trying to do things a certain way. In the end I had to make changes to the code outside of just the **forward()** function to get the best results. It was not too difficult overall, but I did spend a lot of time trying to get the best performance out of my models, especially in training.