CS6320 Assignment 2

https://github.com/nishd8/rnn-fnnn

Group21 Nishad Raisinghani NXR200053 Ishwari Joshi IGJ180000

1 Introduction and Data

In this project, a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN) are implemented for performing a 5-class Sentiment Analysis task. The data used is in 3 json files: training.json, validation.json, and test.json. The json files have two key-value pairs for each review. Key-value pairs are "text": "text from review" and "stars": rating from {1.0, 2.0, 3.0, 4.0, 5.0}. The main experiment is to train the models (FFNN and RNN) with different hyperparameters (hidden layer dimension and number of epochs) and evaluate the models based on accuracy. There are 8000 examples in the training set, 800 examples in the validation set, and 800 examples in the test set. The data given is used as is. No further preprocessing is done. The data is not lowercased, and punctuation is not removed. The data loading is done in the load_data function where the data is loaded and put into lists in which each element is (the text split by whitespace, rating minus 1). The rating minus 1 is used to align with Python's zero-based indexing.

2 Implementations

2.1 FFNN

The forward computation part in `forward()` is filled in to implement a 1 hidden layer FFNN with ReLU activation. The `__init__(self)` function has two parameters: the input dimension and the hidden layer dimension. Layers and parameters are defined in the `__init__(self)` function. `Linear(n,m)` is a PyTorch function that creates a single layer feedforward network with n inputs and m outputs. There is 1 Linear hidden layer W1 in which Linear has two parameters: the input dimension (size of vocabulary) and the hidden unit dimension. There is a ReLU activation function that uses the `ReLU()` function. The Linear output layer W2 has a dimension of 5 (for 5 possible ratings). For W2, Linear has two parameters: the hidden unit dimension and the output layer dimension.

```
def forward(self, input_vector):
    # [to fill] obtain first hidden layer representation
    first_hidden_layer_rep = self.W1(input_vector)

    activation_op = self.activation(first_hidden_layer_rep)

    # [to fill] obtain output layer representation
    output_layer_rep = self.W2(activation_op)

    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(output_layer_rep)

    return predicted_vector
```

Figure 1: Implementation of forward() function for FFNN

Figure 1 shows how the `forward()` function is implemented. (1) The hidden layer representation is obtained by calling W1 with the input vector. The hidden layer is passed through the ReLU activation function (let this be hidden_layer_rep). (2) The output layer representation is obtained by calling W2 with hidden_layer_rep. (3) The probability distribution is obtained by passing the output layer through softmax.

The optimizer used is Stochastic Gradient Descent with the FFNN model parameters, a learning rate of 0.01, and a momentum of 0.9. For each epoch, after the model is trained, the optimizer's gradients are cleared for every parameter x in the optimizer (`optimizer.zero_grad()`). This is again done for each minibatch of size 16 that the data is split into. Each example in the minibatch is iterated through. Before the next minibatch starts, dloss/dx is computed for every parameter x and these are accumulated into their respective gradient (`loss.backward()`) and the value of x is updated using its gradient (`optimizer.step()`).

2.2 RNN

The forward computation part in `forward()` is filled in to implement a 1 hidden layer RNN with tanh activation. The `__init__(self)` function has two parameters: the input dimension and the hidden layer dimension. Layers and parameters are defined in the `__init__(self)` function. `RNN(n,m)` is a PyTorch function that creates a recurrent network with n inputs and m hidden units using numOfLayer hidden layers (in this case 1) and a nonlinearity function (in this case tanh). There is 1 Linear output layer W in which Linear has two parameters: the hidden unit dimension and the output dimension of 5.

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

    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(output_sum)

    return predicted_vector
```

Figure 2: Implementation of `forward()` function for RNN

Figure 2 is how the `forward()` function is implemented. (1) The hidden layer representation is obtained by calling the RNN function with the inputs (let this be hidden_layer_rep). (2) The output layer representation is obtained by calling W with hidden_layer_rep. (3) The sum over the output is obtained (the vectors for each token of the output layer are summed as a representation for the entire sequence). (4) The probability distribution is obtained by passing the summed output layer through softmax.

The optimizer used is Adam with the RNN model parameters and a learning rate of 0.01 which is different than the one used in FFNN. Word embeddings are used for initialization of word representations rather than the bag-of-words vectorization in FFNN. For each review in the data, the input vector is joined by space to get the words, punctuation is removed, and each word's embedding is looked up and used. Instead of running over all epochs like in FFNN, RNN has a stopping condition. By keeping track of the last validation accuracy and last training accuracy (the previous epoch), the model checks if the validation accuracy is less than the last validation accuracy and if the training accuracy is greater than the last training accuracy. If this is true, the model stops. In this way, training is done to avoid overfitting and the last validation accuracy is the best one obtained.

3 Experiments and Results

Evaluations

The models are evaluated by accuracy. To use the model, we pass it the input vector (FFNN) or word embedding (RNN). This executes the model's `forward()` implementation. The gold label is obtained directly from the data and the predicted label is obtained from the maximum value from the model's forward pass. From here, we can check if the predicted label is equal to the gold label. Accuracy is then the number of correct divided by the total.

Results

Hidden Layer Dimension	Total Epochs	Current Epoch	Training Accuracy	Validation Accuracy
5	10	1	0.66475	0.6225
5	10	2	0.68925	0.6
5	10	3	0.6895	0.5775
5	10	4	0.714625	0.53125
5	10	5	0.7325	0.53875

5	10	6	0.751375	0.58875
5	10	7	0.769125	0.615
5	10	8	0.763	0.62125
5	10	9	0.798875	0.6025
5	10	10	0.7825	0.56875

Table 1: Results of FFNN model with hidden layer dimension 5 and 10 epochs

Hidden Layer Dimension	Total Epochs	Current Epoch	Training Accuracy	Validation Accuracy
10	10	1	0.703375	0.59375
10	10	2	0.7225	0.595
10	10	3	0.729375	0.62
10	10	4	0.754625	0.60625
10	10	5	0.7875	0.58625
10	10	6	0.791125	0.6225
10	10	7	0.783375	0.585
10	10	8	0.783	0.535
10	10	9	0.77075	0.56875
10	10	10	0.824375	0.55125

Table 2: Results of FFNN model with hidden layer dimension 10 and 10 epochs

By fixing the values of learning rate = 0.01, momentum = 0.9, and minibatch size = 16, we are able to observe only the effect of hidden layer dimension on training and validation accuracy. The FFNN model is run with hidden layer dimensions 1, 5, 10, and 25 and 1, 5, 10, and 25 epochs. Here, two runs are shown to observe the effect of changing hidden layer dimension on accuracy over the same number of epochs. As hidden layer dimension increases, validation accuracy is highest at a higher number of epochs. Validation accuracy for the hidden layer dimension of 5 is maximum (0.6225) at epoch 1 while validation accuracy for the hidden layer dimension of 10 is maximum (0.6225) at epoch 6. Increasing the hidden layer dimension further to 25 (over 10 epochs) results in the highest validation accuracy at epoch 1 (0.6325), so comparing different models shows different observations. As the current epoch increases, the difference between training and validation accuracy increases since training accuracy continues to increase over the epochs.

Of the two models above, the one with a hidden layer dimension of 10 performs better even though its validation accuracy peaks at a later epoch. This is because the training accuracy at epoch 1 for the model with a hidden dimension layer of 5 (0.665) is lower compared to the training accuracy at epoch 6 for the model with a hidden dimension layer of 10 (0.791). The model with a hidden dimension layer of 10 has a more acceptable training accuracy. The test accuracy for the Table 1 model is 0.329 and for the Table 2 model is 0.282. The low test accuracies may be because the test data has a lot of unseen examples and the models are not generalizing well enough to predict accurately on the test set; in other words, they are mostly overfitting.

Hidden Layer Dimension	Total Epochs	Current Epoch	Training Accuracy	Validation Accuracy
5	10	1	0.45325	0.295
5	10	2	0.476375	0.41375
5	10	3	0.49	0.455
5	10	4	0.493625	0.4475

Table 3: Results of RNN model with hidden layer dimension 5 and 10 epochs

Hidden Layer Dimension	Total Epochs	Current Epoch	Training Accuracy	Validation Accuracy
10	10	1	0.4615	0.50875
10	10	2	0.493875	0.5125
10	10	3	0.476	0.51125
10	10	4	0.50375	0.46375

Table 4: Results of RNN model with hidden layer dimension 10 and 10 epochs

Hidden Layer Dimension	Total Epochs	Current Epoch	Training Accuracy	Validation Accuracy
25	10	1	0.459875	0.51
25	10	2	0.48625	0.52125
25	10	3	0.518375	0.4

Table 5: Results of RNN model with hidden layer dimension 25 and 10 epochs

The RNN model is run with hidden layer dimensions 1, 5, 10, and 25 and 1, 5, 10, and 25 epochs. It is also run with 100 hidden layer dimensions and 25 epochs. Here, three runs are shown to observe the effect of changing hidden layer dimension on accuracy over the same number of epochs. The validation accuracies for the RNN runs are low. When run for 10 epochs, the model stops after 4 epochs for hidden layer dimensions of 5 and 10 and stops after 3 epochs for hidden layer dimension of 25. The best validation accuracy gets higher as the hidden layer dimension increases over the same number of total epochs. From observing the results, we can infer that no matter what the hidden layer dimension is, the model only runs for a few epochs.

The test accuracy for the Table 3 model is 0.13375, for the Table 4 model is 0.00125, and for the Table 5 model is 0.1075. The low test accuracies seem meaningless as a measure of how well the models are performing. Since the training accuracies are low as well (around 0.5 for the models shown above), the models may be undertrained, and the stopping condition may need to be revised to allow more epochs to run for training.

The complete results for all the runs are uploaded to the Github repository.

4 Analysis

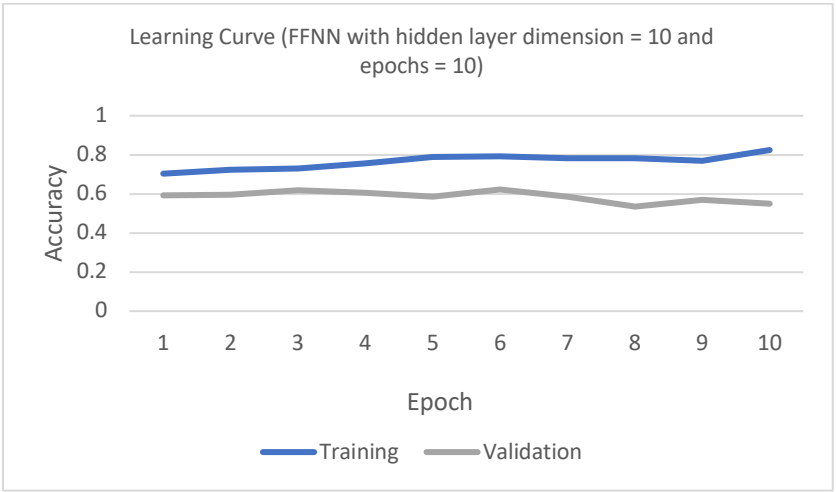

Figure 3: Learning Curve for Best System

The best system for FFNN is the model with a hidden layer dimension of 10 and 10 epochs. Other considerable models have hyperparameters hidden layer dimension of 10 and 5 epochs as well as hidden layer dimension of 25 and 5 epochs; however, these two have training accuracy around 0.6-0.7 which could

be improved with more training. A “good” system has increasing training accuracy over the epochs and validation accuracy that peaks at some epoch before decreasing as shown in Figure 3.

One of the examples of an error could be the reviews that have sarcasm, slang, typos, contradictions, or multiple sentiments. Such inputs can make it difficult for the FFNN to correctly predict a rating based on the input text. On the other hand, RNN considers word embeddings and can reduce error on such inputs. The word embeddings provided currently can be fine-tuned and pre-trained on a larger vocabulary to combat these issues. In conclusion, the models might not be able to capture nuances of modern natural language that change the sentiment of the text vastly.

Other errors that will result in inaccurate predictions are:

- One word having multiple meanings, for example “bank” can refer to a financial institution, a storage area, or the side of a river.
- Overfitting is a very common issue with neural networks, which makes the models inaccurate on new or unseen examples.

Some solutions to improve model performance and avoid these errors are:

- Better data preprocessing to remove noise from the training data whilst still maintaining the context important text.
- Using a larger training dataset if possible or augmenting the existing dataset by including synonyms or paraphrases to prevent overfitting.
- Using fine-tuned pre trained word embeddings.
- Combining different models together and using a weighted average to predict more accurately.

Extra Observations: In addition to the default code, we tried a custom FFNN model with two hidden layers to where on multiple runs with different dimensions, we observed that adding the additional layer did not improve the accuracy for validation or test data sets. It seems that the model is overfitting around 5 epochs on higher dimensions and adding layers will not help unless the training data is expanded to handle more unseen inputs. The model also runs slower on higher dimensions. The reasoning behind having two layers was to prevent data loss while converting the input dimension to a smaller dimension but it can be concluded that training data set even though having a high number of dimensions, does not have enough variety of inputs and therefore adding more dimensions will not lead to a better accuracy.

Additionally, whilst running FFNN for 50 or more epochs it is observed that training accuracy eventually reaches 99%, but the validation accuracy remains in a range of 55-60%, which again solidifies our understanding about the limitations of the given dataset.

5 Conclusion and Others

We worked together on the code and report. Nishad focused more so on the code and Ishwari focused more so on the report. Both members contributed to all parts of the project. The project was not too difficult. It reinforced our understanding of neural networks and the content learned in class. We worked on the project over two weeks, spending about an hour or two working two to three times a week.