Homework 2
CS 6320: Natural Language Processing

Group – 22
Sahana Vinayak (sxv170330)
Sanatan Shrivastava (sxs220062)

November 11, 2023

Github Link: https://github.com/Sanatan-Shrivastava/CS6320-NLP-G22

1. Introduction and Data (5pt)

The project's main goal is to use neural network models — more especially, a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN) to do sentiment analysis. Predicting Yelp reviews' sentiment (rating), which is divided into five classes (1 to 5), is the aim. The dataset we are using is of Yelp reviews containing the training, validation, and test sets provided by the instructor. The number of examples in the training, validation, and test set are as follows:

Set	Number of Examples	Number of labels (from given JSON files)
Training	8000	{1.0: 3200, 2.0: 3200, 3.0: 1600, 4.0: 0, 5.0: 0}
Validation	800	{1.0: 320, 2.0: 320, 3.0: 160, 4.0: 0, 5.0: 0}
Test	800	{1.0: 0, 2.0: 0, 3.0: 160, 4.0: 320, 5.0: 320}

Table 1: Statistics related to the provided Dataset – 'Yelp Reviews'

The main task(s) in the project involves using PyTorch to implement the FFNN and RNN models, with an emphasis on finishing each model's forward computing phase. To get a probability distribution over the five classes, the FFNN performs a feedforward pass using a fixed-length vector as input. In contrast, each input vector is processed individually by the RNN, which then computes updated representations for every location before adding together all of the vectors for every token in the output layer to produce a representation for the entire sequence.

In the case of RNN, Casing involves all input words to lowercase before looking at the word_embeddings. Tokenization is being performed by splitting words using spaces as delimiters. Another step is punctuation removal. All of these steps are discussed in RNN – 2.2

In the case of FFNN, There is no explicit casing involved. The case of the words is considered as is. The code is being tokenized by splitting words using whitespaces. <unk> token is being used to represent Out Of Vocabulary words.

2. Implementations (45pt)

2.1 FFNN (20pt)

Firstly, we perform the data pre-processing steps on the data. The input text data is tokenized by the code into discrete words or tokens. It tokenizes the input into words by splitting it on whitespace using the split() method. This is carried out by splitting the incoming text into words in the load_data function:

```
tra.append(int(elt["stars"] - 1), (elt["text"].split())).
```

No explicit casing normalization is carried out by the code. Using the training data, we create a vocabulary of distinct words and add a token to stand in for words that are not in the vocabulary. When converting the text data to vector representations, every word in the data is added to the vocabulary; if a word is not in the vocabulary, it is represented as **<unk>**.

The completed code of FFNN function is as follows:

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

    # [to fill] obtain output layer representation
    x = self.activation(x)
    x = self.W2(x)

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

    return predicted_vector
```

The implementation of the complete code is as follows:

a. We start by applying the first linear transformation of the input vector w1. This step projects the data into the hidden layer.
b. Next, we apply the activation function to the output of the first layer. In this case, we are replacing the negative values with zero to introduce the non-linear characteristics using a ReLU function.
c. Similar to step (a), we again apply linear transformation using the weight vector W2 to further the transformation on the hidden layer and prepare it for classification.
d. Lastly, we apply the softmax function to the output of the second linear layer. This converts the raw output values into a probability distribution over the classes. The `predicted vector` is a vector of probabilities where each element represents the probability of input belonging to the specific class.

As per our understanding, other parts of the code are responsible for initializing the neural networks (__init__) method. It involves initializing the NN with input dimension, hidden layer of `h` dimension, activation function - ReLU, and Softmax function. Next, we have a loss method to compute the loss b/w gold label and predicted vector. We have a function to load data **which we have modified to run on test data**, and main function for all the pre-processing of all sets of data, taking input values of number of hidden layers and initializing the optimizers.

We have used PyTorch, NumPy, TQDM, JSON and argparse libraries to perform various operations in the code including preprocessing the data. We referred to the dataset on Yelp reviews available on https://www.yelp.com/dataset. We have also referred to the lecture slides and the recommended books for the course.

2.2) RNN (25pt)

Firstly, we perform the data pre-processing on the dataset. The input words are divided into a list of words after initially being joined into a single string. Then, we use.lower() to change every word to lowercase. When a word cannot be located in the word_embedding dictionary, the '<UNK>' (unknown) token is used in its place. Spaces are used as separators to divide the input_words variable into separate words. Before tokenization, punctuation is also eliminated from the text.

This is done using the code below:

```
 # Remove punctuation
input_words     =     input_words.translate(input_words.maketrans("",     "",
string.punctuation)).split()

# Look up word embedding dictionary
vectors = [word_embedding[i.lower()] if i.lower() in word_embedding.keys()
else word_embedding['unk'] for i in input_words]
```

The complete code of RNN is as follows:

```
def forward(self, inputs):
        hidden, _ = self.rnn(inputs)

        # [to fill] obtain output layer representations
        output = self.W(hidden)

        # [to fill] sum over output
        output_sum = torch.sum(output, dim=0)

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

        return predicted_vector
```

The implementation of the code is as follows:

a. We start by passing the input sequence to the RNN Module, this returns the hidden state output with hidden dimensions, etc. In the RNN layer, each step time produces an output, but we are only interested in the final hidden state.
b. Next, we apply the linear transformation on the hidden layer to convert it into output classes.
c. We are using pytorch library to sum the `output` tensor from the linear layer containing the information of each time step to get a single representation for the entire sequence.
d. Lastly, we apply the softmax function over the sum of the output tensor `output_sum` to convert the values into a probability distribution over different classes. At the end, we return the `predicted_vector` is the probability distribution over the classes that RNN predicts for a given input sequence.

As per our understanding, the other parts of the code are responsible for loading the data, and vectorizing it by removing punctuation and converting words to word embedding using a pre-trained word embedding `word_embedding.pkl`. It involves initializing the RNN module with an RNN layer, linear layer, and softmax activation function to take word vectors as input and output a probability distribution for 5 classes. We also utilize Adam optimizer to track training and validation accuracy to avoid overfitting.

We have a function to load data **which we have modified to run on test data**, and the main function for all the pre-processing of all sets of data and initializing the optimizers, and we removed the stopping condition to obtain the accuracy plots.

We have used PyTorch, NumPy, TQDM, JSON, and argparse libraries to perform various operations in the code including preprocessing the data. We referred to the dataset on Yelp reviews available at https://www.yelp.com/dataset. We have also referred to the lecture slides and the recommended books for the course.

3. Experiments and Results (25pt)

We use accuracy as the only metric for model evaluation in our present implementation. The percentage of accurate predictions made in a classification task relative to all guesses is known as accuracy. False Positives and False Negatives indicate inaccurate forecasts, whereas True Positives and True Negatives indicate accurate ones.

The formula to determine accuracy is (TP + TN) / (TP + TN + FP + FN) as shown in the figure below. Although accuracy gives a general idea of a model's performance, imbalanced datasets might not be a good fit for it. We have discussed this in the error analysis section where we observe that data is imbalanced.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Fig 1. Accuracy Formula

Source: https://www.evidentlyai.com/classification-metrics/accuracy-precision-recall

Summary for FFNN: We ran the FFNN model for four hidden dimensions (32, 64, 128, 256) and 10 epochs each. The best training, validation, and test accuracy are below. The model with hd = 128 has the best train accuracy while the model with hidden dimension = 32 has the best test accuracy.

Hidden dimensions	Best train accuracy	Best validation accuracy	Best test accuracy
32	75.76%	63.125%	12.375%
64	78.1625%	61.875%	**12.5%**
128	79.125%	62.125%	11.875%
256	80.1%	61.75%	13.5%

Table 2: FFNN summary

Summary for RNN: We ran the RNN model for four hidden dimensions (32, 64, 128, 256) and 10 epochs each. The best training, validation, and test accuracy are below. The model with hd = 64 has the best train accuracy while the model with hd = 32 has the best test accuracy.

Hidden dimensions	Best train accuracy	Best validation accuracy	Best test accuracy
32	48.71%	58.375%	**18.875%**
64	52.275%	56.375%	8.5%
128	49.7375%	52.125%	11.5%
256	46.8125%	50.875%	12.375%

Table 3: RNN summary

So from the summary above, we observe that the models are overfitting and the test accuracy is significantly low. The error analysis is covered in the ‘Analysis’ section below.

Apart from default hyperparameters, We have tried multiple variations of the FFNN and RNN with different hidden unit sizes (dimensions). The results are as follows:

<u>Small Hidden Unit Size:</u>

FFNN: 32 hidden dimensions

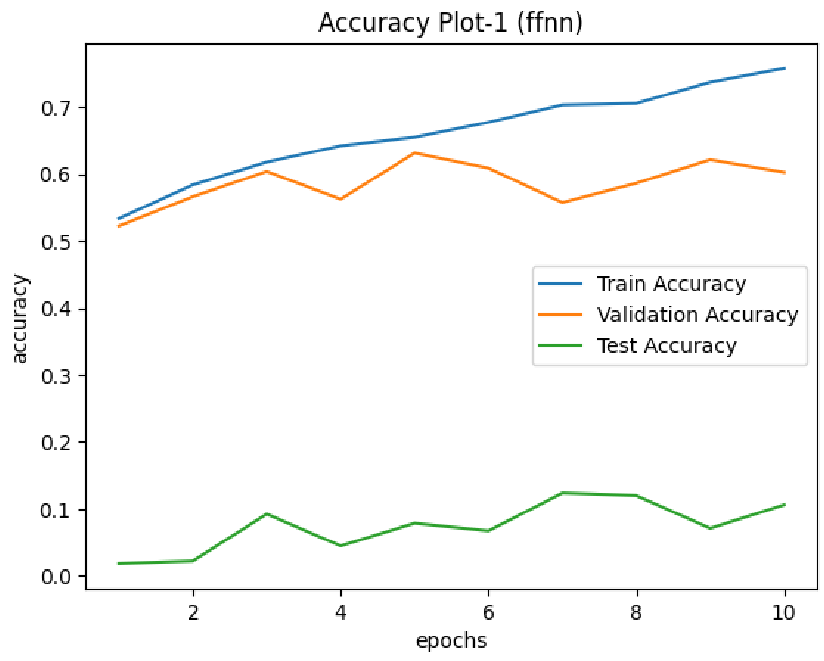

Fig 2. FFNN accuracy plot for model with hidden dimensions = 32

RNN: 32 hidden dimensions

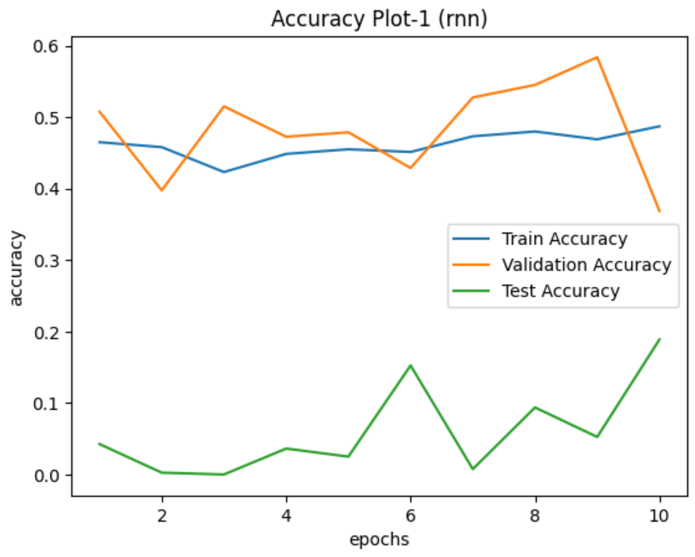

Fig 3. RNN accuracy plot for model with hidden dimensions = 32

Medium Hidden Unit Size:

FFNN: 64 hidden dimensions

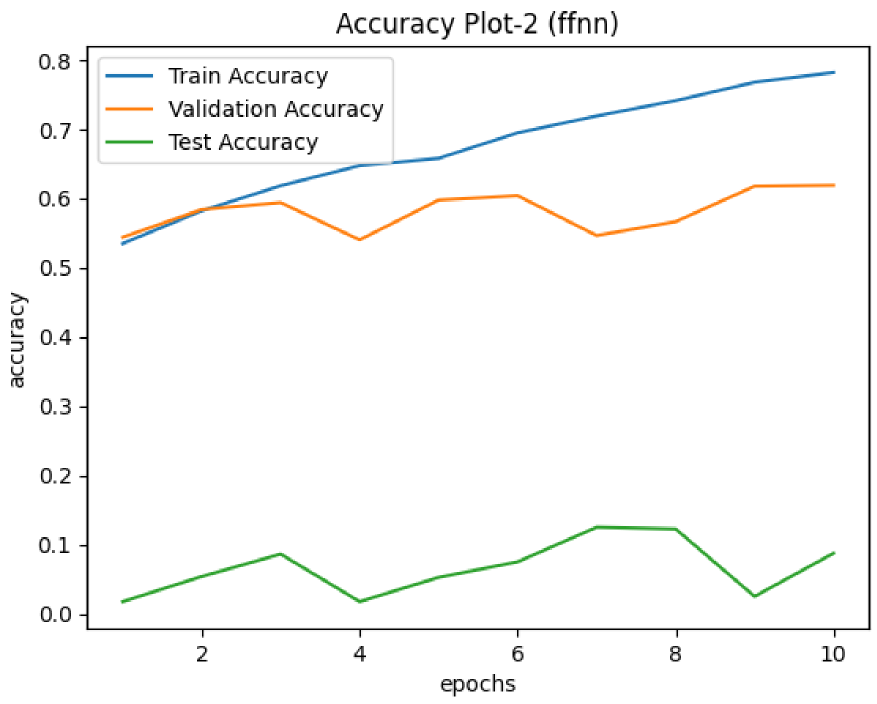

Fig 4. FFNN accuracy plot for model with hidden dimensions = 64

RNN: 64 hidden dimensions

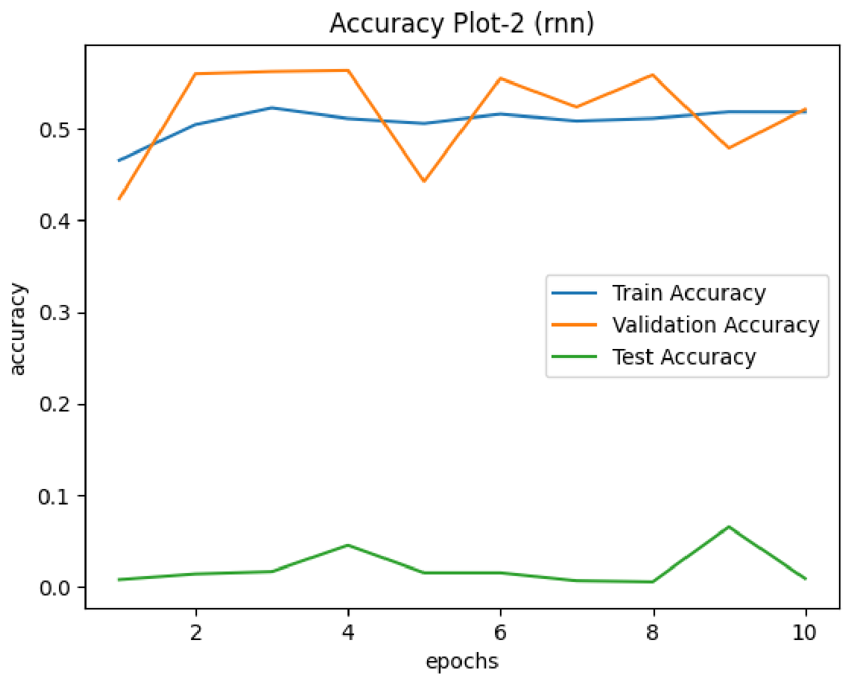

Fig 5. RNN accuracy plot for model with hidden dimensions = 64

Large Hidden Unit Size:

FFNN: 128 hidden dimensions

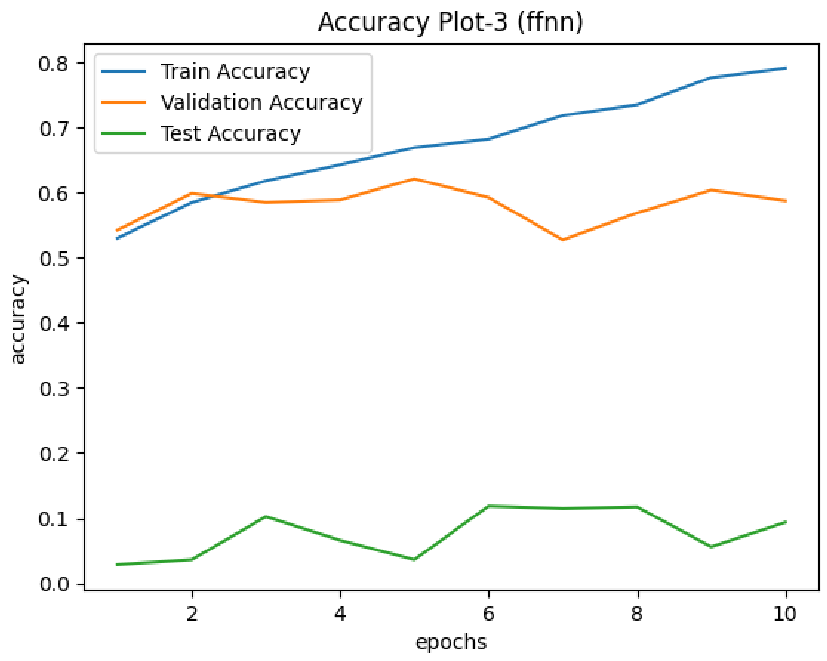

Fig 6. FFNN accuracy plot for model with hidden dimensions = 128

RNN: 128 hidden dimensions

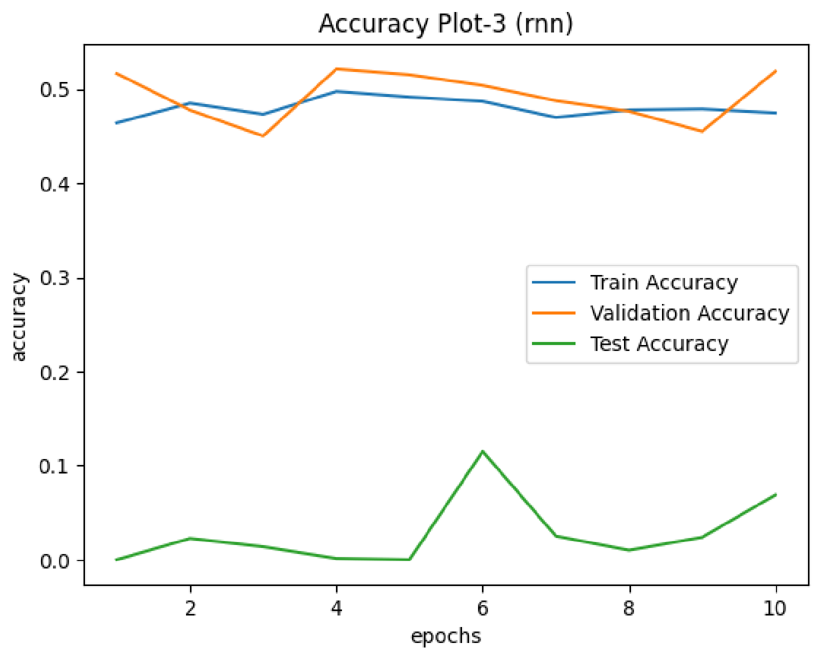

Fig 7. RNN accuracy plot for model with hidden dimensions = 128

<u>Very Large Hidden Unit Size:</u>

FFNN: 256 hidden dimensions

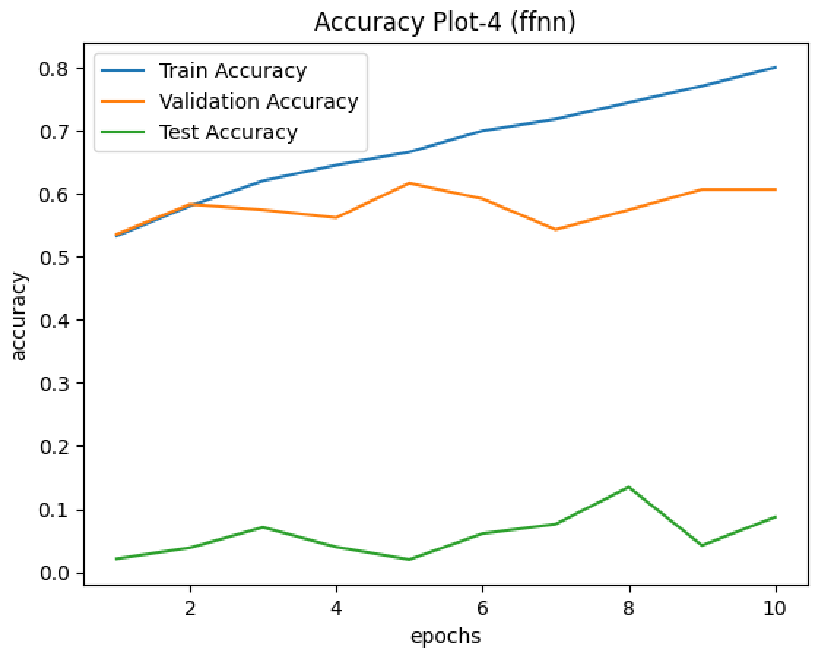

Fig 8. FFNN accuracy plot for model with hidden dimensions = 256

RNN: 256 hidden dimensions

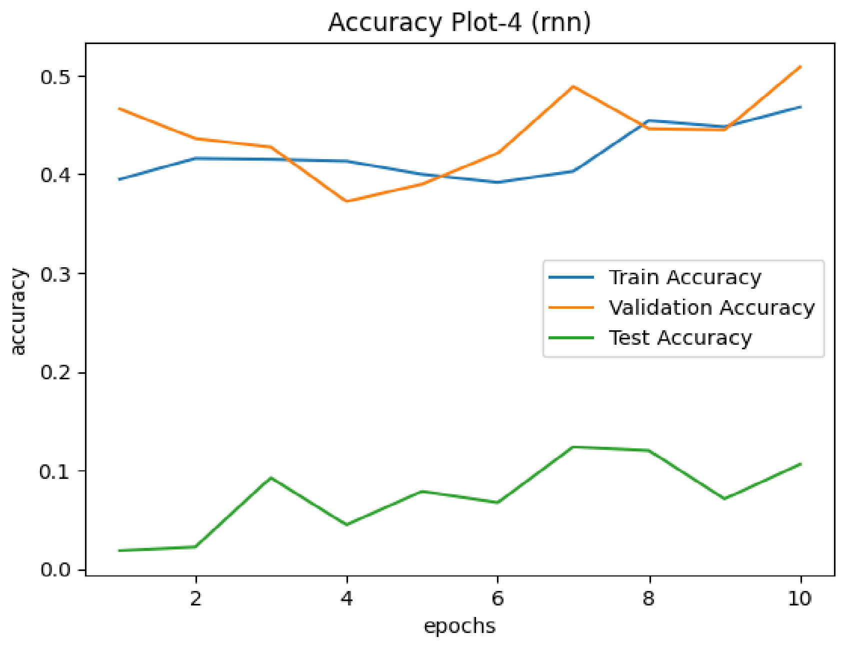

Fig 9. RNN accuracy plot for model with hidden dimensions = 256

(BONUS) We tried modifying the models by changing the provided implementation code. The changes and reports are given below:

Modified FFNN (for Bonus): We added another additional layer to the original model.

The implementation of the code is as follows:

```python
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.activation1 = nn.ReLU()  # first layer
       self.W2 = nn.Linear(h, 16)
       self.activation2 = nn.ReLU() # second layer
       self.output_dim = 5
       self.W3 = nn.Linear(16, self.output_dim)
       self.softmax = nn.LogSoftmax()
       self.loss = nn.NLLLoss()
   def forward(self, input_vector):
       x = self.W1(input_vector)
       # [to fill] obtain output layer representation - 2 layers
       x = self.activation1(x)
       x = self.W2(x)
       x = self.activation2(x)
       x = self.W3(x)
       predicted_vector = self.softmax(x)
       return predicted_vector
```

We ran the model for 5 epochs and 32 hidden dimensions. The accuracy plot of the modified FFNN model:

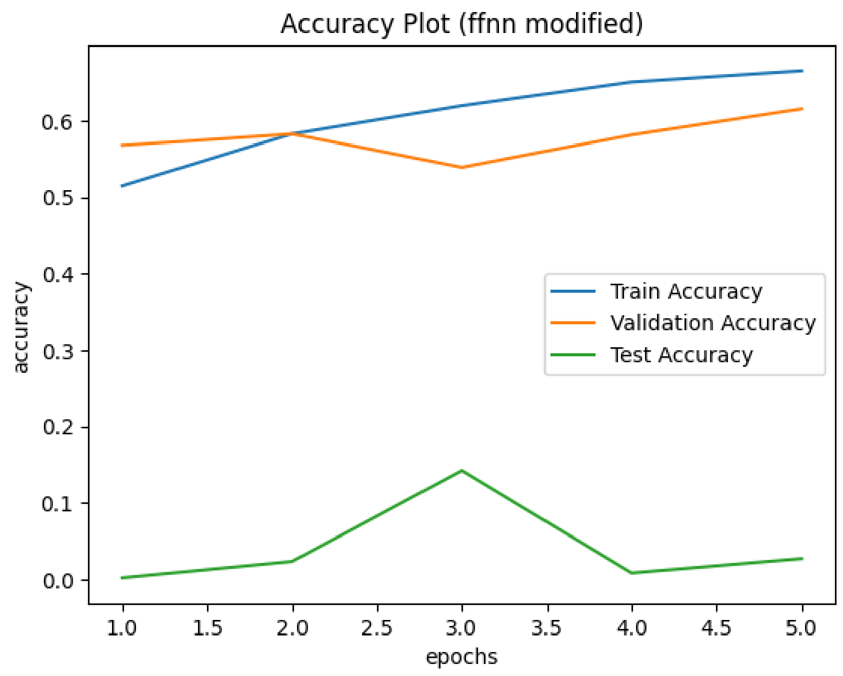

Fig 10. modified FFNN accuracy plot for model with hidden dimensions = 32 (for bonus question)

Modified RNN (for Bonus): To try a different variation of the model, We changed the nonlinearity function to "ReLU" (previously "tanh"). We introduced a dropout of 0.2.

The implementation of the RNN is as follows code:

```
def __init__(self, input_dim, h):  # Add relevant parameters

   super(RNN, self).__init__()

   self.h = h

   self.numOfLayer = 1

   self.rnn = nn.RNN(input_dim, h, self.numOfLayer, nonlinearity='relu',
dropout = 0.2)

   self.W = nn.Linear(h, 5)

   self.softmax = nn.LogSoftmax(dim=1)

   self.loss = nn.NLLLoss()
```

We ran the model for 5 epochs and 32 hidden dimensions. The resulting Accuracy plot for the modified RNN model:

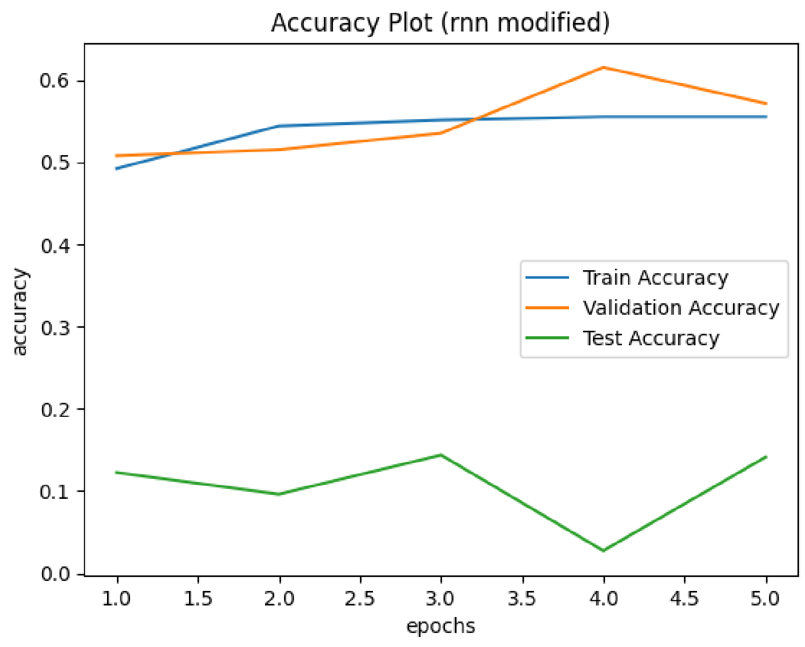

Fig 11. modified RNN accuracy plot for model with hidden dimensions = 32 (for bonus question)

4. Analysis (20pt)

For the development loss of the best system, we are finding the loss for each minibatch, and then summing the losses from each minibatch to get the loss for a particular epoch, and then plotting the loss for each epoch.

FFNN: hidden dimensions = 64, epochs =10, test accuracy (best) = 12.5%

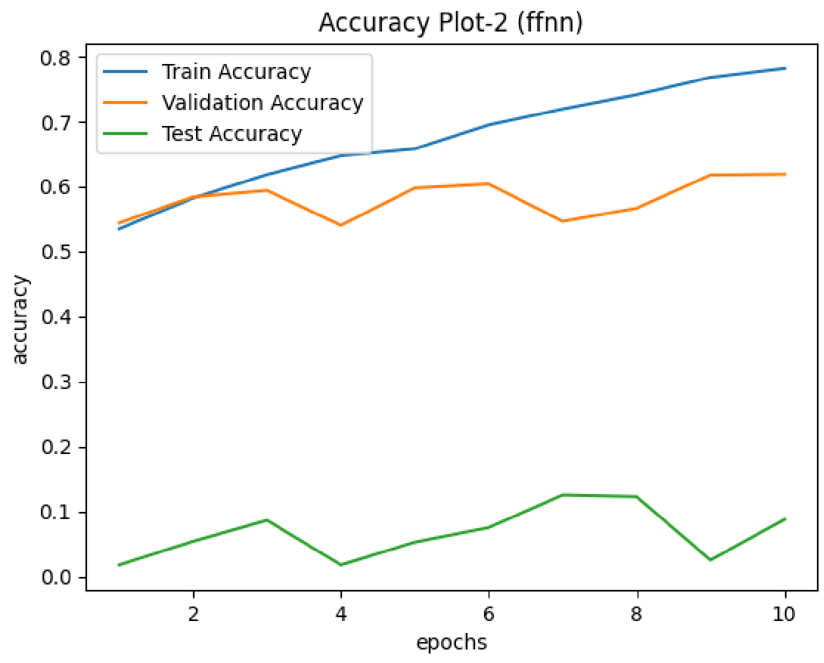

Fig 12. FFNN accuracy plot for model with the best test accuracy (hidden dimensions = 64)

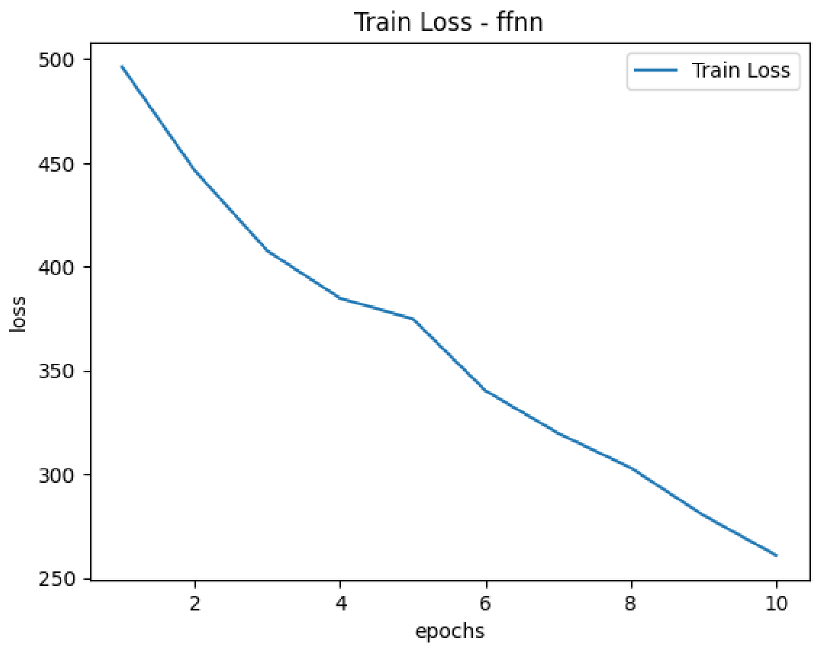

Fig. 13 FFNN training loss plots for model with best test accuracy (hidden dimensions = 64)

RNN: hidden dimensions = 32, epochs =10, test accuracy (best) = 18.875%

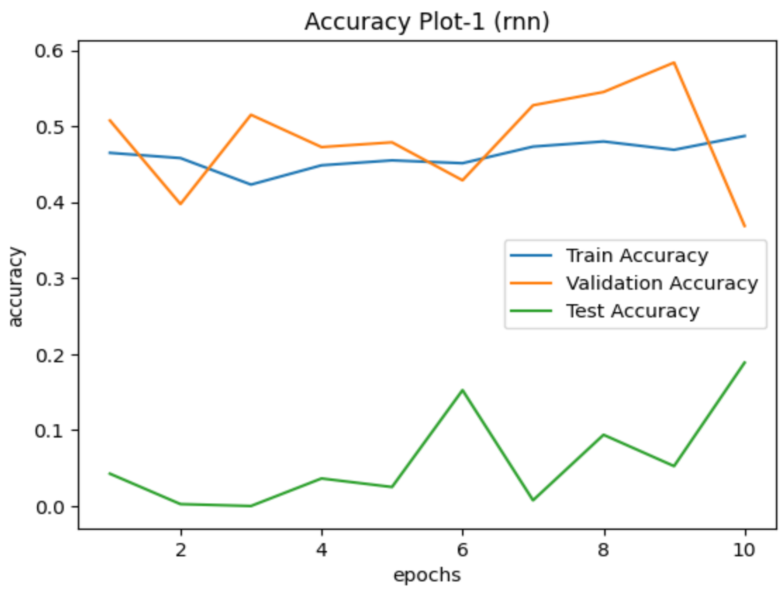

Fig 14. RNN accuracy plot for model with the best test accuracy (hidden dimensions = 32)

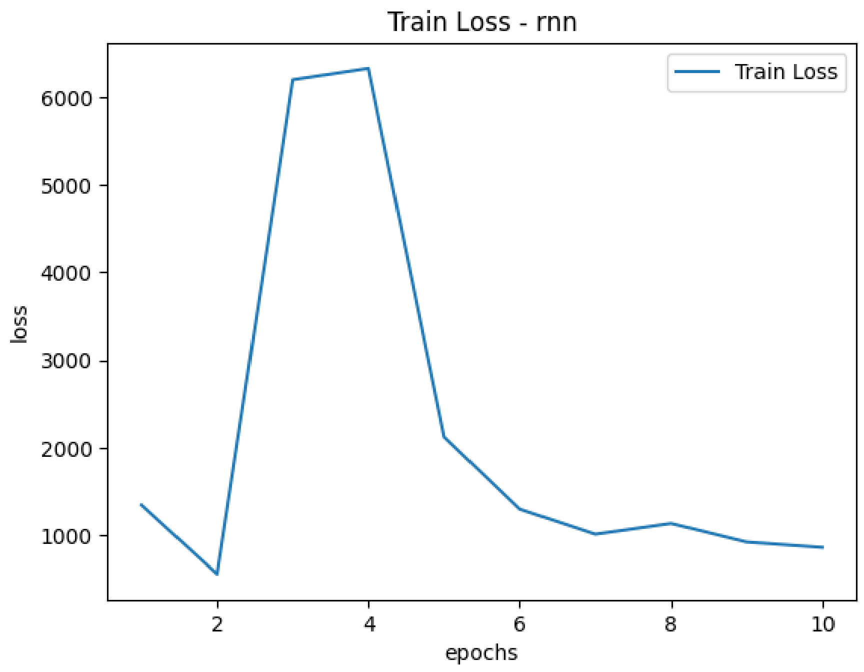

Fig 15. RNN train loss plot for model with the best test accuracy (hidden dimensions = 32)

In both the FFNN and RNN models, the test accuracy values seem significantly lower than the training and validation accuracy as exemplified in the graphs above. This may indicate overfitting, where the model performs well on the training data but poorly on unseen data. The test accuracy should ideally be close to the validation accuracy. (regularization, batch normalization, cross-validation, and drop-out layers)

Not using enough metrics for evaluation – We could have used precision and recall in the implementation.

For FNN, Considering Table 1 where we see the count of labels for the Training, Validation, and Test sets, we clearly see that there is an indication of data imbalance which is strongly reflected in the accuracy difference between training accuracy and validation accuracy. Thus, to improve this, we can have a more balanced dataset with more equally distributed labels across Training, Validation, and Test sets.

(Other Analysis and Discussions - BONUS) As discussed above, data imbalance is a major issue in this project since the accuracies on Training, Validation and Test set are significantly low and vary too much because of it. Thus, to improve model performance, we might have to consider reducing data imbalance in a dataset through the following ways:

I. Ensemble Techniques: Class imbalance can frequently be handled more skillfully by ensemble approaches.

II. We can add more samples to the minority classes across all the 3 sets using some techniques like ADASYN - Adaptive Synthetic Sampling.

5. Conclusion and Others (5pt)

Feedback:

The assignment has been fairly moderate since it deals with the basic implementation of Recurrent Neural Networks (RNN) and Feed-Forward Neural Networks (FFNN). We spent 4-5 days reviewing the slides from the course lecture and the recommended book – (J&M) Dan Jurafsky & James H. Martin, Speech and Language Processing, 3rd edition on topics like, RNN, FFNN, and word embeddings followed by the implementation of the code as instructed in the assignment and reporting the findings.

Individual Member Contribution:

- **Sahana's contributions:** Dataset, Implementations (FFNN), Model Evaluations (Multiple variations), Curve Plotting
- **Sanatan's contributions:** Introduction, Implementation (RNN), Model Evaluations (Metrics), System Performance Summarization, Error Analysis