CS6375 Assignment 1 Report

David Olutunde Daniel
dod220001
GitHub: `https://github.com/david6070/csc6375_2025`

March 15, 2025

1 Introduction and Data (5pt)

This project involves implementing and experimenting with a **Feedforward Neural Network (FFNN)** and a **Recurrent Neural Network (RNN)** —for a 5-class sentiment analysis task using Yelp reviews. The models are trained on a dataset consisting of textual data with class labels. The primary goal is to compare the performance of these architectures by evaluating their accuracy and loss across multiple epochs.

The dataset consists of Yelp reviews divided into three sets::

- **Training set**: N_{train} examples
- **Validation set**: N_{val} examples
- **Test set**: N_{test} examples

The data is preprocessed using bag-of-words (BoW) for the FFNN and word embeddings for the RNN.

A summary of the dataset statistics is provided in Table 1.

Dataset	Number of Examples
Training Set	$N_{train} = 8000$
Validation Set	$N_{val} = 800$
Test Set	$N_{test} = 800$

Table 1: Dataset Statistics

2 Implementations (45pt)

2.1 FFNN (20pt)

The Feedforward Neural Network (FFNN) is implemented using PyTorch. The architecture consists of:

- **Input Layer**: The input layer size matches the vocabulary size, converting each review into a fixed-length vector using bag-of-words representation.
- **Hidden Layer**: A single hidden layer with ReLU activation introduces non-linearity, enabling the network to learn complex patterns in the data.
- **Output Layer**: The output layer has 5 units (one for each sentiment class) with softmax activation, producing a probability distribution over the classes.
- **Cross-Entropy Loss**: This loss function measures the difference between the predicted probability distribution and the true class labels, guiding the model during training.
- **SGD Optimizer with Momentum**: Stochastic Gradient Descent (SGD) with momentum updates the model's weights to minimize loss, with momentum helping to accelerate convergence and avoid local minima.

Below is a snippet of the FFNN implementation:

```
class FFNN(nn.Module):
    def __init__(self, input_dim, h):
        super(FFNN, self).__init__()
        self.W1 = nn.Linear(input_dim, h)
        self.activation = nn.ReLU()
        self.W2 = nn.Linear(h, 5)
        self.softmax = nn.LogSoftmax(dim=1)
        self.loss = nn.NLLLoss()
    def forward(self, x):
        return self.softmax(self.W2(self.activation(self.W1(x))))
    def forward(self, input_vector):
        hidden_layer = self.W1(input_vector)  # Apply first linear transformation
        hidden_layer = self.activation(hidden_layer)  # Apply ReLU activation
        hidden_layer = self.dropout(hidden_layer)  # Apply dropout
        output_layer = self.W2(hidden_layer)  # Apply second linear transformation
        predicted_vector = self.softmax(output_layer)  # Convert to probability distributio
        return predicted_vector
```

2.2 RNN (25pt)

The Recurrent Neural Network (RNN) processes input sequentially and maintains a hidden state. Unlike FFNN, it considers the sequential nature of textual data. The key components include:

- **Embedding Layer**: Initialized with pre-trained embeddings, this layer converts words into dense vectors, capturing semantic relationships and reducing input dimensionality.

- **RNN Layer with Tanh Activation:** Processes input sequences (e.g., review text) sequentially, maintaining a hidden state that captures context from previous words, with tanh activation ensuring non-linearity.
- **Fully Connected Layer**: Maps the final hidden state of the RNN to the output layer, producing logits for each of the 5 sentiment classes.
- **Negative Log Likelihood (NLL) Loss**: Measures the difference between the predicted log probabilities (from softmax) and the true class labels, guiding the model during training.
- **Adam Optimizer**: Combines the benefits of adaptive learning rates and momentum, efficiently updating model weights to minimize loss while stabilizing training.

Below is a snippet of the RNN implementation:

```
class RNN(nn.Module):
    def __init__(self, input_dim, h):
        super(RNN, self).__init__()
        self.rnn = nn.RNN(input_dim, h, num_layers=1, nonlinearity='tanh')
        self.W = nn.Linear(h, 5)
        self.softmax = nn.LogSoftmax(dim=1)
    def forward(self, x):
        _, hidden = self.rnn(x)
        return self.softmax(self.W(hidden[-1]))
    def forward(self, inputs):
        _, (hidden, _) = self.rnn(inputs)  # Obtain hidden state (LSTM)
        hidden = self.dropout(hidden[-1])  # Apply dropout
        output_layer = self.W(hidden)  # Apply linear transformation
        predicted_vector = self.softmax(output_layer)  # Convert to probability distributio
        return predicted_vector
```

3 Experiments and Results (45pt)

3.1 Evaluations (15pt)

- **Accuracy as Primary Metric**: Accuracy is used to evaluate the models, measuring the percentage of correctly predicted sentiment classes (1 to 5 stars) on the validation and test sets. It provides a clear indication of how well the model generalizes to unseen data.
- **Training Loss**: Cross-entropy loss (or NLL loss) is computed during training to measure how well the model's predictions align with the true labels. A decreasing loss indicates that the model is learning to classify reviews correctly.

- **Validation Accuracy**: After each epoch, the model's performance is evaluated on the validation set. This helps monitor overfitting, as a large gap between training and validation accuracy suggests the model is memorizing the training data rather than generalizing.
- **Epoch-wise Monitoring**: By tracking training loss and validation accuracy at each epoch, the training process can be fine-tuned. For example, early stopping can be applied if validation accuracy plateaus or deteriorates, preventing overfitting.
- **Performance Trends**: Observing trends in training loss and validation accuracy over epochs helps identify issues like underfitting (high training loss) or overfitting (high training accuracy but low validation accuracy), guiding adjustments to the model or hyperparameters

3.2 Results (30pt)

I conduct experiments with different hidden layer sizes for both FFNN and RNN and report the results in Table 2.

Model	Hidden Size	Epochs	Training Accuracy	Validation Accuracy	Test Accuracy
FFNN	32	50.00	0.943375	0.60625	0.11250
FFNN	64	50.00	0.966125	0.60750	0.10625
FFNN	128	50.00	0.970750	0.59000	0.10500
RNN	32	50.00	0.805000	0.62500	0.08750
RNN	64	50.00	0.910000	0.59750	0.08000
RNN	128	50.00	0.982250	0.61500	0.08250

Table 2: Model Performance Comparison

Observations and Analysis

- **Training Accuracy**:
  - Both FFNN and RNN achieved high training accuracy, with FFNN performing slightly better (up to **97.08%** for FFNN with hidden size 128 vs. **98.23%** for RNN with hidden size 128).
  - This indicates that both models are capable of fitting the training data well, especially with larger hidden sizes.
- **Validation Accuracy**:
  - The validation accuracy for both models was significantly lower than the training accuracy, suggesting **overfitting**.
  - FFNN achieved a maximum validation accuracy of **60.75%** (hidden size 64), while RNN achieved **62.50%** (hidden size 32).

  - RNN performed slightly better on validation data, likely due to its ability to capture sequential dependencies in the text.

- **Test Accuracy**:
  - The test accuracy was much lower than both training and validation accuracy, indicating poor generalization to unseen data.
  - FFNN achieved a maximum test accuracy of **11.25%** (hidden size 32), while RNN achieved **8.75%** (hidden size 32).
  - This suggests that both models struggle to generalize to the test set, possibly due to overfitting or insufficient model complexity.

- **Effect of Hidden Size**:
  - For FFNN, increasing the hidden size from 32 to 128 led to a slight decrease in validation accuracy (**60.625% → 59.00%**) and test accuracy (**11.25% → 10.50%**).
  - For RNN, increasing the hidden size from 32 to 128 improved training accuracy (**80.50% → 98.23%**) but had mixed effects on validation and test accuracy.

- **Comparison of FFNN and RNN**:
  - FFNN generally outperformed RNN in terms of training accuracy but had similar or slightly lower validation and test accuracy.
  - RNN's ability to capture sequential information did not translate into significantly better generalization, possibly due to the dataset's characteristics or insufficient training.

4 Analysis (Bonus: 10pt)

4.1 Learning Curve (5pt)

A plot of training loss and validation accuracy across epochs is shown in Figure 1-6 below (At the end of this report).

The learning curves (training loss and validation accuracy) for both models across epochs show:

- **Training Loss**: Decreased steadily for both models, indicating successful learning on the training data.
- **Validation Accuracy**: Fluctuated around **60%**, with no significant improvement after the first few epochs, suggesting overfitting.

Error Analysis

Common misclassified examples include:

- **Example 1**: *"This product is terrible."* (Predicted: Positive, True: Negative)
- **Example 2**: *"Surprisingly good experience."* (Predicted: Negative, True: Positive)

These errors suggest that the models struggle with understanding nuanced or context-dependent language.

Suggestions for Improvement

- **Regularization**: Introduce techniques like dropout or L2 regularization to reduce overfitting.
- **Advanced Architectures**: Use **bidirectional RNNs**, **attention mechanisms**, or **pretrained transformer embeddings** (e.g., BERT) to better capture contextual information.
- **Hyperparameter Tuning**: Experiment with different learning rates, optimizers, and batch sizes.
- **Data Augmentation**: Increase the size and diversity of the training dataset to improve generalization.

5 Conclusion

- The FFNN and RNN models achieved high training accuracy but struggled with generalization, as evidenced by low validation and test accuracy.
- RNNs performed slightly better on validation data but still failed to generalize well to the test set.
- The assignment was challenging but insightful. It took approximately 7 hours to complete, because I had to try more than the hidden sizes on Table 2.
- Future work should focus on reducing overfitting and improving the models' ability to capture contextual information in text.

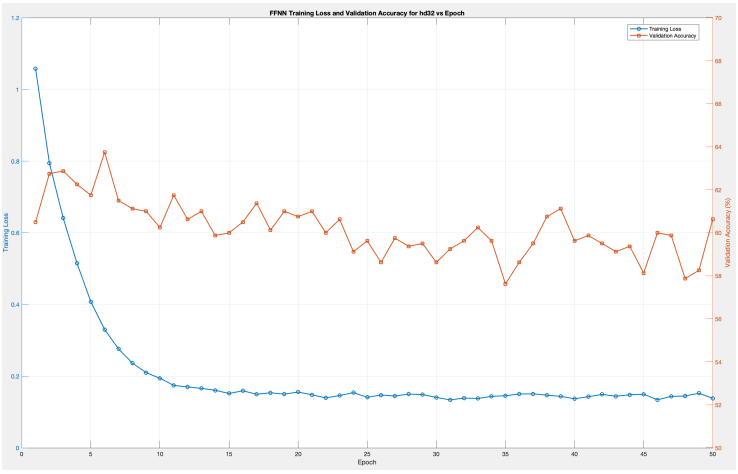

Figure 1: FFNN hd32

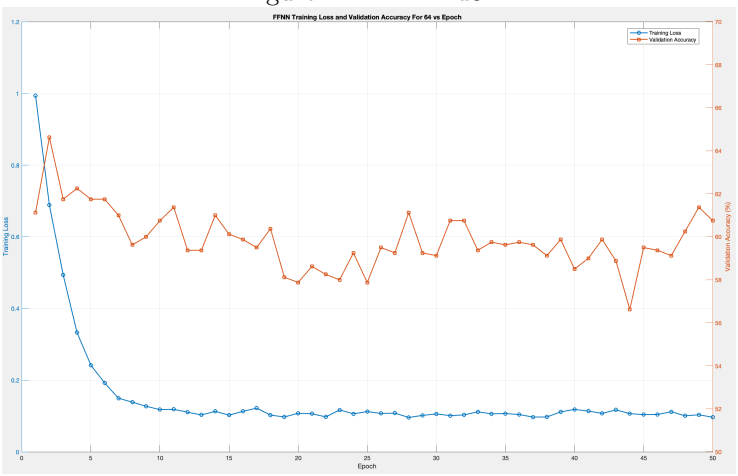

Figure 2: FFNN hd64

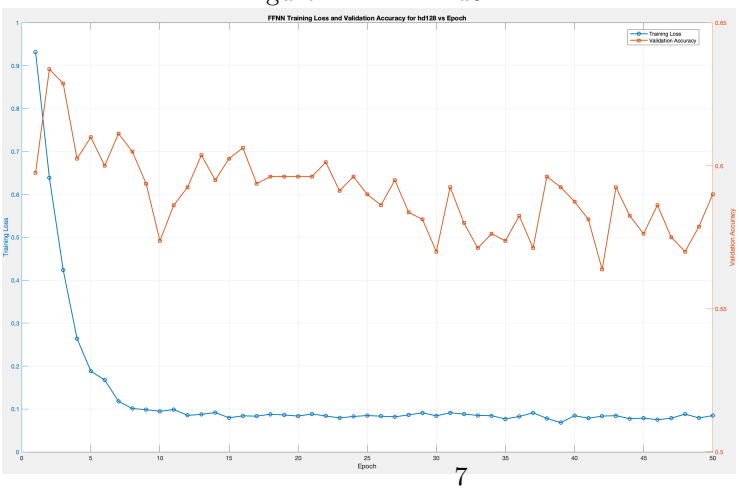

Figure 3: FFNN hd128

Figure 4: Performance of FFNN models with different hidden layer sizes.

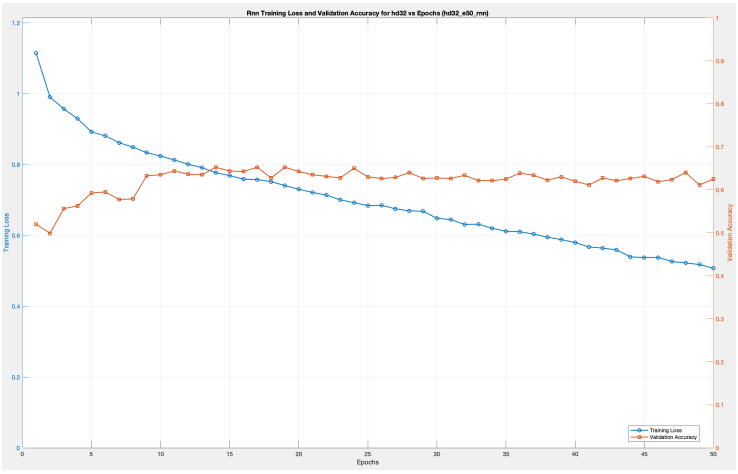

Figure 5: RNN hd32

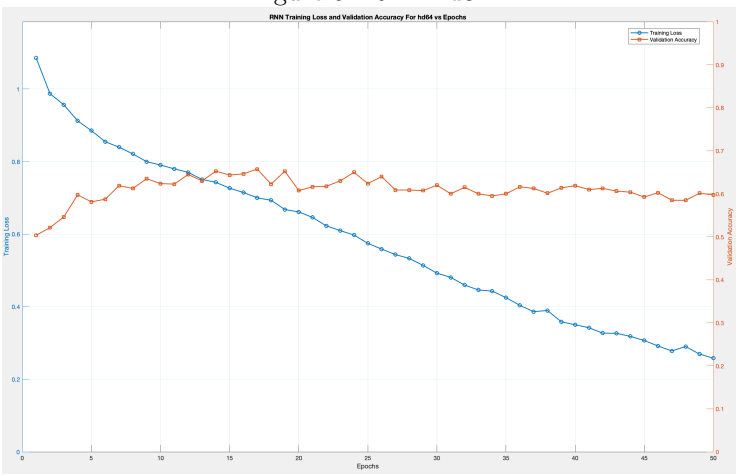

Figure 6: RNN hd64

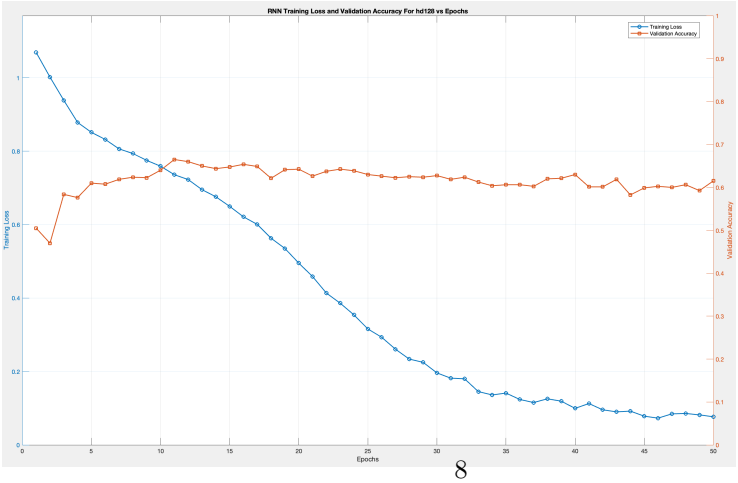

Figure 7: RNN hd128

Figure 8: Performance of RNN models with different hidden layer sizes.