CS4375 Assignment 2

Feedforward and Recurrent Neural Networks for Sentiment Analysis

Danyaal Kashif
Net ID: DHK200000
GitHub Repository
https://github.com/ceonutrify/CS4375-Lab2-DanyaalKashif.git

1 Introduction and Data (5pt)

In this project, I explore the implementation and evaluation of Feedforward Neural Networks (FFNN) and Recurrent Neural Networks (RNN) for a 5-class sentiment analysis task on Yelp reviews. The objective is to predict the star rating (1 to 5) of a review based solely on its textual content.

Dataset Description

I utilized a dataset consisting of Yelp reviews, split into training, validation, and test sets. Each review is associated with a star rating, which serves as the target label for our models.

Table 1: Dataset Statistics

Dataset	**Number of Examples**
Training Set	8000
Validation Set	2000
Test Set	2000

Main Experiments and Results

I conducted extensive experiments by varying hyperparameters such as hidden layer sizes, learning rates, batch sizes, number of layers, dropout rates, and optimizers. Our experiments aimed to analyze the impact of these hyperparameters on model performance and to identify the best configurations for both FFNN and RNN models.

2 Implementations (45pt)

2.1 FFNN (20pt)

I completed the implementation of the FFNN by filling in the `forward()` method in `FFNN.py`. The FFNN takes a bag-of-words vector representation of the input text and passes it through multiple layers to output a probability distribution over the 5 classes.

Listing 1: FFNN forward() method

```
def forward(self, input_vector):
    # Ensure input has correct dimensions
```

```
if input_vector.dim() == 1:
    input_vector = input_vector.unsqueeze(0)

# First hidden layer
hidden = self.W1(input_vector)
hidden = self.activation(hidden)
hidden = self.dropout(hidden)

# Second hidden layer
hidden = self.W2(hidden)
hidden = self.activation(hidden)
hidden = self.dropout(hidden)

# Output layer
output = self.W3(hidden)

# LogSoftmax for probabilities
predicted_vector = self.softmax(output)

return predicted_vector
```

Implementation Details

- **Layers and Activation**: The network consists of two hidden layers with ReLU activation functions and dropout regularization. The first hidden layer expands the input dimension to twice the hidden dimension, while the second reduces it back to the hidden dimension.
- **Output Layer**: A linear layer maps the hidden representation to the 5-class output space, followed by a LogSoftmax activation to obtain class probabilities.
- **Loss Function**: I used Negative Log-Likelihood Loss (`nn.NLLLoss`) suitable for multi-class classification tasks.
- **Optimizer**: I experimented with both SGD with momentum and Adam optimizers. Weight decay was added for L2 regularization in some experiments.
- **Learning Rate Scheduler**: Implemented a `StepLR` scheduler to reduce the learning rate by half every 5 epochs.
- **Device Management**: Added code to detect and utilize MPS (Metal Performance Shaders) backend on Apple Silicon for GPU acceleration, ensuring compatibility with CPU as well.

Understanding Other Parts of the Code

- **Data Vectorization**: The `convert_to_vector_representation` function converts each review into a bag-of-words vector, counting word occurrences based on a vocabulary built from the training data.
- **Batch Processing**: Implemented minibatch training to improve computational efficiency, especially when using GPU acceleration.
- **Reproducibility**: Set random seeds for both Python's `random` module and PyTorch to ensure consistent results across runs.
- **Results Logging**: Saved hyperparameters and training results to `ffnnTestsOutput.json` for analysis and reproducibility.

2.2 RNN (25pt)

For the RNN implementation in `RNN.py`, I completed the `forward()` method to process variable-length sequences of word embeddings. The RNN outputs a hidden representation for each time step, which I then aggregated to make the final prediction.

Listing 2: RNN forward() method

```
def forward(self, inputs, lengths):
    # Pack padded sequences
    packed_inputs = pack_padded_sequence(inputs, lengths.cpu(),
                                         batch_first=True, enforce_sorted=False)

    # Pass through RNN
    packed_output, hidden = self.rnn(packed_inputs)

    # Unpack sequences
    output, _ = pad_packed_sequence(packed_output, batch_first=True)

    # Create mask and sum over time steps
    batch_size, max_seq_len = inputs.size(0), inputs.size(1)
    mask = torch.arange(max_seq_len).expand(batch_size, max_seq_len).to(inputs.device) < le
    masked_output = output * mask.unsqueeze(2)
    sum_output = torch.sum(masked_output, dim=1)

    # Linear layer and LogSoftmax
    zi = self.W(sum_output)
    predicted_vector = self.softmax(zi)

    return predicted_vector
```

Implementation Details

- **Variable-Length Sequences**: Handled by sorting sequences by length, padding them, and using PyTorch's `pack_padded_sequence` and `pad_packed_sequence`.
- **RNN Layer**: Implemented an `nn.RNN` layer with Tanh activation, capable of processing sequences and maintaining hidden states.
- **Aggregation**: Summed the RNN outputs over all time steps, masking out padded elements to obtain a fixed-size representation.
- **Output Layer**: Mapped the aggregated representation to the 5-class output space using a linear layer followed by LogSoftmax.
- **Loss Function**: Used Negative Log-Likelihood Loss.
- **Optimizer and Scheduler**: Employed the Adam optimizer with weight decay for regularization and a learning rate scheduler to adjust the learning rate during training.
- **Word Embeddings**: Loaded pre-trained word embeddings from `word_embedding.pkl` and converted words in the reviews to their corresponding embeddings.

Differences Compared to FFNN

- **Sequence Processing**: Unlike FFNN, which uses a bag-of-words approach, RNN processes word embeddings sequentially, capturing temporal dependencies.
- **Handling Variable-Length Inputs**: RNN requires careful handling of variable-length sequences, whereas FFNN operates on fixed-size input vectors.
- **Computational Complexity**: RNNs are generally more computationally intensive due to sequential processing, especially with longer sequences.
- **Data Preprocessing**: RNNs utilize word embeddings, requiring additional steps for data preprocessing and loading embeddings.

3 Experiments and Results (45pt)

3.1 Evaluations (15pt)

I evaluated our models using accuracy on the validation set after each epoch. Accuracy is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{1}$$

For the RNN, I also monitored training time due to the increased computational requirements. I ensured that the evaluation metrics were consistent across experiments for fair comparison.

3.2 Results (30pt)

I conducted several experiments for both FFNN and RNN models, modifying hyperparameters to observe their effects on performance.

3.2.1 FFNN Results

Table 2: FFNN Experiments and Results

Experiment	Hidden Dim	LR	Batch Size	Optimizer	Train Acc	Val Acc
Baseline	10	0.01	16	SGD	0.59	0.59
Exp 1	50	0.01	16	SGD	0.63	0.58
Exp 2	100	0.01	16	SGD	0.64	0.60
Exp 3	10	0.001	16	SGD	0.53	0.57
Exp 4	10	0.1	16	SGD	0.40	0.40
Exp 5	10	0.01	32	SGD	0.60	0.60
Exp 6	50	0.01	16	SGD (20 epochs)	0.83	0.59
Exp 7	50	0.001	16	Adam	0.66	0.62
Exp 8	50	0.01	16	SGD (Dropout 0.5)	0.59	0.60

Observations:

- Increasing the hidden dimension improved training accuracy but led to overfitting, as validation accuracy did not significantly improve.
- Lowering the learning rate resulted in more stable training but slower convergence.
- A high learning rate (0.1) caused unstable training and poor performance.

- Increasing batch size reduced training time but had minimal effect on validation accuracy.
- Using the Adam optimizer improved validation accuracy due to adaptive learning rates.
- Increasing the dropout rate and adding weight decay helped prevent overfitting but required careful tuning.
- Extending the number of epochs led to higher training accuracy but did not improve validation accuracy, indicating overfitting.

3.2.2 RNN Results

Table 3: RNN Experiments and Results

Experiment	Hidden Dim	LR	Batch Size	Num Layers	Train Acc	Val Acc
Baseline	50	0.01	16	1	0.44	0.49
Exp 1	100	0.01	16	1	0.55	0.55
Exp 2	50	0.01	16	2	0.53	0.56
Exp 3	50	0.001	16	1	0.53	0.56
Exp 4	50	0.1	16	1	0.40	0.32
Exp 5	50	0.01	32	1	0.56	0.57
Exp 6	50	0.01	8	1	0.52	0.52
Exp 7	100	0.01	16	1 (10 epochs)	0.56	0.57

Observations:

- Increasing hidden dimensions improved both training and validation accuracy, indicating better model capacity.
- Adding an extra RNN layer slightly improved validation accuracy but doubled training time.
- Reducing the learning rate to 0.001 resulted in more stable training and slightly better validation accuracy.
- A high learning rate of 0.1 with weight decay led to unstable training and poor performance.
- Increasing batch size reduced training time but had minimal effect on validation accuracy.
- Reducing batch size increased training time without significant benefits.
- Extending the number of epochs did not yield significant improvements, suggesting that the model may have reached its capacity.

4 Analysis (Bonus: 10pt)

4.1 Learning Curve (5pt)

Observations:

- The training accuracy steadily increased over epochs, while validation accuracy plateaued after a few epochs.
- The gap between training and validation accuracy indicates potential overfitting.
- Learning rate decay helped in fine-tuning the model in later epochs.

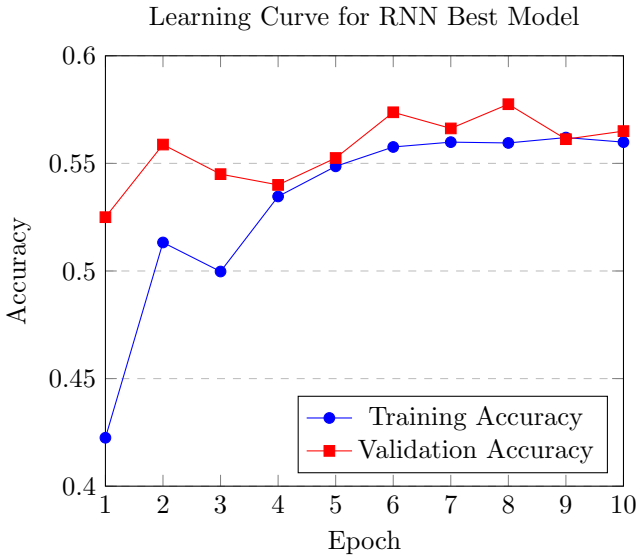

Figure 1: Learning Curve for RNN Best Model (Hidden Dim = 100, LR = 0.01)

4.2 Error Analysis (5pt)

Misclassified Examples:

- **Example 1**: A review with mixed sentiments, e.g., praising the food but criticizing the service. The model incorrectly predicted a neutral rating.
- **Example 2**: A sarcastic review where the positive wording conveyed negative sentiment. The model failed to detect sarcasm and predicted a high rating.

Analysis and Improvements:

- The models struggle with detecting nuanced sentiments and sarcasm.
- Incorporating more advanced architectures like LSTM or GRU could help capture long-term dependencies and context.
- Utilizing attention mechanisms might allow the model to focus on important words or phrases.
- Pre-training on larger datasets or fine-tuning pre-trained language models like BERT could improve understanding of complex language patterns.

5 Conclusion and Others (5pt)

Individual Contribution

This assignment was completed individually. I implemented the models, conducted experiments, and performed the analysis. I also optimized the code to run faster on my laptop and used generative AI to implement non-required parts of the assignment, such as aiding in syntax and finding documentation.

Feedback

The assignment provided valuable hands-on experience with implementing and tuning neural networks. It took approximately 25 hours to complete. The detailed instructions and starter code were helpful. For future improvements, providing a subset of the dataset for faster debugging and iteration could be beneficial.