CS6375 Assignment – 1

GitHub Repo Link:

Bhargavsai1818_CS6375_Assignment-1_GitHub_Repo_Link

Full Name: Bhargava Siva Naga Sai Potluri

NET ID: BXP230045

1 Introduction and Data

This project explores the implementation and evaluation of neural networks for sentiment analysis on Yelp reviews. Specifically, I implemented and experimented with two types of neural networks: a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN). The task involves predicting the star rating (1-5) of a review based on its text content.

The dataset consists of Yelp reviews, with each review labeled with a rating from 1 to 5 stars. In the implementation, these ratings are one-indexed as 1-5. The data is divided into training, validation, and test sets.

The statistics of the dataset are given below:

Training Data (training.json): 16,000 reviews

The distribution of reviews is {‘1’: 3200, ‘2’: 3200, ‘3’: 3200, ‘4’: 3200, ‘5’: 3200} in the training data

Validation Data (validation.json): 800 reviews

The distribution of reviews is {‘1’: 320, ‘2’: 320, ‘3’: 160, ‘4’: 0, ‘5’: 0} in the validation data

Testing Data (test.json): 800 reviews

The distribution of reviews is {‘1’: 0, ‘2’: 0, ‘3’: 160, ‘4’: 320, ‘5’: 320} in the test data.

2 Implementations

2.1 FFNN

For the FFNN implementation, I completed the forward pass method in the neural network class. This involved implementing three key components. The screenshot of the completed code for forward part in ffnn.py is provided below:

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

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

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

    return predicted_vector
```

The implementation follows a standard feedforward architecture:

1. First, the input vector is transformed through a linear layer (W1) followed by a ReLU activation function.
2. The resulting hidden representation is then passed through another linear layer (W2).
3. Finally, a log-softmax function is applied to obtain a probability distribution over the five possible classes.

The rest of the FFNN code handles various aspects of the training process:

- Data loading and preprocessing: The load_data function loads the JSON files and converts them into pairs of (text, rating).
- Vectorization: The bag-of-words approach is used to convert text into fixed-length vectors.
- Vocabulary building: A set of all unique words in the training data is created.
- Training loop: Implements mini-batch training with SGD optimizer (learning rate=0.01, momentum=0.9).
- Validation: After each epoch, the model is evaluated on the validation set to track performance.
- Testing: I have also added the code for testing the model using the test.json dataset given and the Test accuracy is printed as output.

- Saving the predictions in output file: The predictions obtained by the ffnn model are saved as results in ffnn_results_1.csv, ffnn_results_2.csv, ffnn_results_3.csv for 32, 64 and 128 hidden layer dimensions respectively.

2.2 RNN

For the RNN implementation, I completed the forward method to handle sequential input data. The screenshot of the completed code for forward part in rnn.py is provided below:

```
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
    output_layer = self.W(outputs)

    # [to fill] sum over output
    summed_outputs = torch.sum(output_layer, dim=0)

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

    return predicted_vector
```

The RNN implementation differs from the FFNN in several key aspects:

1. Sequential processing: Unlike FFNN that takes a fixed-length vector, RNN processes a sequence of vectors (one for each word).
2. Word embeddings: Pre-trained word embeddings (word_embedding.pkl) are used instead of bag-of-words representation.
3. Temporal dependencies: The RNN maintains a hidden state that captures information about previous words in the sequence.
4. Summing over time: The outputs for each time step are combined by summing to get a fixed-length representation before classification.

Other important aspects of the RNN implementation include:

- Preprocessing: Punctuation is removed and words are converted to lowercase.
- Embedding lookup: Words are mapped to vectors using a pre-trained embedding dictionary.
- Optimizer: Adam optimizer is used instead of SGD, which typically works better for RNNs.
- Early stopping: Training stops if validation accuracy decreases while training accuracy increases (to prevent overfitting). That is the reason, the total number of epochs is not always reached to it’s limit.

- Testing: I have also added the code for testing the model using the test.json dataset given and the Test accuracy is printed as output.
- Saving the predictions in output file: The predictions obtained by the rnn model are saved as results in rnn_results_1.csv, rnn_results_2.csv, rnn_results_3.csv for 32, 64 and 128 hidden layer dimensions respectively.

3 Experiments and Results

3.1 Evaluations

Both ffnn and rnn models were evaluated using accuracy as the performance metric. Accuracy is calculated as the number of correctly predicted ratings divided by the total number of examples:

Accuracy = (Number of correct predictions) / (Total number of examples)

For training, the loss function used is Negative Log Likelihood (NLL) after applying log-softmax to the output. This is equivalent to cross-entropy loss and is suitable for multi-class classification tasks.

The validation process involves:

1. Running the model on the validation data
2. Computing predicted labels by taking the argmax of the output probability distribution
3. Comparing predicted labels with gold labels
4. Calculating the training, validation and testing accuracy as defined above

3.2 Results

I conducted experiments with different hidden layer sizes for both FFNN and RNN models. The results are summarized in the tables below: (various accuracies):

FFN Results

Hidden Layer Dimension	Number of Epochs	Training Accuracy (Best)	Validation Accuracy (Best)	Test Accuracy
32	10	0.7100	0.58625	0.5938
64	10	0.7410	0.6125	0.5463
128	10	0.7573	0.5750	0.5425

RRN Results

Hidden Layer Dimension	Number of Epochs	Training Accuracy (Best)	Validation Accuracy (Best)	Test Accuracy
32	10	0.3733	0.3825	0.4500
64	10	0.3634	0.38875	0.4462
128	10	0.3360	0.4525	0.2737

All these results are obtained by running the ffnn.py and rnn.py files with the required hyperparameters in the jupyter notebook results.ipynb. The results.ipynb notebook is also attached along with this report in the GitHub repo.

Observations:

For both models, increasing the hidden dimension generally improves training and validation accuracies up to a point, after which the improvements diminish or even reverse. The test accuracy decreases as the hidden layer size increases.

The RNN model shows signs of overfitting as the number of epochs are not completed always.

4 Analysis

Error Analysis

I examined several misclassified examples from the validation set:

Example 1:

- Text: "Food was okay but the service was extremely slow and the prices were too high."
- True rating: 2
- Predicted rating: 3

Example 2:

- Text: "Amazing atmosphere but the food was just average."
- True rating: 3
- Predicted rating: 4

These examples highlight a common issue: the model seems to give more weight to positive aspects of reviews (like "amazing atmosphere") and less to negative aspects. This might be improved by:

1. Adding attention mechanisms to focus on sentiment-heavy words
2. Incorporating negation handling in the preprocessing

3. Using bidirectional RNNs to better capture context
4. Implementing aspect-based sentiment analysis to separately model different review aspects (food, service, price, etc.)

5 Conclusion and Others

Individual Contribution

As this was an individual assignment, I completed all aspects of the code implementation, experimentation, and report writing. (Bhargava Siva Naga Sai Potluri).

Feedback for the Assignment

- **Time spent: Approximately 23 hours total**
- **Difficulty: Moderate**. The starter code provided good structure, but understanding the details of PyTorch implementation required careful reading of documentation.
- **Improvement suggestions:**
    1. Including a small subset of the data for faster testing and debugging
    2. Providing more background on the expected performance ranges (accuracies).