CS6320 Assignment 2

https://github.com/AdemOdza/CS6320_Assignment2

Group 10 Abhirup Mukherjee AXM240026 Adem Odza AXO180008 Buvana Seshathri BXS240020

Samiha Kuncham SXK240025

1 Introduction and Data

In this assignment, we complete and evaluate two neural network model implementations for 5-class sentiment analysis using Yelp restaurant reviews. Given the review data, the task is to predict the star rating (between 1 and 5 stars). The models we experiment with are: a Feedforward Neural Network (FFNN) using BOW representation, and a Recurrent Neural Network (RNN) using word embeddings.

1.1 Task Description

The task is a multi-class classification problem wherein the model learns to map features from the text to discrete ratings. Given a resturant review, it must correctly predict the corresponding star rating where 1 star is very negative and 5 stars is very positive.

1.2 Dataset Overview

The dataset provided by this assignment was pre-split into training, validation, and testing sets. Below are some statistics about the dataset:

Dataset	**Examples**	**Avg Length**	**Min Length**	**Max Length**
Training	16,000	124.7 words	1 word	989 words
Validation	800	140.4 words	5 words	980 words
Test	800	109.8 words	9 words	782 words

Table 1: Dataset Statistics

1.3 Main Experimental Results

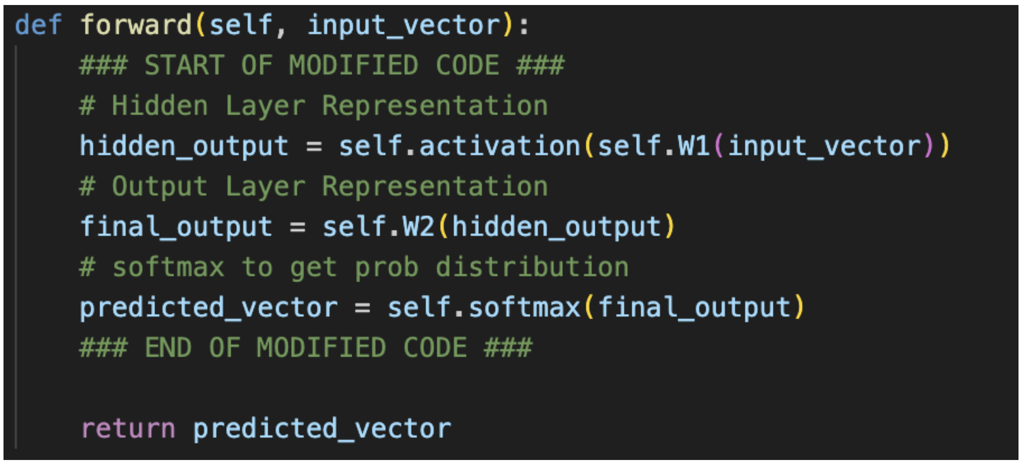
For the FFNN model, we performed our experiments by varying the following hyperparameters: number of hidden dimensions (16, 32, 64, 128) and number of epochs (5, 7, 10, 15). We also tried varying the learning rate to see how it had an effect on overfitting. Out of all the experiments, our FFNN implementation achieved a highest test accuracy of **59.38%** using these hyperparameter values: `hidden_dim=32`, `epochs=10` (with default `lr=0.01`).

2 Implementations

2.1 FFNN

2.1.1 Forward Pass Implementation

The primary task of the FFNN section was to complete the `forward()` function in `ffnn.py`. This is the feedforward computation through the layers of the network. Figure 1 shows our implementation.



```python
def forward(self, input_vector):
    ### START OF MODIFIED CODE ###
    # Hidden Layer Representation
    hidden_output = self.activation(self.W1(input_vector))
    # Output Layer Representation
    final_output = self.W2(hidden_output)
    # softmax to get prob distribution
    predicted_vector = self.softmax(final_output)
    ### END OF MODIFIED CODE ###

    return predicted_vector
```

Figure 1: FFNN Forward Pass Implementation

We used three lines of code to compute the forward pass:

1. **Hidden Layer Computation**: To compute the hidden later we first do a linear transformation `W1` followed by `ReLU` activation.
2. **Output Layer Computation**: Next, we apply another linear transformation `W2` that maps the the hidden dimensions to the 5 rating output classes.
3. **Probability Distribution**: Finally, we apply softmax to get our log probabilities which are a probability distribution over the 5 ratings.

2.1.2 Additional Modifications

1. Test Data Loading Apart from completing the `forward()` function in `ffnn.py`, we also modified the `load_data()` function to take the test data as another argument.

2. Learning Curve Tracking To plot the learning curve in Section 3, we also added used two lists to track and accumulate the training losses and validation accuracies for each epoch.

3. Test Evaluation To run the inference on the test data, we wrote a loop similar to the validation one that iterates through the test data and calculates the model's accuracy. We also set the model to evaluation mode using `model.eval()` which switches the model to eval mode by disabling dropout layers, etc.

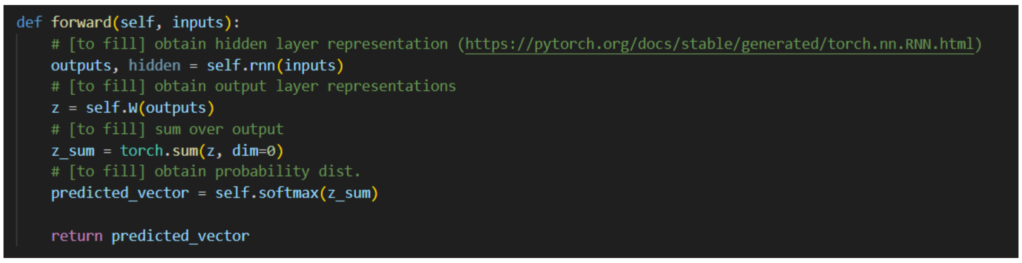
2.1.3 Understanding Other Components

The optimizer used is **Stochastic Gradient Descent** which performs mini-batch gradient descent. We also noticed that the current implementation does not have any stopping mechanism such as **early stopping** where we stop the training once the validation accuracy starts to drop.

2.2 RNN

2.2.1 RNN forward() implementation

- **Inputs:** Each example is converted into a sequence of word embeddings shaped $[T, 1, 50]$, where T is the number of tokens, batch size = 1, and embedding dimension = 50.
- **Obtain hidden layer representation:** A 1-layer tanh RNN maps each timestep to a hidden state, returning outputs with shape $[T, 1, h]$.
- **Obtain output layer representations:** A linear layer $\mathbf{W}$ projects each timestep's hidden state to class logits, resulting in an output of shape $[T, 1, 5]$.



```
def forward(self, inputs):
    # [to fill] obtain hidden layer representation (https://pytorch.org/docs/stable/generated/torch.nn.RNN.html)
    outputs, hidden = self.rnn(inputs)
    # [to fill] obtain output layer representations
    z = self.W(outputs)
    # [to fill] sum over output
    z_sum = torch.sum(z, dim=0)
    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(z_sum)

    return predicted_vector
```

Figure 2: RNN Forward Pass Implementation

- **Sum over output:** Per-timestep logits are aggregated by summing along the time dimension ($dim = 0$) to produce a single vector of shape $[1, 5]$ for the entire sequence:

$$\mathbf{z}_{agg} = \sum_{t=1}^{T} \mathbf{z}_t$$

- **Obtain probability dist:** The aggregated vector is passed through a softmax layer to produce a probability distribution over the output classes:

$$\mathbf{p} = softmax(\mathbf{z}_{agg})$$

  Each value in $\mathbf{p}$ represents the model's predicted probability for a class.

2.2.2 Changes in rnn.py

We have made some modifications in the code to increase accuracy.

- Test data is loaded together with the training and validation data, and additional code is included to compute test accuracy.
- Code has been added to plot the training loss and validation accuracy across epochs for better visualization of learning progress.
- The learning rate was adjusted to 0.001 (from 0.01) to improve model performance and achieve more stable accuracy.

2.2.3 Rest of rnn.py

- **Data Loading:** Reads JSON files into (`tokens`, `label`) pairs and reprocesses text by removing punctuation for consistency.
- **Vectorization:** Converts words to embeddings from `word_embedding.pkl`, forming tensors of shape $[T, 1, 50]$ for RNN input.
- **Model & Optimization:** The RNN sums output vectors across timesteps, applies a linear layer, then LogSoftmax for class probabilities; trained using NLLLoss and Adam optimizer.
- **Training & Validation:** Trains over minibatches, tracks accuracy, and performs early stopping if validation accuracy decreases.
- **Testing:** Evaluates on the test set using `argmax` predictions to compute the final accuracy.

2.2.4 RNN vs FFNN

- **Sequential Processing:** RNN code processes input words one at a time in sequence. FFNN code processes the entire input as a single fixed-length vector without considering order.
- **Hidden State:** RNN maintains a hidden state (`hidden`) that carries information from previous timesteps. FFNN has no memory, each input is processed independently.
- **RNN Layer vs Linear Layer:** RNN uses `nn.RNN(input_dim, h, num_layers)` to handle sequential data. FFNN uses `nn.Linear(input_dim, h)` for a single transformation step.

- **Dependencies:** RNN captures context and dependencies between words across time. FFNN cannot capture sequence relationships or order information.
- **Input Representation:** RNN takes word embeddings (e.g., pretrained vectors) as sequential inputs. FFNN uses bag-of-words or frequency-based vector representations.
- **Computation Flow:** RNN's forward pass depends on previous outputs (recurrent connection). FFNN's forward pass depends only on the current input (no recurrence).

3 Experiments and Results

3.1 Evaluations

3.1.1 FFNN

We evaluated our FFNN model using **classification accuracy** as the main metric. Accuracy is calculated using the formula:

$$Accuracy = \frac{NumberofCorrectPredictions}{TotalNumberofExamples} \times 100\%$$

For each review in the test set, the model gives a probability distribution and we select the class with the highest probability using `torch.argmax(predicted_vector)` and compare it to the gold label. We also track the **training loss** (negative log likelihood) for all epochs. This helps us track whether or not the model converges.

3.1.2 RNN

We evaluated our RNN model using classification accuracy as the main metric. The training objective was to minimize the Negative Log-Likelihood Loss (NLLLoss).

- **Training (per epoch):** While accumulating minibatch loss, it also counts correct predictions to compute training accuracy for that epoch.
- **Validation (per epoch):** After each epoch, the model is run on the validation set to compute accuracy (`correct/total`). A simple early-stopping heuristic halts training if validation accuracy drops while training accuracy rises, acting as an overfitting guard.
- **Test (after training):** Evaluates once on the test set and reports final test accuracy. The training loop computes training accuracy, the validation loop computes validation accuracy, and the final block computes test accuracy and prints:

```
Test accuracy:  0.4850
```

3.2 Results

3.2.1 FFNN

We conducted 7 experiments by varying `hidden_dim`, `epochs`, and `lr`. Table 2 shows our results.

Exp. No.	Hidden Dim	Epochs	LR	Final Train Acc	Best Val Acc	Final Val Acc	Test Acc
1	16	7	0.01	62.29%	58.00%	51.13%	56.50%
2	32	5	0.01	60.29%	55.88%	55.75%	47.63%
3	32	7	0.01	64.49%	58.63%	56.88%	48.00%
4*	32	10	0.01	71.01%	58.63%	48.63%	**59.38%**
5	64	7	0.01	64.86%	**60.13%**	57.50%	46.13%
6	32	10	0.005	74.16%	58.25%	54.75%	56.25%
7	32	15	0.001	76.13%	58.25%	56.25%	53.88%

Table 2: FFNN Experimental Results *Best test accuracy

3.2.2 RNN

We have tried different hidden_dim values and below are the results produced by those multiple variations.

Model	Hidden Dim	Epochs	Val Acc	Test Acc
RNN	32	10	46.5%	43.5%
RNN	64	10	46.875%	48.50%
RNN	128	10	42.625%	46.88%

Table 3: RNN Experimental Results

3.2.3 Best Performing Model

Our best model (FFNN Experiment 4) achieved a test accuracy of **59.38%**, training accuracy of **71.01%**, and best validation accuracy of **58.63% (at epoch 6)** with the following configuration: `hidden_dim=32`, `epochs=10`, `lr=0.01`. In spite of showing signs of overfitting, this model gave us the highest test accuracy which implies that moderate overfitting does not have a considerable impact on the model's ability to generalize on unseen data.

3.2.4 Hyperparameter Effects and Key Observations

The model with hidden dimension of **32** had the ideal tradeoff between model size and generalization, and while the larger model (dim=64) had the highest validation accuracy (60.13%), it did not perform well on the test data (46.13%) since it was clearly overfitting. Training the model for 10 epochs resulted in underfitting, and a larger number of epochs (15) resulted in diminishing returns. A learning rate of **0.01** performed the best as lower learning rates (0.005, 0.001) resulted in more overfitting since the model took longer to converge. We also noticed a clear pattern in each experiment that the validation accuracy peaked relatively early (around epochs 3-6) and then decreased. This leads us to believe that early stopping could have been used to prevent overfitting.

4 Analysis

4.1 Learning Curves

We have plotted the learning curve with training loss and validation accuracy for both FFNN and RNN.

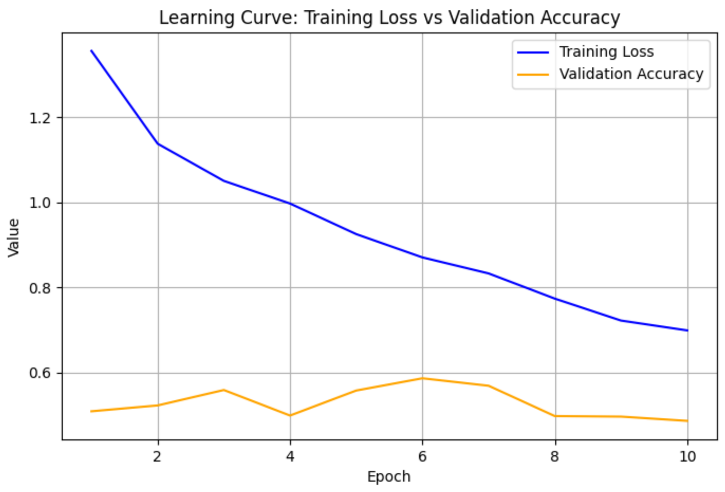


Figure 3: Learning curves - FFNN

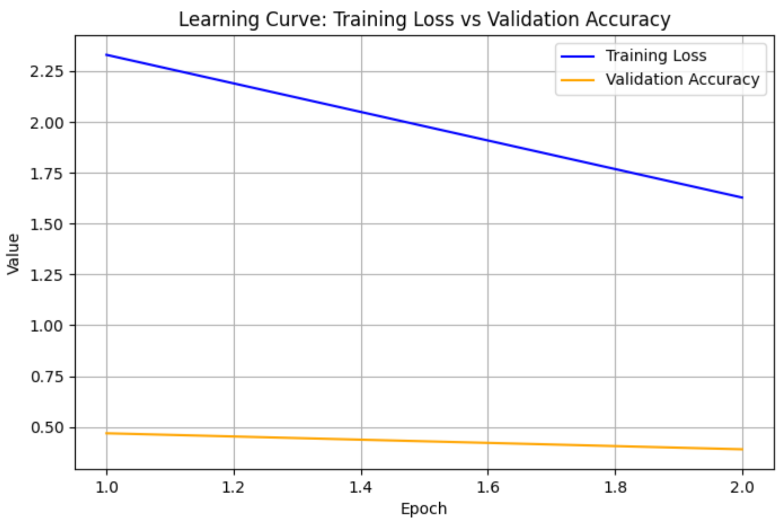


Figure 4: Learning curves - RNN

4.2 Error Analysis

4.2.1 FFNN

We noticed a class imbalance between validation and test sets which would mean the validation accuracy does not accurately predict test performance. The model also overfitting as training progresses. To improve performance, we could implement: (1) early stopping based on validation loss, (2) dropout or L2 regularization to reduce overfitting

4.2.2 RNN

We noticed the following:

- Adjacent star ratings (e.g., 3 vs 4) are commonly confused in sentiment tasks.
- Reviews with mixed sentiment (e.g., 'good food but slow service') tend to be misclassified.
- Very short reviews and those heavy on sarcasm or ambiguity are harder to classify.

Thus, some possible improvements could be:

- Use a bidirectional RNN or LSTM/GRU to better capture long-term dependencies.
- Apply pretrained embeddings (e.g., GloVe, FastText) for richer word semantics.
- Introduce dropout regularization to reduce overfitting.
- Increase the size or diversity of the training dataset.

5 Conclusion and Others

5.1 Individual Member Contribution

Abhirup and Adem worked on the FFNN implementation. Both of them brainstormed to complete the FFNN `forward()` function. Abhirup added additional code to load the test data and calculate the test accuracy. Adem implemented the learning curve plot. They worked together on experimenting with varying hidden_dim, number of epochs, and learning rates.

Buvana and Samiha worked on the RNN implementation. Buvana completed the RNN `forward()` function and added the code for test accuracy calculation, and Samiha developed the parts for computing training loss, validation accuracy, and plotting the learning curves. Together, they experimented with different values of hidden_dim, number of epochs, and learning rates to determine the optimal values.

5.2 Feedback for the Assignment

We feel that this assignment gave us a chance to apply the theory that was presented during the lectures as a hands-on exercise. It cleared our misconceptions and deepened our understanding of neural networks. As such, we feel that the difficulty level of this assignment was appropriate when compared to the material covered in the lectures. We enjoyed working on this assignment and are thankful for our instructor Dr Xinya Du's guidance.