# CS6375 Assignment 1

https://github.com/micahwarner/CS6375_Assignment1

Micah Warner
maw210009

## 1 Introduction and Data (5pt)

**Project Overview:**
This project involves implementing and training two types of neural networks—a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN)—to perform 5-class sentiment analysis on Yelp reviews. The task is to predict the star rating of a review (ranging from 1 to 5 stars) based solely on the review text. The main goal is to evaluate the effectiveness of these neural network architectures in classifying sentiment from textual data.

**Main Experiments and Results:**
Two models were trained:
- **FFNN:** The input was represented using a bag-of-words model. After several experiments, the hidden layer size and the number of epochs were tuned for the best performance.
- **RNN:** The model utilized pre-trained word embeddings, processed each review as a sequence, and aimed to capture the temporal structure of the text.

The training, validation, and test sets were used to evaluate model performance. Key metrics such as accuracy were measured across both models. Early stopping was employed to prevent overfitting for the RNN, and the results showed that the FNN was generally quite a bit more accurate than the RNN, with a dimension size of 128 performing the best.

**Data Overview:**
The dataset consists of Yelp reviews and their associated star ratings. The data was split into training, validation, and test sets, as summarized in the table below:

| **Dataset** | **Number of Examples** |
|---|---|
| **Training Set:** Used to train the models and tune parameters. | 8,000 |
| **Validation Set:** Used to monitor the model's performance and adjust hyperparameters. | 800 |
| **Test Set:** Used to evaluate the model's final performance. | 800 |

## 2 Implementations (45pt)

### 2.1 FFNN (20pt)

```
# [to fill] obtain first hidden layer representation
hidden_layer = self.activation(self.W1(input_vector))  # Apply linear transformation and ReLU

# [to fill] obtain output layer representation
output_layer = self.W2(hidden_layer)  # Apply linear transformation

# [to fill] obtain probability dist.
predicted_vector = self.softmax(output_layer)  # Apply LogSoftmax to get log probabilities
```

In the forward() method of the Feedforward Neural Network (FFNN), I filled in the above key components. The first transformation applies a linear operation (W1) to the input vector, followed by a ReLU activation function to introduce non-linearity, allowing the network to learn complex patterns. The hidden layer representation is then passed through another linear transformation (W2) to map the hidden layer to the output layer. Finally, the logits are passed through a LogSoftmax function to convert them into a log probability distribution.

**Code Explanation:**

**Model Initialization (__init__):**

- Initilializes two linear layers: W1 (input to hidden layer) and W2 (hidden to output layer).
- A ReLU activation function (self.activation) is applied after the first linear transformation to introduce non-linearity.
- The output from the hidden layer is passed through W2, followed by LogSoftmax to convert the final logits into log-probabilities.

**Forward Pass (forward()):**

- The forward pass computes the hidden layer representation using the input vector.
- The output of the hidden layer is passed through the second linear transformation to produce the logits.
- These logits are then passed through LogSoftmax() to get the log-probability distribution across the 5 classes (sentiment ratings).

**Part Explanations:**

- self.W1(input_vector): Transforms the input vector into a hidden layer representation.
- self.activation(): Applies the ReLU activation to introduce non-linearity.
- self.W2(hidden_layer): Transforms the hidden representation into logits for the output layer.
- self.softmax(output_layer): Converts the logits into log probabilities across the classes.

**Key Components:**

**Optimizer:**

- The code uses the Stochastic Gradient Descent (SGD) optimizer with momentum (optim.SGD) to update the model parameters.
- This optimizer is initialized with a learning rate (lr=0.01) and a momentum value (momentum=0.9). Momentum helps the optimizer converge faster by using the previous gradients to smooth the updates.

**Loss Function:**

- The model uses Negative Log Likelihood Loss (NLLLoss), which is appropriate for multi-class classification tasks when using log probabilities (as produced by LogSoftmax).

**Libraries/Tools Used:**

- PyTorch: The primary deep learning framework used for building and training the neural network models.
- tqdm: Used for displaying progress bars during training to track the completion of each epoch.

**2.2 RNN (25pt)**

```
# [to fill] obtain hidden layer representation (https://pytorch.org/docs/stable/generated/torch.nn.RNN.html)
rnn_out, hidden = self.rnn(inputs) #Pass the input through the RNN

# [to fill] obtain output layer representations
output = self.W(rnn_out) #Pass the RNN output through a fully connected layer (W)

# [to fill] sum over output
output_sum = torch.sum(output, dim=0) #Sum over the time steps

# [to fill] obtain probability dist.
predicted_vector = self.softmax(output_sum) #Apply LogSoftmax to get log probabilities
```

In RNN.py, I filled in the above parts of the forward pass for the Recurrent Neural Network (RNN). The input is passed sequentially through the RNN layer. The RNN processes the input and outputs a hidden state for each time step, where rnn_out contains the output for each time step in the sequence, and hidden represents the final hidden state after processing the entire input sequence. The RNN's output for each time step (rnn_out) is then passed through a fully connected layer (W), which projects the RNN output into the output space. Since the RNN outputs a sequence of vectors, I summed these vectors across the time steps to create a single vector representing the entire input sequence. Finally, the summed logits are passed through a LogSoftmax layer to convert them into log probabilities over the 5 sentiment classes.

**FFNN vs. RNN**

**Sequential Processing:**

- RNN: Processes input sequentially, capturing the temporal dependencies between words in a review. This allows the model to maintain an understanding of the order of words, making it better suited for handling sequential data.
- FFNN: Processes the input as a fixed-length vector (bag-of-words), where word order is not considered. It simply uses the counts of words.

**Hidden State:**

- RNN: Maintains and updates a hidden state as it processes each word in the sequence. This hidden state summarizes the information seen so far in the sequence.
- FFNN: Does not have a concept of hidden states over time. It directly transforms the input vector into a hidden layer, which is then passed to the output layer.

**Summation Over Time:**

- RNN: The model outputs a vector for each time step (each word). These vectors are summed across the time steps to obtain a single representation of the sequence.
- FFNN: Works directly on a single vector (bag-of-words representation) and does not involve summation over time steps.

**Word Embeddings:**

- In the RNN, pre-trained word embeddings are used to represent each word as a dense vector. These embeddings are looked up from a pre-trained embedding matrix.
- FFNN: Uses a bag-of-words representation, where each word is treated as a separate feature, and no pre-trained embeddings are used.
- Additionally, the RNN uses the library pickle which is used to load the pre-trained word embeddings from a .pkl file.

## 3 Experiments and Results (45pt)

**Evaluation Metric:**

- **Accuracy:** This is the primary metric used to evaluate the performance of both the Feedforward Neural Network (FFNN) and the Recurrent Neural Network (RNN) models. Accuracy is defined as the proportion of correctly predicted labels out of the total number of examples and is tracked after each epoch to monitor the performance.

**Accuracy Calculation:**

For each example, the model outputs a predicted label (based on the highest log probability). Accuracy is then computed as *accuracy = correct/total.*

Where correct_predictions is the number of reviews where the predicted label matches the actual label and total_examples is the total number of reviews in the dataset (training, validation, or test).

**Hyperparameter Variations:**

For both models, I experimented with different configurations of the hidden unit size to observe how it impacts the model's performance (each with 10 epochs).

**Results Summary:**
The following table summarizes the performance of the models under different configurations, representing the validation accuracy after the final epoch.

| Model Type | Hidden Layer Dimension | Validation Accuracy | Test Accuracy |
|---|---|---|---|
| FFNN | 32 | 0.602 | 0.106 |
| FFNN | 64 | 0.618 | 0.087 |
| **FFNN** | **128** | **0.626** | 0.075 |
| **FFNN** | **256** | 0.580 | **0.120** |
| RNN | 32 | 0.463 | 0.106 |
| **RNN** | **64** | **0.555** | 0.002 |
| **RNN** | **128** | 0.456 | **0.123** |
| RNN | 256 | 0.460 | 0.070 |

**Observations and Analysis:**
**FFNN Performance:**

- The FFNN shows the best validation accuracy at a hidden layer dimension of 128 (62.6%). The most accurate model for test accuracy was for a hidden layer dimension of 256 (12.0%).
- Reducing the hidden size to 32 and 64 also shows a reduction in performance when it comes to validation accuracy, but a slight increase in test accuracy.
- Conclusion: For the FFNN, the optimal hidden dimension in this task seems to be 128, where the model achieves the highest validation accuracy.

**RNN Performance:**

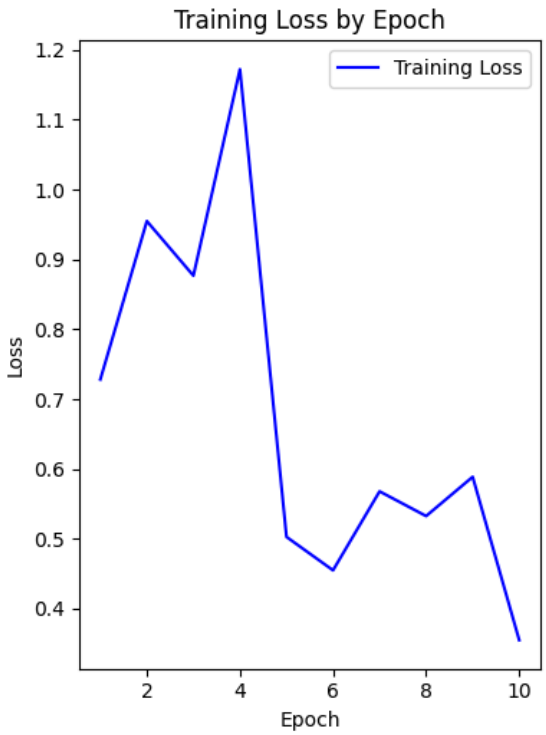
- The RNN's best performance is achieved with a hidden dimension of 64, with a validation accuracy of 55.5%, and a dimension of 128 resulting in the best test accuracy for all models (12.3%).
- Interestingly, both smaller (32) and larger (256) dimensions lead to significantly lower validation accuracy, with a noticeable performance dip at 256 (46.0%).
- Conclusion: The RNN appears to perform best with a more moderate hidden size of 64 or 128, and increasing or decreasing the hidden size too much negatively impacts performance.

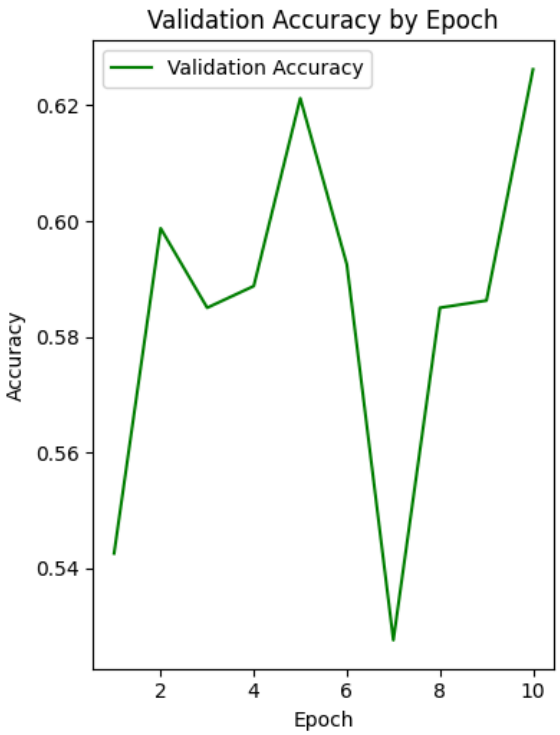
**Comparison Between Models:**

- The FFNN consistently outperforms the RNN across all hidden dimension sizes for validation accuracy, with the FFNN (128) achieving the best overall performance. For test accuracy, however, the RNN with a dimension of 128 achieves the best performance.
- This suggests that for this particular sentiment analysis task, the FFNN model is more suited to the dataset, possibly due to the nature of the data.

## 4 Analysis (bonus: 10pt)

**Plots for FNN with hidden_dim = 128**




| Input: | Predicted Rating: | Actual Rating: |
|---|---|---|
| Burger was so so, milkshake was a 10, the place is really nice, service was not the best, I will Def give it another shot and try a different burger. | 1 | 2 |
| Overly priced :/ they do not clean under nails. Very nice workers though. Great customer service. ;) | 1 | 2 |
| Ordered the lava cheese tart. They made it fresh so about 10 to 15 minutes wait. Too sweet and watery inside. Didn't like it. | 1 | 2 |

All of these inputs are somewhat similar, where they mention a few negatives and then one or two positives about the place, so it seems the model may focus too much on the negative aspects of reviews while ignoring some of the positive aspects.

## 5 Conclusion and Others (5pt)

I worked on this project individually and thus worked on completing the forward() functions for both the FFNN and RNN models, performed hyperparameter tuning, conducted experiments to evaluate model performance, implemented the learning curve plot, and carried out error analysis.

**Feedback:**

I spent around 5 hours on the project. Most of the time was spent on understanding the PyTorch framework, debugging code, and this write-up. The assignment was somewhat challenging but mostly just time-consuming, as I had to wait a lot to repeatedly train the different models. Overall, the assignment was useful in gaining practical experience with PyTorch and neural networks.