

CS6320 Assignment 2

https://github.com/aankitdas/cs6320_assignment2

Group 22

Aankit Das
Net ID - axd220192

Kedar Rajendra Kadu
Net ID - kxk230024

1 Introduction and Data (5pt)

This project implements sentiment analysis on Yelp reviews using two neural network architectures: Feedforward Neural Network (FFNN) and Recurrent Neural Network (RNN). The task involves predicting star ratings ($y \in 1, 2, 3, 4, 5$) based on review text, essentially treating it as a 5-class classification problem. The dataset used in this project consists of text reviews sourced from Yelp, with star ratings ranging from 1 to 5 as the output labels. The data is formatted as JSON files, each containing the review text along with its corresponding star rating. Additionally, for the RNN implementation, pre-trained word embeddings (provided in 'word_embedding.pkl') are utilized to enhance performance. The preprocessing methods differ between the two models: the FFNN model employs a bag-of-words representation for input processing, while the RNN model leverages the provided word embeddings to handle input sequences effectively.

The data loading was handled by the provided 'load_data' function, which processes the JSON files and creates pairs of (text, rating) for both training and validation sets. The project focuses on implementing neural architectures rather than data preprocessing, with an emphasis on understanding and comparing the performance of FFNN and RNN approaches for sentiment analysis.

2 Implementations (45pt)

2.1 FFNN (20pt)

As introduced in class, for FFNN, it takes as input a fixed-length vector to conduct a feedforward pass. The full implementation should include the following computations:

input : $\mathbf{x} \in \mathbb{R}^d$

hidden layer : $\mathbf{h} \in \mathbb{R}^{|h|}$

output layer : $\mathbf{z} \in \mathbb{R}^{|y|}$ where d is the vocab size, $|h|$ is the hidden layer dimension.

The implementation completed in the 'forward()' function first transforms the input vector through the first linear layer (W_1), applies the ReLU activation function to obtain the hidden layer representation, and then passes this through a second linear layer (W_2) to get the output layer $z \in \mathbb{R}^{|y|}$. Finally, a softmax function is applied to z to obtain y , which represents the probability distribution over the possible classes (ensuring $\sum_{i \in |y|} y[i] = 1$). Figure 1 shows the code snippet that implements the forward network of the FFNN. Additionally, to make it easier

```
def forward(self, input_vector):
    # obtain first hidden layer representation
    hidden_layer = self.activation(self.W1(input_vector))
    # obtain output layer representation
    output_layer = self.W2(hidden_layer)
    # obtain probability dist.
    predicted_vector = self.softmax(output_layer)
    return predicted_vector
```

Figure 1: FFNN forward function.

to evaluate the model after training, an extra argument is added to the parser. called the 'do_eval' argument. The training loop is kept inside the 'do_train' flag, so in order to train the model, the '-do_train' flag must be mentioned as:

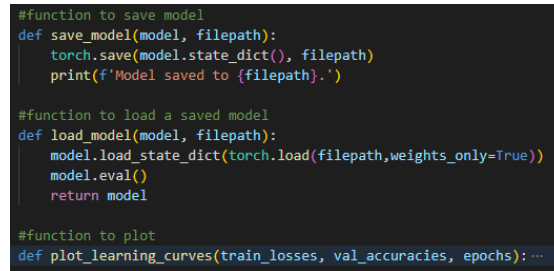
```
python ffnn.py --hidden_dim [hparam] --epochs [hparam]
--train_data [train data path] --val_data [val data path] --do_train
```

After training is done, the model is automatically saved to the 'saved_models' folder. While evaluating, we have to make sure to add the correct saved model name. In the command line, we have to enable the '--do_eval' flag like:

```
python ffnn.py --hidden_dim [hparam] --epochs [hparam]
--train_data [train data path] --val_data [val data path]
--test_data [test data path] --do_eval
```

The 'compute_Loss()' function calculates the negative log-likelihood loss between the predicted and gold labels. The class works with data processing functions that handle bag-of-words vectorization for the input text reviews. The implementation uses PyTorch's neural network modules (nn.Linear, nn.ReLU, nn.LogSoftmax) and is trained using SGD optimizer with momentum, processing data in minibatches of size 16 (sometimes varied to 32), with early stopping based on validation accuracy to prevent overfitting.

Three new functions have been added for evaluating the model, they are: 'save_model', 'load_model', and 'plot_learning_curves'. As the names suggest, these are for saving the best model after training, loading the model, and plotting the required graphs to note the training progress. Figure 2 shows the code snippet of these functions in the script.



```
#function to save model
def save_model(model, filepath):
    torch.save(model.state_dict(), filepath)
    print(f'Model saved to {filepath}.')

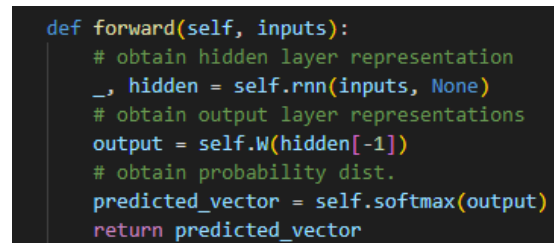
#function to load a saved model
def load_model(model, filepath):
    model.load_state_dict(torch.load(filepath, weights_only=True))
    model.eval()
    return model

#function to plot
def plot_learning_curves(train_losses, val_accuracies, epochs):...
```

Figure 2: Added helper functions.

2.2 RNN (25pt)

In the RNN implementation, the core logic focuses on the RNN layer and its ability to handle sequences of input data, which is fundamentally different from the FFNN approach. Figure 3 shows the code snippet that implements the forward function for RNN. Unlike FFNN, where inputs are passed through the network all at once, RNNs process one element of the sequence at a time and update the hidden state. Also, RNN uses the hyperbolic tangent activation function for its hidden states as specified by the command `nonlinearity='tanh'`. In an RNN, the hidden states are updated at each time step, and the last hidden state is used for classification. This is different from FFNN, where the entire input is processed in parallel without sequential dependencies.



```
def forward(self, inputs):
    # obtain hidden layer representation
    _, hidden = self.rnn(inputs, None)
    # obtain output layer representations
    output = self.W(hidden[-1])
    # obtain probability dist.
    predicted_vector = self.softmax(output)
    return predicted_vector
```

Figure 3: RNN forward function.

Additionally, the default early stopping condition was very simple and therefore, was getting activated very frequently. Therefore, we implemented a new stopping criteria with a patience value, as shown in figure 4.

```

#early stopping condition
if validation_accuracy > best_validation_accuracy:
    best_validation_accuracy = validation_accuracy
    epochs_without_improvement = 0 #reset the counter if we have improved
else:
    epochs_without_improvement += 1

if epochs_without_improvement >= patience:
    stopping_condition = True
    print(f"Training stopped after {epoch + 1} epochs due to no improvement in validation accuracy.")
    print(f"Best validation accuracy achieved: {best_validation_accuracy}")
else:
    last_train_accuracy = training_accuracy
    last_validation_accuracy = validation_accuracy

```

Figure 4: New stopping criteria.

3 Experiments and Results (45pt)

Evaluations (15pt) We implemented classification accuracy as the main metric, calculated as the number of correct predictions divided by the total number of predictions. A prediction is considered correct when the predicted star rating matches the gold (true) label. The model is evaluated on a separate validation dataset after each training epoch. The model of the model is changed to evaluation mode by using the command `model.eval()` and predictions are made without gradient computation by using the command `torch.no_grad()`. We followed this same approach for both FNN and RNN. Figure 5 shows the code snippet that performs the evaluation process.

```

for input_vector, gold_label in test_data:
    predicted_vector = model(input_vector)
    predicted_label = torch.argmax(predicted_vector)
    correct += int(predicted_label == gold_label)
    total += 1
print(f"Test accuracy: {correct / total}")

```

Figure 5: Evaluation process.

Additionally, we kept track of all the training losses and validation accuracies for each epoch inside the training cycle. Figure 6 shows the additional code that performs this task for plotting the graphs.

```

avg_epoch_loss = epoch_loss / (N // minibatch_size)
train_losses.append(avg_epoch_loss)

val_accuracy = correct / total
val_accuracies.append(val_accuracy)

total_time = time.time() - start_time
print(f"\nTotal training time: {total_time:.2f} seconds")

#save model after training
save_path = 'saved_model/ffnn_model.pt'
save_model(model, save_path)
plot_learning_curves(train_losses, val_accuracies, args.epochs)

```

Figure 6: Tracking the losses and accuracies every cycle.

Results (30pt) To optimize the performance of our models, we experimented with different hidden unit sizes. The variations aimed to analyze the effect of hidden layer sizes on model accuracy, training time, and convergence. For FFNN, we tested with hidden units 128 and 512. The validation accuracies were almost similar while the time taken to train was huge for the model with 512 hidden layers. For RNN, we tried with a very low hidden layer of 64 first, which resulted in a bad validation accuracy which was expected. We then bumped up the hidden layers to 512 along with changing the learning rate to 0.001 which improved the performance. Table 1 summarizes the validation accuracy and training time for each model variation.

For both FFNN and RNN models, increasing the hidden unit led to an additional improvement in performance. However, they came at the cost of increased training time. Practically, choosing 128 hidden units balanced both accuracy and training efficiency for our models.

Model	Hidden Units	Validation Accuracy	Training Time (mins)
FFNN-A	128	0.62	90
FFNN-B	512	0.59	180
RNN-A	64	0.49	30
RNN-B	512	0.55	64

Table 1: Performance summary of model variations with different hidden unit sizes.

4 Analysis (bonus: 5pt)

Plots The project implements two neural network architectures for sentiment analysis on Yelp reviews: FFNN and RNN. The learning curves demonstrate the models' training progression over epochs. In figure 7, we plot the training loss and validation accuracy with respect to the number of epochs. The number of hidden layers was 512 and we ran for 50 epochs. We observe an initial decrease in training loss from 1.00 to 0.09 in the first 20 epochs, followed by gradual stabilization, while validation accuracy improved to 62% but then settled at 59%. Similarly, figure 8 shows the plots for 128 hidden layers and 25 epochs. The general trend of the plots is similar, although the latter took much less time to train compared to the former.

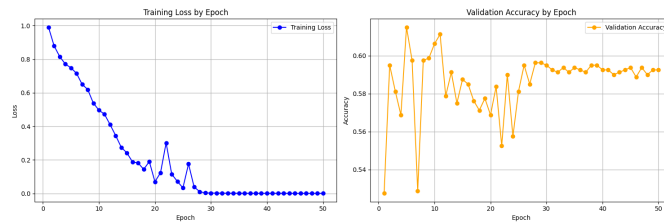


Figure 7: FFNN plots for 512 hidden layers and 50 epochs.

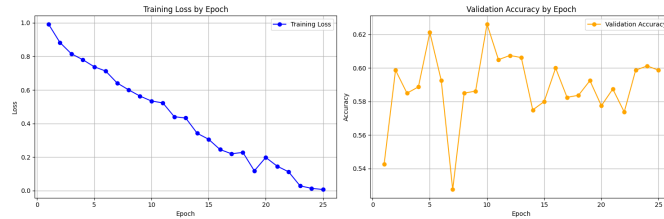


Figure 8: FFNN plots for 128 hidden layers and 25 epochs.

On the other hand, the RNN's ability to process sequential information through its hidden states provides an advantage over the FFNN's bag-of-words approach, particularly in capturing word order and context in the reviews. However, our experimental results showed that the RNN performed worse than the simpler FFNN model. This counter-intuitive result could be attributed to several factors. The single-layer RNN architecture (with 'numOfLayer=1') may be overly simplistic for capturing the complex sequential patterns present in the review data, potentially limiting its ability to model dependencies across longer sequences effectively. Additionally, the method of summing the output vectors in the RNN might dilute important sequential information, as it can obscure the significance of individual time steps and their contributions to the overall context. Another potential limitation is that the use of fixed pre-trained embeddings may not be fully aligned with the specific sentiment classification task, which could lead to suboptimal performance since these embeddings are not fine-tuned to adapt to the nuances of the dataset at hand.

Potential improvements could include: implementing attention mechanisms to better weight important parts of the sequence, using a bidirectional RNN to capture context from both directions, fine-tuning the word embeddings during training. Figure 9 and figure 10 shows the two plots for hidden layer 64 and 512 respectively.

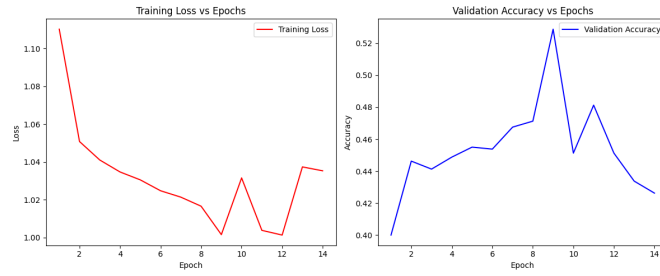


Figure 9: FFNN plots for 128 hidden layers and 25 epochs.

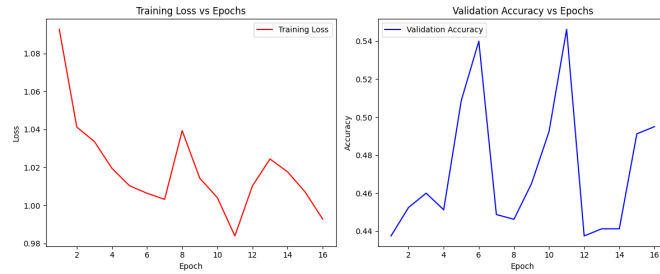


Figure 10: FFNN plots for 128 hidden layers and 25 epochs.

Error Analysis During experimentation, we initially trained the model using the Adam optimizer with a learning rate of 0.01 which was provided by default. However, the validation accuracy remained suboptimal, hovering around 0.4. Figure 11 shows the plot which shows the suboptimal accuracy.

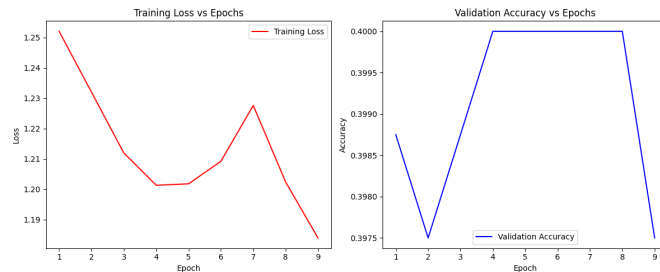


Figure 11: Erroneous RNN plots showing sub-optimal results.

To address this, I switched the optimizer to Stochastic Gradient Descent (SGD) and decreased the learning rate to 0.001. Additionally, I extended the patience for early stopping from 5 to 10 epochs. These adjustments led to a noticeable improvement in the model's validation accuracy, suggesting that the SGD optimizer and a higher learning rate helped the model converge more effectively. The extended patience allowed the model more time to improve, ultimately yielding better generalization on the validation set.

5 Conclusion and Others (5pt)

The first part of the assignment, that is, manipulating the FFNN, is done by Aankit Das. The second part of exploring RNN, is done by Kedar Kadu. The evaluation part is done by both Aankit Das and Kedar Kadu so that we both have a comprehensive look at each of the models.

The implementation of both the FFNN and RNN models provided valuable hands-on experience with neural networks for text classification. Approximately 20-30 hours total, have been spent understanding the code structure, implementing the forward passes, debugging, training, and analyzing results. The code structure helped a lot in understanding the flow of the project making the difficulty level moderate.