CS4395 Assignment 2 Final Report

https://github.com/FarhanJamil0001/hltA2

Farhan Jamil
Net ID: fxj200003

April 15, 2025

1 Introduction and Data

In this assignment, we compare two deep learning models for sentiment classification on Yelp review data: a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN). The task is a 5-class classification problem, with the goal of predicting a review rating from 1 to 5 stars based on the text. We use bag-of-words representations for FFNN and pretrained word embeddings for RNN.

The Yelp dataset is provided in JSON format, split into training, validation, and test sets.

2 Dataset Statistics

This project uses a Yelp reviews dataset for sentiment analysis, framed as a 5-class classification task. Each example is a review string paired with its associated star rating (1–5). For internal computation, we remap ratings to labels 0–4. The reviews are preprocessed via tokenization, punctuation removal, and vectorization (bag-of-words or word embeddings).

Metric	Training Set	Validation Set	Test Set
# Examples	8,000	800	800
Avg Length (words)	141.25 ± 124.60	140.37 ± 117.93	109.77 ± 96.45
Min Length	2	5	9
Max Length	989	980	782

Table 1: Summary statistics of the Yelp dataset for training, validation, and test sets.

Class Distribution:

- **Training Set:** Ratings are mostly 1- and 2-star reviews (3,200 each; 40% each) and 3-star reviews (1,600; 20%).
- **Validation Set:** Same ratio as the training set for 1-star (40%), 2-star (40%), and 3-star (20%).
- **Test Set:** Contains 3-star (20%), 4-star (40%), and 5-star (40%) reviews, indicating a distinct distribution from training/validation.

Key Observations:

1. The training and validation sets are skewed toward lower-star reviews, while the test set is skewed toward higher-star reviews.
2. The average review length is about 140 words for training and validation, and 110 words for the test set, with high variance in length.
3. Mapping star labels 1–5 to 0–4 is done internally for the classification labels.
4. The difference in rating distribution between training/validation and test sets may increase the task's difficulty.

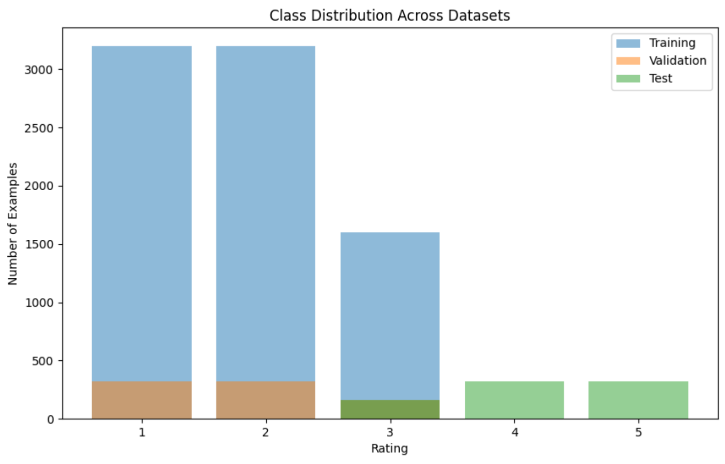

Figure 1: Bar graph on class distribution across datasets.

3 Implementations

3.1 FFNN

Our Feedforward Neural Network (FFNN) implementation uses a classic bag-of-words approach:

- **Input Layer:** A $|V|$-dimensional vector, where $|V|$ is the vocabulary size after tokenizing the data and mapping words to indices.
- **Hidden Layer:** A single linear layer of dimension h (a hyperparameter). We apply a ReLU activation (`nn.ReLU`) to introduce non-linearity. Optionally, a `Dropout` layer (`p=0.2` or other rate) can be included to mitigate overfitting.
- **Output Layer:** A final linear layer of size 5 (one logit per sentiment class), followed by `LogSoftmax` to generate probability distributions.

The forward method computes a hidden representation and applies dropout and softmax:

```
def forward(self, input_vector):
    hidden = self.activation(self.W1(input_vector))
    hidden = self.dropout(hidden)
    output = self.W2(hidden).view(1, -1)
    predicted_vector = self.softmax(output)
    return predicted_vector
```

Loss and Optimization: We use negative log-likelihood loss with:

```
self.loss = nn.NLLLoss()
...
def compute_Loss(self, predicted_vector, gold_label):
    return self.loss(predicted_vector, gold_label)
```

Training: We train the model for 10 epochs using a mini-batch size of 16. During training, we record loss and metrics like accuracy, precision, recall, and F1 score. Optimization is performed using either SGD with momentum or Adam. Here's a key training loop excerpt:

```
for epoch in range(epochs):
    model.train()
    for minibatch in train_data:
        ...
        predicted_vector = model(input_vector)
        loss = model.compute_Loss(predicted_vector, torch.tensor([gold_label]))
        ...
        loss.backward()
        optimizer.step()
```

Visualization: At the final epoch, we generate a confusion matrix for the validation set:

```
cm = confusion_matrix(valid_labels, valid_preds)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.title('Validation Confusion Matrix')
plt.savefig('ffnn_confusion_matrix_epoch10.png')
```

The FFNN implementation is modular and supports CLI arguments for hyperparameter tuning, dropout experiments, and generating learning curves.

3.2 RNN

Our Recurrent Neural Network (RNN) is designed for sequence modeling of Yelp reviews:

- **Embedding Layer:** We load a pretrained embedding dictionary (dimension 50) and convert each word in a review into a 50-dimensional vector. These are stacked into a sequence tensor of shape $(sequencelength, 1, 50)$.
- **Recurrent Layer:** Depending on a user-specified `cell_type`, we instantiate `nn.RNN`, `nn.LSTM`, or `nn.GRU`. By default, we use a single layer with hidden dimension h. The final hidden state (or combined output in the RNN case) encodes the review.
- **Output Layer:** A linear transformation maps the final hidden state to a 5-dimensional output, and `LogSoftmax` produces class probabilities.

The recurrent architecture is modular and supports dynamic selection of cell type:

```
if self.cell_type == 'rnn':
    self.rnn = nn.RNN(input_dim, h, self.numOfLayer)
elif self.cell_type == 'lstm':
    self.rnn = nn.LSTM(input_dim, h, self.numOfLayer)
elif self.cell_type == 'gru':
    self.rnn = nn.GRU(input_dim, h, self.numOfLayer)
```

The forward method handles all three cell types and applies classification:

```
def forward(self, inputs):
    if self.cell_type == 'lstm':
        _, (hidden, _) = self.rnn(inputs)
    else:
        _, hidden = self.rnn(inputs)

    output = self.W(hidden.squeeze(0))
    summed_output = torch.sum(output, dim=0).view(1, -1)
    predicted_vector = self.softmax(summed_output)
    return predicted_vector
```

Training Process: The training script tokenizes and cleans the input, converts it into tensors of word embeddings, and performs batch-wise optimization using Adam. Training is done for 10 epochs with batch size 16.

```
vectors = [word_embedding.get(word, word_embedding['unk']) for word in text_line]
vectors = torch.tensor(vectors, dtype=torch.float32).view(len(vectors), 1, -1)
output = model(vectors)
loss = model.compute_Loss(output, torch.tensor([gold_label]))
loss.backward()
optimizer.step()
```

Loss Function:

```
self.loss = nn.NLLLoss()
...
def compute_Loss(self, predicted_vector, gold_label):
    return self.loss(predicted_vector, gold_label)
```

Validation Visualization: We plot a confusion matrix after the final validation epoch:

```
cm = confusion_matrix(valid_labels, valid_preds)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.title('Validation Confusion Matrix')
plt.savefig('rnn_confusion_matrix_epoch10.png')
```

Hyperparameter Modes:

- `--multi`: Sweeps hidden dimensions ($h \in \{64, 128, 256\}$)
- `--multi_cell`: Compares `rnn`, `lstm`, and `gru` variants

Both FFNN and RNN implementations are modular and tunable via command-line arguments, making the framework well-suited for structured experimentation and reproducibility.

4 Experiments and Results

Evaluations

Our primary evaluation metric is **accuracy** (correct predictions / total predictions). We also track average **precision**, **recall**, and **F1** across classes for both training and validation, to capture additional nuances of model performance.

Results

We conduct experiments with different hidden-layer sizes (64, 128, 256) for both the Feedforward Neural Network (FFNN) and the Recurrent Neural Network (RNN). Each is trained for 10 epochs using a mini-batch size of 16, and we record training and validation metrics each epoch.

FFNN Results

Hidden Dim = 64

- Final Validation Accuracy: 61.88%
- Precision/Recall/F1: Around 0.63 accuracy, with F1 up to 0.62 by the final epoch.

Hidden Dim = 128 (Best)

- Final Validation Accuracy: **62.62%**
- Precision/Recall/F1: Peaks near 0.62–0.63 at final epoch.

Hidden Dim = 256

- Final Validation Accuracy: 59.88%
- Demonstrated some overfitting, with final F1 around 0.58.

Observations (FFNN):

- Hidden dimension 128 yields the best validation accuracy (˜62.62%).
- Larger hidden dimension can slow training and risk overfitting.
- Precision/recall/F1 trends largely mirror accuracy, confirming that 128 is optimal among tested sizes.

Dropout Experiment

To evaluate the impact of dropout on overfitting, we compared a baseline FFNN (no dropout) to a variant with a dropout rate of $p = 0.2$. Both models used the same hyperparameters (hidden dimension of 128, 5 training epochs).

Baseline (No Dropout)

- Final Validation Accuracy: 62.12%
- Observed steady improvement in training accuracy, peaking with a validation accuracy of over 62%.

Model with Dropout ($p = 0.2$)

- Final Validation Accuracy: 60.25%
- While training accuracy also improved, the final validation accuracy was slightly lower than the baseline.

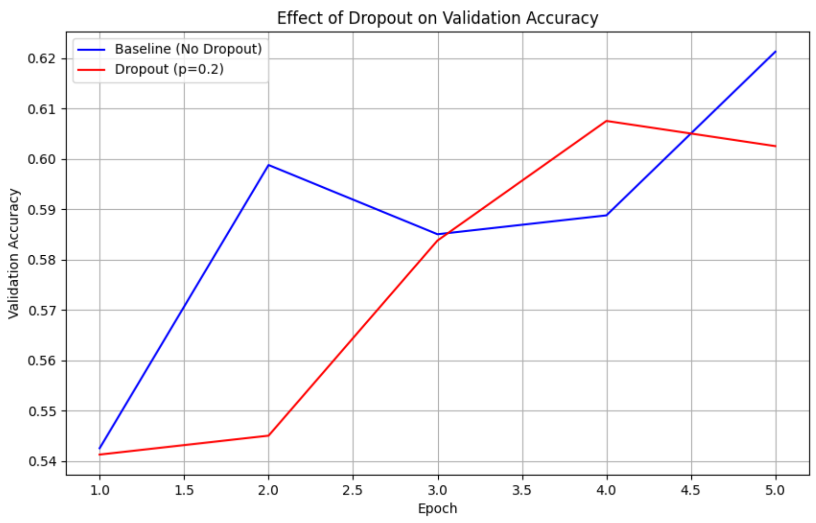

Figure 2: Validation accuracy comparison between baseline and dropout model ($p = 0.2$).

Observations:

- Overall performance did not improve with dropout at $p = 0.2$.
- Dropout often helps prevent overfitting in deeper or more prolonged training scenarios. In these 5 epochs, the benefit was not fully realized.
- Fine-tuning dropout probability or running more epochs could yield different outcomes.
- Precision, recall, and F1 values were broadly comparable to the baseline but did not surpass it in final accuracy.

RNN Results

Our RNN uses pretrained word embeddings (dimension 50) as input. We evaluated hidden dimensions 64, 128, and 256, as well as different cell types: `rnn`, `lstm`, and `gru`. For each setting, we recorded accuracy, precision, recall, and F1 across 10 epochs.

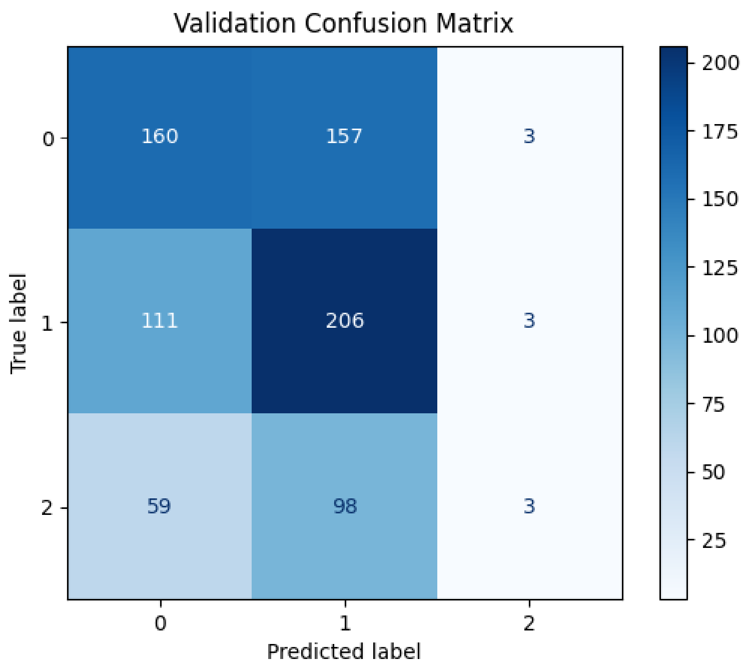

Figure 3: Validation accuracy comparison between baseline and dropout model ($p = 0.2$).

Hidden Dim = 64 (Best among hidden size experiments)

- Final Validation Accuracy: **46.12%**
- Precision, Recall, F1: Final epoch sees precision ∼0.44, recall ∼0.46, and F1 ∼0.42.
- Training time per epoch: 29–42 seconds.

Hidden Dim = 128

- Final Validation Accuracy: 40.88%
- F1 remains under 0.40 across most epochs.
- Shows more fluctuation and less improvement over epochs.

Hidden Dim = 256

- Final Validation Accuracy: 43.75%
- Training time: up to 130s/epoch.
- Slight improvement over 128-dim, but still worse than 64-dim.

Cell Type: Vanilla RNN

- Final Validation Accuracy: 42.25%
- F1 scores fluctuate, ranging between 0.23 and 0.39.
- Struggles with generalization despite stable training.

Cell Type: LSTM (Best among all RNN variants)

- Final Validation Accuracy: **56.87%**
- Precision, Recall, and F1 steadily improve; F1 reaches 0.75 during training and 0.57 at validation.
- Shows strong generalization and consistently better performance.

Cell Type: GRU

- Final Validation Accuracy: 47.00%
- Performs better than vanilla RNN but worse than LSTM.
- Final F1 on validation: around 0.61.

Observations (RNNs overall):

- Among fixed-size RNNs, hidden dimension 64 gives best results.
- LSTM significantly outperforms both vanilla RNN and GRU with a validation accuracy of 56.87%, closing the gap with FFNN.
- GRU provides a reasonable compromise between training time and performance.
- LSTM's stability and higher validation metrics make it the best candidate for future improvements.
- Future experiments could include bi-directional RNNs, multi-layer architectures, and hyperparameter tuning.

5 Analysis

Learning Curve

Both FFNN and RNN models were monitored with learning curves for training loss and validation accuracy. The FFNN showed a consistent downward trend in training loss with a corresponding rise in validation accuracy, indicating effective learning and generalization. Notably, the best-performing FFNN model (hidden dim = 128) peaked at over 62% validation accuracy without significant signs of overfitting. The dropout experiment showed that while dropout modestly reduced training accuracy, it did not improve generalization in the short 5-epoch window.

The RNN-based models exhibited more erratic learning behavior. Vanilla RNNs showed considerable fluctuation in both training and validation metrics across epochs. Validation accuracy frequently plateaued early, often below 45%. The LSTM variant improved significantly on this front, reaching nearly 57% accuracy and demonstrating better convergence and generalization. GRU also outperformed the vanilla RNN but failed to match the LSTM's stability.

Error Analysis

Confusion matrices revealed frequent misclassifications between adjacent rating classes (e.g., predicting 3 stars when the true label was 4). This suggests that the models, especially RNNs, struggled to pick up on subtle semantic shifts in sentiment that differentiate close scores.

Additionally, we observed that longer or more nuanced reviews posed challenges for all models. The FFNN, lacking sequential information, often failed to leverage context across distant tokens. Meanwhile, the simple RNN suffered from vanishing gradient issues, leading to inconsistent performance on lengthy sequences. Although LSTM and GRU helped address this to a degree, their training times were significantly longer, and even their best F1 scores on validation did not exceed 0.61.

Another key factor was the class distribution mismatch between training/validation (mostly 1–3 star reviews) and the test set (mostly 3–5 stars). This mismatch likely affected generalization, especially for RNN models trained on sequences optimized for lower-star semantics. Future training with a balanced dataset or data augmentation may help alleviate this skew.

Takeaways

- FFNNs are simple yet effective for bag-of-words-based sentiment classification.
- RNNs require more careful tuning, longer training times, and architectural enhancements (like LSTM/GRU) to be competitive.
- Pretrained embeddings alone are not enough to close the performance gap without stronger sequence modeling.
- The label distribution mismatch between train and test sets adds to the difficulty and should be considered in future splits or augmentation strategies.

6 Conclusion and Others

Conclusion

FFNN outperformed RNN for this sentiment classification task. The best model was FFNN with hidden dim 128, achieving 62.62% validation accuracy. RNN reached only 46.12%, highlighting the challenges of sequence modeling with limited epochs and simple RNNs.

Individual Contribution

I implemented all additional parts of the FFNN and RNN models, performed experiments, generated results, and compiled the report.

Feedback

The assignment was engaging and helped reinforce core concepts in neural modeling. Tuning RNNs was more difficult. It would be helpful to provide more debug utilities or logs for early performance analysis.