CS6375 Assignment 1

https://github.com/sy-hong/CS6375-Assignment-1.git

Shi Yin Hong
SXH230097

1 Introduction and Data (5pt)

The objective of the project is to complete the *forward* function implementation of a feedforward neural network (FFNN) and recurrent neural network (RNN) for a 5-class sentiment analysis task over restaurant reviews in the Yelp dataset. Specifically, given training, validation, and test sets with restaurant reviews labeled with a rating $y \in Y = \{1, 2, 3, 4, 5\}$, the goal is to predict y. Table 1 shows the statistics of Yelp reviews.

Experiments are conducted to evaluate the classification performance of FFNN and RNN using validation accuracy as the main evaluation metric. Hyperparameter tests are also conducted by tuning the hidden dimension of the models. The results support that for the provided datasets, both models tend to overfit. Furthermore, since different classes of reviews are missing in adopted data splits, both models could not adequately perform the sentiment analysis classification task.

Data	Reviews	Avg. Length	1-Star	2-Star	3-Star	4-Star	5-Star
Training	8000	141.2	3200	3200	1600	0	0
Validation	800	140.4	320	320	160	0	0
Testing	800	109.8	0	0	160	320	320

Table 1: Yelps dataset statistics

2 Implementation (45pt)

2.1 Feedforward Neural Network (FFNN) (20pt)

Overall, the implementation consists of data preparation followed by training and validation with the FFNN.

Data Preparation: A vocabulary is built based on the training and validation set. Unknown words are handled. The word2index dictionary is created to map the token to its index. Another dictionary, index2word, invert word-to-index, data are converted to vectorized format to be processed by FFNN. Before training starts, an instance of the FFNN is set to be the model.

Training and Validation: Stochastic gradient descent (SGD) is set to be the optimizer in error gradient computation based on the training instances to minimize loss. The learning rate is set to 0.01, which determines the step size at which the SDG performs the optimization. A momentum of 0.9 helps to increase the pace at which the optimization is performed in the correct gradient direction. Training is performed on the training set before and validating on the validation set with minibatches of size 16 on randomly shuffled data. The loss and accuracy are computed per epoch for both training and validation.

***forward* function:** The *forward* function (Figure 1) integrates components of the FFNN defined in the constructor. To obtain the hidden layer representation, the vector representation of reviews undergoes the first linear transformation, *W1*, reducing its original dimension to the pre-defined hidden state. Then, the ReLU activation function introduces non-linearity. This hidden layer representation undergoes the second layer of linear transformation, *W2*, resulting in the output layer representation. Finally, the softmax function transforms the resulting vector representation into a 1-D tensor with probability distribution of the five classes.

```python
def forward(self, input_vector):
    # [to fill] obtain first hidden layer representation
    hidden = self.W1(input_vector)          # pass through 1st linear layer
    # print(">> hidden: ", hidden.size())     # hidden:  torch.Size([32]) - torch.Size([hidden])
    hidden = self.activation(hidden)        # pass the hidden layer through the activation function
    # print(">> hidden: ", hidden.size())     # hidden:  torch.Size([32]) - torch.Size([hidden])

    # [to fill] obtain output layer representation
    output_layer = self.W2(hidden)          # pass through the 2nd linear layer
    # print(">> output_layer: ", output_layer.size()) # output_layer: torch.Size([5]) - torch.Size([hidden])

    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(output_layer)  # pass through softmax
    # print(">> predicted_vector: ", predicted_vector.size()) # predicted_vector: torch.Size([5]) - torch.Size([hidden])

    return predicted_vector
```

Figure 1: Forward function of the feedforward neural network (FFNN).

2.2 RNN (25pt)

Similar to the FFNN, the implementation consists of data preparation followed by training and validation.

Differences to FFNN: Other than the model architecture, the first major difference is that word embedding is adopted in vectorizing data. Further, the implementation adopts the Adams optimizer. Early stopping is applied to avoid overfitting, which is further explained in Section 3.1.

***forward* function:** Similar to the FFNN, the constructor of the RNN function specifies the elements to be integrated into the *forward* function (Figure 2). Based on the PyTorch documentation [1], Figure 2 shows an implementation that begins with the explicit declaration of the initial hidden layer, h_0, that accepts the number of layers, batch size, and hidden dimension as parameters. Next, h_0 and inputs in the form of tensors are passed in the RNN layer to obtain the hidden layer representation. The results consist of two parts: the output tensor (*output*) contains output features (*hidden*) from RNN's last layer and each element's final hidden state. The output layer is then acquired by passing *output* through a linear layer, W, to transform the hidden dimension to the five, or the number of classes. The output is summed over the first dimension that consists of the output features, and the squeeze operation further eliminates the batch dimension for preparing the tensor dimension to be accepted by the softmax function to the probability distribution. Hence, the dimension parameter for softmax in the constructor is adapted to -1 as the summed output changes to a 1-D tensor.

```python
def forward(self, inputs):
    # [to fill] obtain hidden layer representation (https://pytorch.org/docs/stable/generated/torch.nn.RNN.html)
    h_0 = torch.zeros(self.numOfLayer, inputs.size(1), self.h) # explicitly initialize h_0 -- optional based on the documentation
    output, hidden = self.rnn(inputs, h_0)                  # pass inputs (seq length, batch, hidden) & h_0 to the RNN layer
    # print(">> output: ", output.size())                     # output:   (sample) torch.Size([29, 1, 16])  - torch.Size([output features, batch, class #])
    # print(">> hidden: ", hidden.size())                     # hidden:   torch.Size([1, 1, hidden]) - torch.Size([final hidden state, batch, hidden])

    # [to fill] obtain output layer representations
    output_layer = self.W(output)                           # perform the linear transformation on the final hidden state
    # print(">> output_layer: ", output_layer.size())         # output_layer:  torch.Size([29, 1, 5]) - torch.Size([output features, batch, class #])

    # [to fill] sum over output
    # print(">> output_layer.sum(0): ", output_layer.sum(0).size())  # torch.Size([1, 5]) - sum over the layer dimension
    sum_output = output_layer.sum(0).squeeze(0)             # squeeze the batch dimension out --> need to change line 26's dim from 1 to -1
    # print(">> sum_output: ", sum_output.size())             # sum_output:  torch.Size([5]) - torch.Size([class #])

    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(sum_output)             # pass sum_output to the softmax to get probability distribution
    # print(">> predicted_vector: ", predicted_vector.size())   # predicted_vector:  torch.Size([5]) - torch.Size([class #])

    return predicted_vector
```

Figure 2: Forward function of the recurrent neural network (RNN).

3 Experiments and Results (45pt)

3.1 Evaluations (15pt)

Codes are adapted to log all print statements to track evaluations. The best model should achieve the *highest validation accuracy* as the main evaluation metric.

FFNN: Batchwise evaluation (size = 16) is performed during training and validation on randomly shuffled data. The training and validation times per epoch are further recorded. Batchwise evaluations are performed to calculate the negative log-likelihood loss (accumulative loss/batch size) and accuracy (correct prediction based on comparison to the gold label/total in batch) during training and validation per epoch. Negative log-likelihood loss is adopted during training

[1] https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

and validation.

RNN: Batchwise evaluation (size = 16) on randomly shuffled data is only adopted during training, where the model's prediction is compared against the gold label in obtaining the training accuracy. The negative log-likelihood loss is also only employed during training, where the loss of each sample accumulates during each iteration of batch-wise evaluation. The average loss is calculated after each batch. Validation is conducted on randomly shuffled data, where predicted instances are compared against gold labels to obtain validation accuracy. Early stopping is applied to avoid overfitting. Negative log-likelihood loss is only adopted during training. The evaluation terminates when the validation accuracy of the current epoch is less than the validation accuracy of the previous epoch while the current epoch's training accuracy is greater than the training accuracy of the previous epoch. Then, the best validation accuracy is reported.

3.2 Results (30pt)

3.2.1 FFNN

Figure 3 displays the training accuracy, validation accuracy, training loss, and validation loss of FFNN adopting hidden dimension sizes of 8 ($FFNN_8$) and 16 ($FFNN_{16}$), respectively. As shown, $FFNN_{16}$ demonstrates better performance, although both models exhibit signs of overfitting. The validation accuracy of $FFNN_8$ revolves around the range of 0.52 to 0.61 after the initial increase up to 0.59625 in the first three epochs, while the training accuracy demonstrates a positive trend. The training accuracy and validation accuracy of $FFNN_{16}$ also follow the same pattern, reaching a validation accuracy of 0.6125 by epoch three before oscillating in the range of 0.49625 to 0.62375 in later epochs. The training losses and validation losses of the two model variations further validate the issue of overfitting, as the validation losses are repetitively greater than their respective training losses after epoch 14. One approach to alleviate the issue is to adopt a well-representative dataset with more samples.

Since the provided implementation further records the training and validation times per epoch of the FFNN, Table 2 summarizes such timing performance of $FFNN_8$ and $FFNN_{16}$. The results indicate that as the size of the hidden dimension increases from 8 to 16, the lower and upper bounds of the training time per epoch increase by approximately 2.5 and 3 times, respectively. The validation times per epoch elevate by approximately 2 times for the lower bound and 2.4 times for the upper bound. The significant timing differences between the training and validation times for both models are how the training stage involves more training samples, resulting in longer training time per epoch.

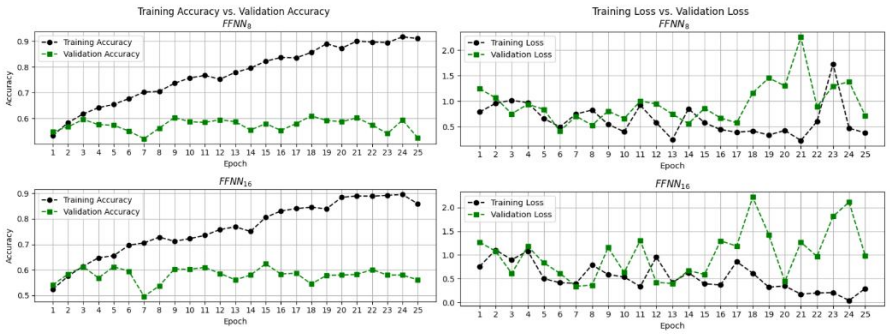

Figure 3: Performance of $FFNN$ varied by the size of hidden dimension.

Model	Training Time (s)	Validation Time (s)
$FFNN_8$	17.95 - 21.91	0.29 - 0.60
$FFNN_{16}$	44.44 - 64.30	0.58 - 1.46

Table 2: The ranges of training and validation times per epoch of $FFNN_8$ and $FFNN_{16}$ trained with 25 epochs.

3.2.2 RNN

Table 2 displays the experimental results of the RNN model implemented with early stopping. Hidden dimension sizes of $(8, 16, 32, 64)$ are considered. For all model variants, the training and validation terminate before the training epoch hyperparameter (e.g., 10) to prevent the model from overfitting during training. As shown, RRN_8 terminates upon the early stopping epoch (ESE) of 2, achieving the best validation accuracy of 0.565 at epoch 1 with a respective training accuracy of 0.476. Although randomness is involved, the experimental results suggest that increasing the size of the hidden dimension does not necessarily lead to better performance.

Model	ESE	Training Accuracy Before ESE	Best Validation Accuracy
RNN_8	2	0.476	**0.565**
RNN_{16}	3	0.501	0.501
RNN_{32}	2	0.462	0.521
RNN_{64}	5	0.451	0.515

Table 3: Performance of $RNN_{\text{hidden dimension}}$ models upon varying the size of hidden dimension. ESE indicates the early stopping epoch. The best validation accuracy is in bold.

4 Analysis (10pt)

4.1 Learning Curve of the Best System

The experimental results indicate that $FFNN_{16}$ is the best system. Figure 4 shows the learning curve of $FFNN_{16}$. As shown, the training loss decreases upon as the validation performance stabilizes.

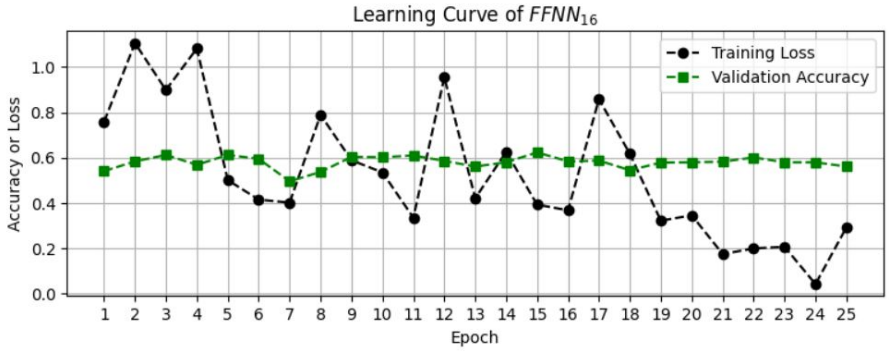

Figure 4: Learning curve of $FFNN_{16}$.

4.2 Error Analysis

The major problem with the current system traces to its dataset. As shown in Table 1, the training and validation sets only contain reviews of 1 to 3 stars, while the testing set contains reviews ranging from 3 to 5 stars. Training and validation are performed on unrepresentative data. Therefore, if testing were to be performed, both the FFNN and RNN models are unlikely to adequately classify these reviews when given any 4-star and 5-star reviews in the testing set. The training, validation, and testing sets need to be balanced on all classes of reviews in their adopted splits to alleviate the issue.

5 Conclusion and Others (5pt)

- **Member Contribution:** Shi Yin Hong*
- **Feedback:** The project took over a week to complete with doable difficulty. I can improve by getting better at understanding the provided hints (e.g., irrelevant comments, spurious parts, line 34 in the original rnn.py template) and spending more time going in-depth with the analysis.