# CS6375 Assignment 1

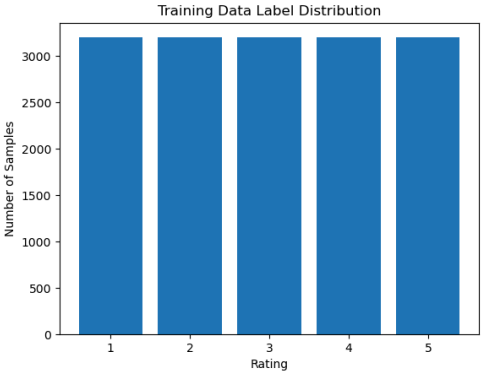
https://github.com/HenryVu27/CS6375/tree/main/A1

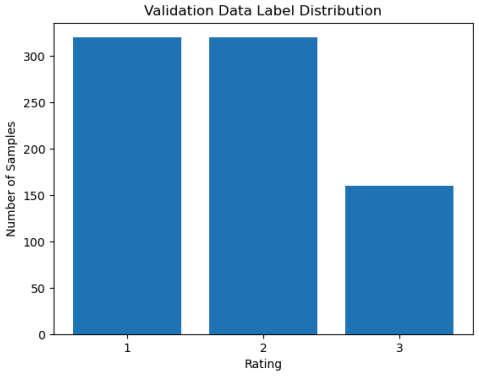
Henry Vu
ddv240000

## 1 Introduction and Data (5pt)

- This project involves building and evaluating two neural network models for sentiment analysis: a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN). The task is to predict Yelp review ratings (1 to 5 stars) based on review text. The dataset consists of pre-split training, validation, and test sets, and word embeddings are provided for initializing word representations in the RNN model.
- The task uses Yelp reviews each with rating $y \in \{1,2,3,4,5\}$. Below is a summary of the data statistics:

| | Number of examples |
|---|---|
| Training set | 16000 |
| Validation set | 800 |




## 2 Implementations (45pt)

### 2.1 FFNN (20pt)

```
def forward(self, input_vector):
    # [to fill] obtain first hidden layer representation
    hidden = self.activation(self.W1(input_vector))
    # hidden = self.dropout(hidden)
    # [to fill] obtain output layer representation
    hidden = self.activation(self.W2(hidden))
    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(hidden)
    return predicted_vector
```

**Model Architecture: PyTorch used**

- The implemented FFNN has 1 input layer, 1 hidden layer, and 1 output layer. A linear transformation (nn.Linear) is applied to the input data, then the output is passed through an activation function (ReLU in this assignment). Then the results are fed to another linear layer and then a softmax layer to compute the final class probabilities.

**Code Understanding:**

- The code first parses the command-line arguments using ArgumentParser() to get the hyperparameters. The random seed is fixed to ensure consistency across different machines.
- Training and validation data are loaded using load_data() to a suitable format. A vocabulary is built from the training data using make_vocab(train_data), which stores all unique words present in the dataset. From this, we can create a mapping from words to indices using make_indices(vocab) and vice versa. Data are converted to vector representations using convert_to_vector_representation(), which converts text data into numerical vectors based on the vocabulary indices.
- The optimizer is stochastic gradient descent (SGD) with momentum: optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9).
- The training loops for a number of epochs (args.epochs). Before each epoch, the training data is shuffled (random.shuffle(train_data)) to help with generalization. The data is processed example by example. After each training epoch, the model is evaluated on the validation set.

## 2.2 RNN (25pt)

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

**Model Architecture:**

- The implemented RNN takes in the input which are word embeddings, then pass it through a tanh gate to calculate the hidden representation. torch.nn.RNN does all the sequential computations of combining the old hidden state of the previous word with the next word in a sequence. For nn.RNN with 1 recurrent layer, “output” contains the hidden representation of all words in the sequence, while “hidden” contains just the hidden representation of the last word.
  Therefore, sum would be taken across dimension 0 of “output” after linear transformation to get the hidden representations of all tokens in the input sequence.
- Unlike FFNN which uses bag of words (BoW) representation, RNN uses word embeddings, i.e. each word is mapped to a float vector of fixed size.

**Code Understanding:**

- Input: Each review text is split into individual words, then each word gets mapped to a word embedding vector using the provided word embedding dictionary.
- The training and validation then proceeds similarly to ffnn.py

# 3 Experiments and Results (45pt)

- Both the FFNN and RNN are trained using negative log-likelihood, which is well-suited for a multiclass classification problem.
- To evaluate the model, I used classification reports and reported the F1-score of each model

  **FFNN:**
- The model performed very well on the training dataset, meaning it was able to capture and learn the sentence nuances for each rating. However, the high accuracy on the

training dataset also suggests overfitting which could result in poor generalization performance.

| | Precision | Recall | F1-score | Support |
|---|---|---|---|---|
| 0 | 0.72 | 0.66 | 0.69 | 320 |
| 1 | 0.55 | 0.37 | 0.44 | 320 |
| 2 | 0.36 | 0.48 | 0.41 | 160 |
| 3 | 0.00 | 0.00 | 0.00 | 0 |
| 4 | 0.00 | 0.00 | 0.00 | 0 |
| Accuracy | | | 0.49 | 800 |
| Macro Avg | 0.32 | 0.28 | 0.30 | 800 |
| Weighted Avg | 0.57 | 0.49 | 0.53 | 800 |

Table 1: Classification report for FFNN with hidden_dim = 128, epoch = 35 on the validation set.

- As observed in Table 1, the validation dataset only has 3 classes instead of the 5 that the model was trained on, therefore, despite its ability to capture the complexities in the training samples, the model does not generalize well. The weighted average F1-score for hidden_dim = 128 FFNN after 35 training epochs is 0.53. Its accuracy on the validation

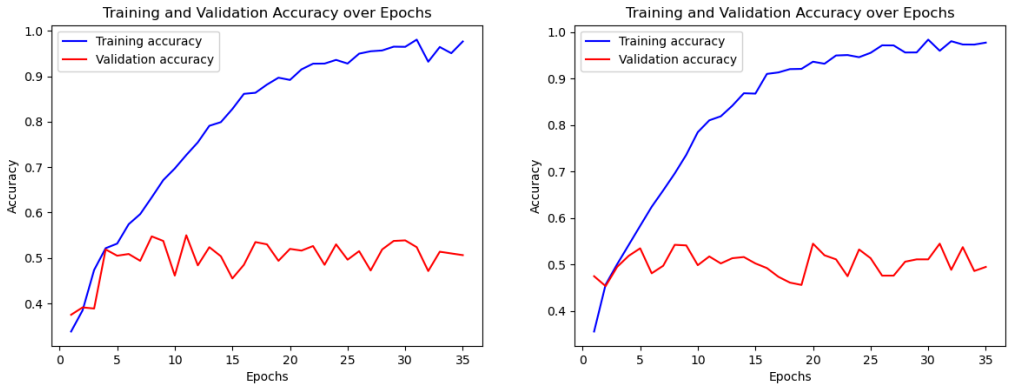



Figure 1: Accuracy and loss over 35 epochs for FFNN with hidden_dim = 128 (left) and hidden_dim = 256 (right).

**RNN:**

- The training accuracy gradually increased in the first 10 epochs until it plateau around 33%. This could mean the model had learned as much as it could from the data given its capacity and may have converged to a local minimum.
- Since we sum up the hidden representation of all time steps, the model treats all timesteps as equal and might not capture temporal dependencies correctly. This might cause the model to not learn correctly. The weighted average F1-score is only 0.35 for the RNN model with hidden_dim = 128 over 35 training epochs. This is surprising as we expected RNN to capture more information and thus be more accurate than FFNN.

| | Precision | Recall | F1-score | Support |
|---|---|---|---|---|
| 0 | 0.64 | 0.36 | 0.46 | 320 |
| 1 | 0.40 | 0.16 | 0.23 | 320 |
| 2 | 0.24 | 0.62 | 0.35 | 160 |
| 3 | 0.00 | 0.00 | 0.00 | 0 |
| 4 | 0.00 | 0.00 | 0.00 | 0 |
| Accuracy | | | 0.34 | 800 |
| Macro Avg | 0.26 | 0.23 | 0.21 | 800 |
| Weighted Avg | 0.47 | 0.34 | 0.35 | 800 |

Table 2: Classification report for RNN with hidden_dim = 128, epoch = 35 on the validation set.

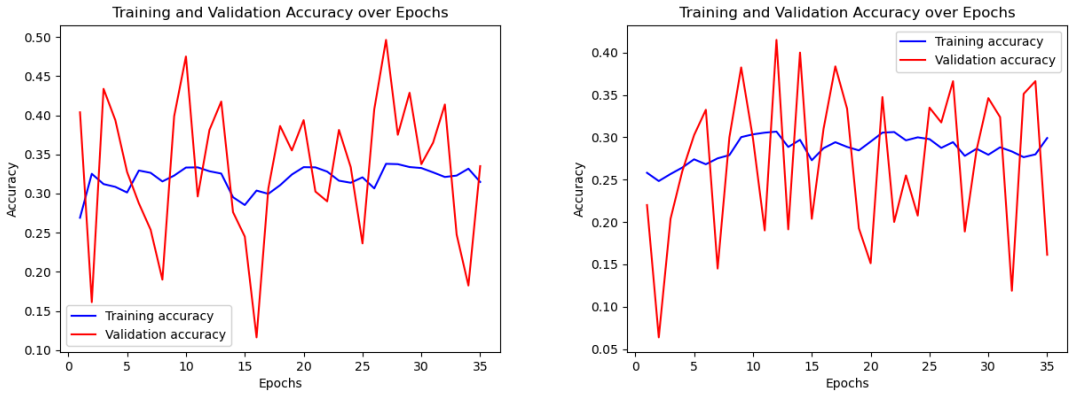



Figure 2: Accuracy and loss over 35 epochs for RNN with hidden_dim = 128 (left) and hidden_dim = 256 (right).

## 4 Analysis (bonus: 10pt)

**Data:**

- There are 16000 samples in the training set and the labels have equal distribution. The total number of unique words is 97304. There is a potential mismatch in class distribution (since we are computing a softmax vector which computes the probability for all classes) and vocabulary between the train and test set, which could lead to overfitting and poor generalization in the test set.

**FFNN:**

- We are using BoW representation, meaning that if the vocab between the training set and test set differ significantly, several issues may arise leading to lower accuracy:
    - Sparsity: There could be many 0s in the word vectors since the vocab size of the training set is very large.

    - Learned features: The model could learn many features different from the test set if trained on a dataset that has little in common with the test set.
    - Out of vocab words in the test set
- **Improvements:** More information about the test set could help, but we could try changing the learning rate, hidden dimension size, dropout, etc. to reduce overfitting to the training set. Or we could use word embeddings instead of BoW

**RNN**

- The current RNN has poor learning capability, so a more sophisticated model with an embedding layer, more RNN layers, bi-direction, etc. could help significantly improve training accuracy.

## 5 Conclusion and Others (5pt)

- **Contribution:** Single-member team, equal contribution
- **Time spent**: For both rnn and ffnn of hidden_dim = 128, each epoch took around ~5 minutes to train on CPU. The starter code could be more cuda friendly (batching data, moving model and data to 'cuda', etc.). For validation model should be in eval() mode and torch.no_grad().
  Since the data samples are processed one at a time, the inner training and validation loop could be further simplified. The dataset could be batched to train more efficiently.