CS4375 Assignment 2

https://github.com/arihu/CS4375-Assignment-2

Group 9 Ari Hu axh200041 Daniel Morandi drm210004 Daniel Nguyen dnn200003

Joel Varghese jjv200001

1 Introduction and Data (5pt)

For this project, we are filling the forward function for FFNN and RNN models that would predict Yelp! review scores. The main experiments are testing the FFNN and RNN models with different hidden dimension sizes, and the results are a better accuracy of 65% with the FFNN model at 32 dimensions, with the average RNN accuracy at around 30%. The data we used were training, validation, and test sets provided in the Data Embeddings folder.

The task is sentiment analysis where given a set of training, validation, and test set containing Yelp! reviews associated with a rating(stars) of y = {1,2,3,4,5}, we are to predict y. The example count for each data set: test.json includes 800 reviews, training.json 16,000 reviews, validation.json 800 reviews. All JSON data sets includes a list of reviews where each review has a text field and a stars field.

For loading and vectorizing the raw data for the FFNN, the load_data function is called to convert the training and validating json to a training and validating list containing pair of text and stars. The make_vocab function then creates a set of the vocabulary given this training data list. This vocab set is then passed through make_indices function to create vocab with an unknown token $\langle UNK \rangle$, a word2index which maps token/word to index, and index2word to map index to token/word. Next, we vectorize both the training and validation data with the word2index mappings by creating pairs of inputs and results. After these steps, the training and validation data is shuffled and batched to be able to be trained by the model.

For loading and vectorizing the raw data for RNN, the load_data function is called to convert the training and validating json to a training and validating list containing pair of text and stars. Then the given word_embedding.pkl file is loaded to word_embedding, this would be used to convert input words into vectors before feeding them into the RNN by having each word mapped to a vector representation. After these steps, the training and validation data is shuffle and batched, punctuation is removed, vector word representation is initialized by locating words in word_embedding, and the vector is transformed to have the shape [batch size, sequence length, input dimension] to be able to be trained by the model

2 Implementations (45pt)

2.1 FFNN (20pt)

The Feedforward Neural Network (FFNN) implementation in FFNN.py was designed for text classification, such as sentiment analysis. The focus was on completing the neural network architecture, including the choice of activation functions (ReLU for the hidden layer, LogSoftmax for the output layer) and the use of the NLLLoss function for training. To accomplish this, libraries like pytorch for nn were used to help do the calculations for Relu, NLLLoss, and more. Other libraries include data manipulation like numpy and random.

Throughout the development process, various debugging techniques were employed, such as print statements for tracking tensor shapes and intermediate outputs, and comparing training and

validation accuracy to detect over fitting. Many of the values were also written into files, so they could be compared to make sure the correct trend were taking place.

The FFNN's architecture features an input layer, a hidden layer with ReLU activation for non-linearity, and an output layer utilizing LogSoftmax for generating class probabilities. Stochastic Gradient Descent (SGD) with momentum is used for optimization, providing efficient parameter updates and aiding convergence. Mini-batch training (batch size of 16) was employed for computational efficiency and improved training dynamics. Furthermore, fixing random seeds ensures result reproducibility, which is crucial for debugging and comparing different runs.

```
def forward(self, input_vector):
      # [to fill] obtain first hidden layer representation
      hidden_representation = self.activation(self.W1(input_vector))
      # [to fill] obtain output layer representation
      output_representation = self.W2(hidden_representation)
      # [to fill] obtain probability dist.
      predicted_vector = self.softmax(output_representation)
      return predicted_vector
```

Listing 1: FFNN forward function.

The only non logging changes to the code are shown in listing 1. The forward method within the FFNN class defines the core process of how a text input is transformed into a prediction by the NN. It starts by taking an input vector ($inputvector$), which represents the text data. This input is passed through the first linear layer ($self.W1$) and an activation function ($self.activation$) to obtain the hidden layer representation. This hidden representation captures non linear relationships within the input. Sequentially, the hidden representation is fed into the ouput layer ($self.W2$) to produce the raw output score. Finally, the softmax function ($self.softmax$) transforms these scores into a probability distribution over the possible classes. This probability distribution represents the model's prediction for the input text. The final change is to write the output to result/test.out.

2.2 RNN (25pt)

The Recurrent Neural Network (RNN) implementation in FFNN.py is a more sophisticated model for sentiment analysis. The RNN relies on a tanh activation function for the hidden layer, LogSoftmax to compute probability distribution for the outputs, and NLLLoss function for training the model. The required libraries that were used was pytorch for faster calculations and usage of tensors, random to shuffle the data, pickle to read the embedding file, and numpy.

Throughout implementing the forward function, debugging techniques like printing the shape, print statements of the outputs for self.RNN and the output tensors, and checking the validation and training accuracy for overfitting was used.

The RNN's architecture by Pytorch is the Multi-Layer Elman RNN. In this architecture, the hidden layer is connected to the input and output layers and the activation(tanh) are used across multiple time steps. The benefit is that it can use context from previous hidden layers. For the optimizer, unlike the FFNN that uses SGD, the RNN uses ADAM, an optimization algorithm that uses momentum. Another difference between the FFNN, is that the starter code does not utilize epoch args. The rnn.py stops the training loop when the validation accuracy is less than the last validation accuracy and the training accuracy is greater than the last training accuracy. This is meant to stop the RNN from over training and overfitting.

The changes made to rnn.py is adding the forward function seen in listing 2. The forward function works like the FFNN but the difference is that it takes inputs rather than a single input_vector. The inputs variable is a packed sequence tensor which includes information [batch size, sequence length, input dimension]. To gather information of the hidden vectors, self.rnn(inputs) was used to return two parameters, a tensor of all hidden layers and a tensor of the last hidden layer. Because

```python
def forward(self, inputs):
        # [to fill] obtain hidden layer representation
        rnn_output, _ = self.rnn(inputs)
        # [to fill] obtain output layer representations
        output = self.W(rnn_output)
        # [to fill] sum over output
        summed_output = torch.sum(output, dim=0)
        # [to fill] obtain probability dist.
        predicted_vector = self.softmax(summed_output)
        return predicted_vector
```

Listing 2: RNN forward function, and minor changes.

the assignment required us to sum up the vectors for the output layer, the first parameter was used and assigned to rnn_output. After we have our list of hidden layers, self.W(rnn_output) was used to generate a list of outputs called output given the list of hidden layers. After this we sum up each tensor of the output with torch.sum to get our summed_output. Finally, the self.softmax function transforms the summed_output to a probability distribution that represent the prediction of the input text. The final change is to write the output to result/test_RNN.out.

3 Experiments and Results (25pt)

Evaluations (5pt) In the context of neural networks for tasks like sentiment analysis, it is essential to focus on specific metrics to evaluate the models. Hidden dimensions, Training Accuracy, and Validation Accuracy are all important metrics to track when evaluating our models. Hidden dimensions refer to the size of the hidden layers within neural networks. In the context of FFNNs and RNNs it is a measure of the layer's capacity to capture information and patterns from the input data. The size of this layer has a direct impact on the model's complexity and ability to learn from the data. A larger number of hidden units allows the model to learn more complex patterns but also increases the risk of overfitting. Conversely, a smaller number of hidden units might lead to underfitting. Next, training accuracy is the percentage of correct predictions made by the model on the training dataset. This is a direct measure of how well the model has learned to classify or predict outcomes based on the data it was trained on. High training accuracy can be a good indicator that the model is effectively learning and capturing relationships between the input features and target outputs. However, this is not enough to determine a model's effectiveness. This is because a model can achieve high training accuracy through overfitting which is essentially memorizing the training data without truly learning the underlying patterns in it that are needed to generalize unseen data. This is where validation accuracy comes into the picture. Validation accuracy is the percentage of correct predictions made by the model on a separate dataset not used during training, known as the validation set. High validation accuracy paired with high training accuracy can fully suggest that the model can effectively make accurate predictions on unseen data. A significant drop-off between the training and validation accuracies typically indicates overfitting. Finding the optimal size of hidden layers while monitoring overfitting will eventually lead to optimal model performance. For both FFNN and RNN we have both Training and Validation Accuracy as our main metrics. For FFNN we are adjusting the epoch count until it overfits, and for RNN we are letting the loop continue to run until it overfits.

Results (20pt) The FFNN models each demonstrated an improvement in validation accuracy with increasing hidden dimensions up to a point, with the highest accuracy being achieved at a hidden dimension size of 32. After this, the accuracy slightly decreases indicating overfitting with an overly complex model. The number of epochs required to reach the best validation accuracy varied greatly, suggesting that the networks might need more training time but do not guarantee better performance on the validation set. The RNN models showed a clear trend where increasing the hidden dimension size didn't necessarily lead to better validation accuracy. The

highest validation accuracy was achieved with the smallest hidden dimension of 10, suggesting that a simpler model might be the most effective for this specific task. The best accuracy pairings for the RNN model were nearly identical for the hidden dimension sizes of 10 and 32. Early stopping was an important training strategy in preventing overfitting for the RNN model. In both models, it seemed like the largest hidden dimension size of 128 was too complex and did not yield the best results. For the RNN, the smaller hidden dimension sizes of 10 and 32 seemed similarly effective, potentially due to the nature of the data being more suited towards simpler representations. For the FFNN, the moderate hidden dimension size of 32 clearly seemed the most effective and offered the best trade-off between complexity and performance. In this particular example, the FFNN models consistently outperformed the recurrent neural network RNN. A significant factor contributing to this is that FFNNs are generally better suited for processing non-sequential data. In the context of Yelp reviews, the sentiment conveyed by a single word often outweighed that of an entire phrase. Hence, there was less reliance on retaining memory across layers, so FFNN helped minimized noise and ultimately enhanced accuracy within the network.

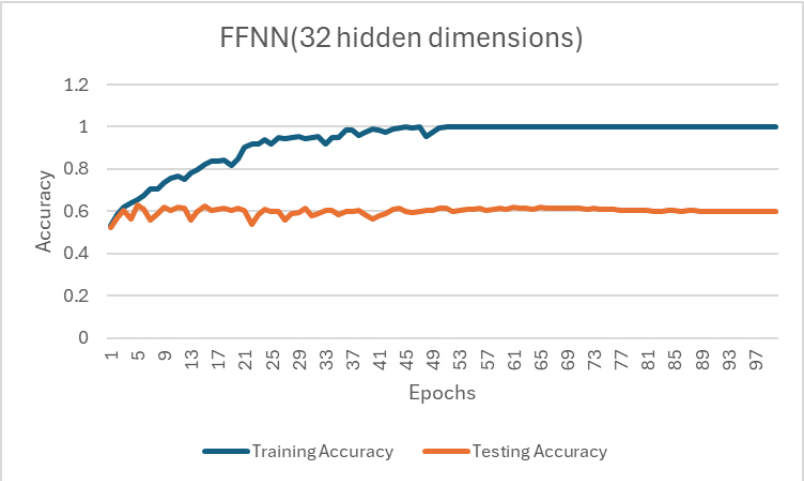
Best Accuracy given Model and arguments: python rnn.py/ffnn.py –hidden_dim y –epochs x –train_data ./training.json –val_data ./validation.json			
Model	Epoch(x)	Hidden Dim(y)	Best Accuracy (Epoch, Training, Validation)
FFNN	100	10	2, 0.5645, 0.554
FFNN	100	32	5, 0.655, 0.631
FFNN	100	128	2, 0.586, 0.59
RNN	N/A*	10	2, 0.378, 0.38
RNN	N/A*	32	6, 0.3855, 0.39
RNN	N/A*	128	3, 0.271, 0.311

* RNN runs until it overfits so epoch arguments are ignored

(Extra Bonus) Tried a variation of the RNN model by running the model with hidden dimensions 10, and with the ReLU activation function instead of the tanh activation function by default. The ReLU function actually ran for more epochs with higher accuracy. The tanh activation function maxed out at 2 epochs while the ReLU activation maxed out at 7 epochs. A hypothesis for this behavior is that ReLU doesn't deal with vanishing gradient problem as the derivatives are either 0 or 1.

Using ReLU activation function for RNN:			
Model	Epoch(x)	Hidden Dim(y)	Best Accuracy (Epoch, Training, Validation)
RNN	N/A*	10	5, 0.4525, 0.4465

4 Analysis (20pt)



Accuracy of FFNN network with 32 hidden dimensions

The main error example that can be clearly seen is a classic case of overfitting. The training accuracy increases consistently and reaches perfection or 100 percent by the 80th epoch. However, the validation accuracy peaks at about 61.875 percent and then fluctuates without significant improvements, suggesting a disconnect between the model's performance on the training and validation datasets. With a high number of epochs (100 in our case), the model becomes increasingly confident in its training data predictions which is why it eventually reaches perfect accuracy. This suggests that the model might be memorizing the training data rather than generalizing features. To improve this model, we might look towards implementing more regularization techniques and focusing on model architecture tuning. For example, dropout regularization could help mitigate overfitting and should prevent the model from placing too much importance on any single feature or instance, which in theory will help it learn more general patterns. Also, experimenting with different numbers of hidden layers and altering activation functions could potentially yield improved results. The above suggestion was proved to be helpful in our experiments with the RNN model. We demonstrated that modifying the activation function and adjusting the hidden dimensions significantly enhanced the model's performance and resulted in a more effective learning process.

(Extra Bonus) Other discussions is why the RNN accuracy was lower than the FFNN accuracy. A hypothesis we had is that it was easier for the neural network to find patterns in negative words and positive words from one pass with the FFNN rather than finding sequential patterns among reviews with the RNN. The RNN also suffers from vanishing gradient, which can happen if the sentence is long. This can cause the RNN to not recognize certain sequential dependencies. The RNN seems more suited if the reviews were all made by the same account, where sequential patterns are more important.

5 Conclusion and Others (5pt)

- Individual member contribution.

  Ari Hu - Added Introduction and Data section, RNN forward function and RNN implementation section, and Extra Bonus for results and analysis section.

  Daniel Morandi - Added the FFNN forward function, and FFNN implementation section. Provided output for both FFNN and RNN.

  Daniel Nguyen - Create Analysis Graph, proof-read the document, modified the Results section, and added feedback.

  Joel Varghese - Added Evaluations section, RNN/FFNN Accuracy Results based on output provided by team, results summary and error analysis.

- Feedback for the assignment. e.g., time spent, difficulty, and how we can improve.

  The difficulty and time required for the assignment is fair. The assignment was a great way to introduce students to generate neural networks. However, the project did not seem to be organized in a way that allowed for group work. A good way to improve the assignment is make it so that the parts can be assigned to each individual where and later consolidated for the final report.