CS6375 Assignment1

FULL_GITHUB_REPOSITORY_URL

IGS Yasaswi - GXI230000

1 Introduction and Data (5pt)

TODO: Briefly describe the project and your main experiments and results, including mentioning the data you use.

This project involved performing sentiment analysis on Yelp reviews using two neural network models: a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN). The task was to predict the sentiment score (1-5) based on review text. The FFNN used a Bag-of-Words approach, while the RNN employed pre-trained word embeddings.

We experimented with different hyperparameters like hidden layer sizes and epochs. The RNN, due to its sequential processing ability, yielded better results than the FFNN. Both models were assessed based on their validation accuracy and training loss, with further error analysis conducted to identify and address common misclassification issues.

TODO: Briefly describe task and data (e.g. how many examples are in the training, develop-ment, and test sets.), it is best to report all the statistics, including counts, in a table.

The project aimed to classify reviews into five sentiment categories. The dataset used consisted of the following splits:

Dataset	**Number of Samples**
Training	16,000
Validation	800
Test	800

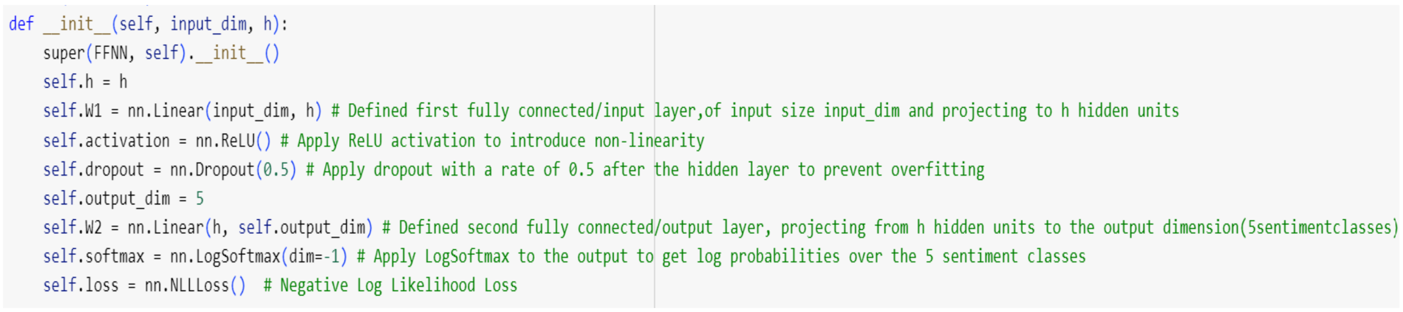
2 Implementations (45pt)

2.1 FFNN (20pt)

TODO: Explain briefly how you implemented filled in the incomplete code for FFNN.py in the form of screenshot (and explanations) in the report. Provide any other libraries/tools that are used; Try to understand what other part of the code is doing, and write your understandings here (e.g. optimizers, initializations, stopping, etc.)

Model Architecture Summary

- **Input Layer**: Takes Bag-of-Words vectors.
- **Hidden Layer**: Fully connected layer with h units, followed by **ReLU** activation.
- **Dropout**: Applies 0.5 dropout for regularization.
- **Output Layer**: Fully connected layer with 5 units (one for each sentiment class), followed by **LogSoftmax** to output log probabilities.



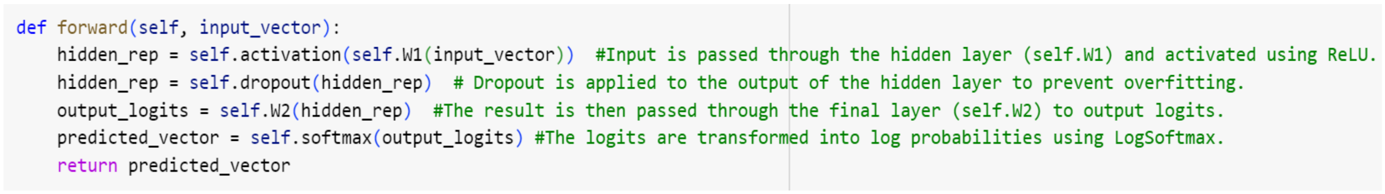
```
def __init__(self, input_dim, h):
    super(FFNN, self).__init__()
    self.h = h
    self.W1 = nn.Linear(input_dim, h) # Defined first fully connected/input layer,of input size input_dim and projecting to h hidden units
    self.activation = nn.ReLU() # Apply ReLU activation to introduce non-linearity
    self.dropout = nn.Dropout(0.5) # Apply dropout with a rate of 0.5 after the hidden layer to prevent overfitting
    self.output_dim = 5
    self.W2 = nn.Linear(h, self.output_dim) # Defined second fully connected/output layer, projecting from h hidden units to the output dimension(5sentimentclasses)
    self.softmax = nn.LogSoftmax(dim=-1) # Apply LogSoftmax to the output to get log probabilities over the 5 sentiment classes
    self.loss = nn.NLLLoss()  # Negative Log Likelihood Loss
```

Forward Function: Defines how input flows through the model:

- Input is passed through the hidden layer, ReLU, and dropout.
- The result is then passed through the output layer, and log probabilities are computed using LogSoftmax.

Loss Function:

- Negative Log Likelihood Loss (NLLLoss) is Used for classification with log probability output. It computes the loss between the predicted log probabilities and the true label for each example.



```
def forward(self, input_vector):
    hidden_rep = self.activation(self.W1(input_vector))  #Input is passed through the hidden layer (self.W1) and activated using ReLU.
    hidden_rep = self.dropout(hidden_rep)  # Dropout is applied to the output of the hidden layer to prevent overfitting.
    output_logits = self.W2(hidden_rep)  #The result is then passed through the final layer (self.W2) to output logits.
    predicted_vector = self.softmax(output_logits) #The logits are transformed into log probabilities using LogSoftmax.
    return predicted_vector
```

Training Process:

- **Random seeds** are fixed using random.seed(42) and torch.manual_seed(42) to ensure **reproducibility** of results.

```
random.seed(42)
torch.manual_seed(42)
```

- The code uses the Adam optimizer(`optim.Adam`), an adaptive learning rate method for gradient-based optimization. The learning rate is set to 0.001, with weight decay (1e-4) for L2 regularization to prevent overfitting.

```
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)
```

- The training process iterates through the dataset in mini-batches (size 32). For each batch, model computes loss using **Negative Log-Likelihood Loss**(NLLLoss), performs **backpropagation**, and updates model's weights using the optimizer.

```
loss.backward()
optimizer.step()
```

- Early Stopping: It is used to halt training if the validation loss does not improved for a specified number of epochs (patience = 3). This helped prevent overfitting by stopping the model before it begins to memorize the training data after 4 epochs.

Data Preparation:

- Bag-of-Words vectorization of the review text. Each review is transformed into a vector of word counts based on the vocabulary built from the training set. Unknown Token (<UNK>) added to handle out-of-vocabulary words.

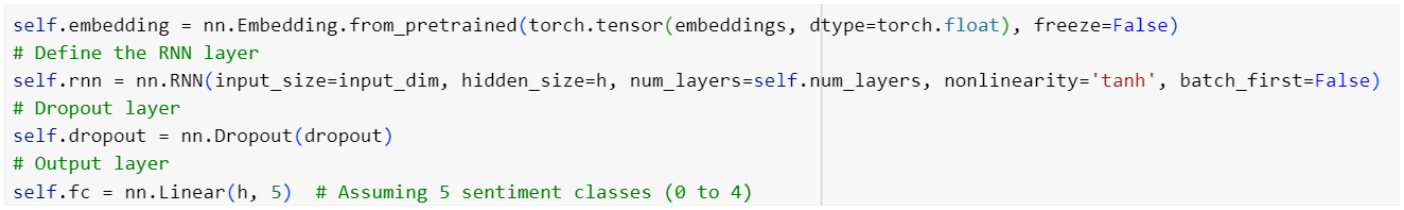
Libraries/Tools: The following tools are used: **PyTorch**, **tqdm**, **Matplotlib**, **JSON**

2.2 RNN (25pt)

TODO: Explain briefly how you implemented filled in the incomplete code for RNN.py in the form of code-snippet screenshot (and explanations) in the report. Try to understand what other part of the code is doing, and write your understandings here (especially parts that is functioning differently as compared to FFNN).

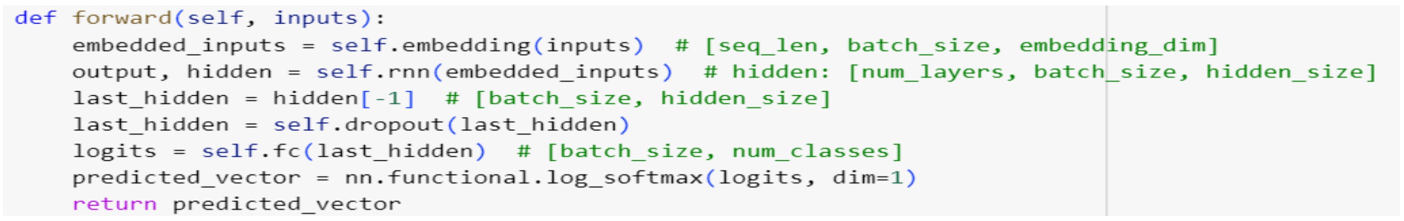
Model Architecture Summary

- **Embedding Layer**: Converts input word indices into dense vectors using pre-trained embeddings.
- **RNN Layer:** Processes sequential data with tanh activation to capture temporal dependencies.
- **Dropout:** Regularizes the hidden layer output to prevent overfitting.
- **Output Layer:** Maps the final hidden state to predict 5 sentiment classes (0 to 4).



```
self.embedding = nn.Embedding.from_pretrained(torch.tensor(embeddings, dtype=torch.float), freeze=False)
# Define the RNN layer
self.rnn = nn.RNN(input_size=input_dim, hidden_size=h, num_layers=self.num_layers, nonlinearity='tanh', batch_first=False)
# Dropout layer
self.dropout = nn.Dropout(dropout)
# Output layer
self.fc = nn.Linear(h, 5)  # Assuming 5 sentiment classes (0 to 4)
```

Forward Function: In the forward pass, the input word indices are first transformed into embeddings. These embeddings are then passed sequentially through RNN, producing a hidden state at each time step, with the hidden state from the last time step being used. This final hidden state is regularized using dropout before passing through a fully connected layer. LogSoftmax is applied to the output to generate log probabilities for the sentiment classes.



```
def forward(self, inputs):
    embedded_inputs = self.embedding(inputs)  # [seq_len, batch_size, embedding_dim]
    output, hidden = self.rnn(embedded_inputs)  # hidden: [num_layers, batch_size, hidden_size]
    last_hidden = hidden[-1]  # [batch_size, hidden_size]
    last_hidden = self.dropout(last_hidden)
    logits = self.fc(last_hidden)  # [batch_size, num_classes]
    predicted_vector = nn.functional.log_softmax(logits, dim=1)
    return predicted_vector
```

Training Process:

- Both FFNN and RNN models follow a similar training loop, iterating over the dataset, computing loss using Negative Log-Likelihood Loss (NLLLoss), and updating the model with the Adam optimizer.
- Early Stopping: Both models utilize early stopping with a patience of 3 epochs to prevent overfitting when validation loss stops improving.

Key Differences:

- **Sequential Processing**:
  - In the **FFNN**, input is treated as a fixed-length vector (Bag-of-Words), losing word order information.
  - In contrast, the **RNN** processes input as a sequence, capturing **temporal dependencies** between words, which is crucial for tasks like sentiment analysis where word order affects meaning.
- **Gradient Clipping**:
  - While **FFNN** doesn't require gradient clipping, **RNNs** are prone to exploding gradients due to their sequential nature. Hence, **gradient clipping** is applied (torch.nn.utils.clip_grad_norm_) in RNN training to stabilize learning.
- **Word Representation**:
  - **FFNN** uses a **Bag-of-Words** representation, treating words as independent entities.
  - **RNN**, on the other hand, uses **pre-trained word embeddings**, which capture richer semantic information in a dense, continuous space.

In summary, **RNNs** are better suited for sequential data tasks, leveraging temporal dependencies and embeddings for more contextual learning, while **FFNNs** are simpler and operate on unordered word representations, making them less capable of capturing context.

- **Libraries/Tools:** The following tools are used: **PyTorch, NumPy, Pickle, tqdm, Matplotlib, ArgParser, JSON**

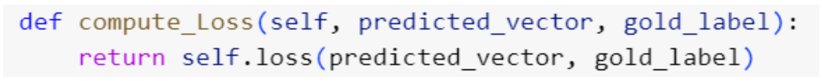
3 Experiments and Results (45pt)

Evaluations (15pt) **TODO:** Explain how you evaluate the models. What metric is used – you can refer to the current implementation.

Both the FFNN and RNN models are evaluated using Accuracy and Loss. `accuracy = correct / total`

Accuracy: Accuracy is the primary evaluation metric, calculated as the percentage of correct predictions out of total examples.

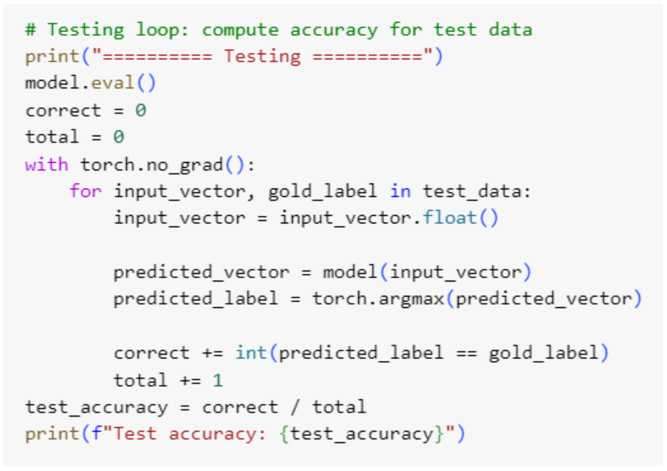
Loss Function: Both models use Negative Log-Likelihood Loss (NLLLoss) to measure the difference between predicted probabilities and true labels.



```
def compute_Loss(self, predicted_vector, gold_label):
    return self.loss(predicted_vector, gold_label)
```

Validation: After each epoch, models are evaluated on the validation set, tracking validation accuracy and loss for generalization and performing early stopping when the model has reached it saturation in terms of learning.

Test Set: After training, final accuracy is calculated on the test dataset by predicting the test accuracy.

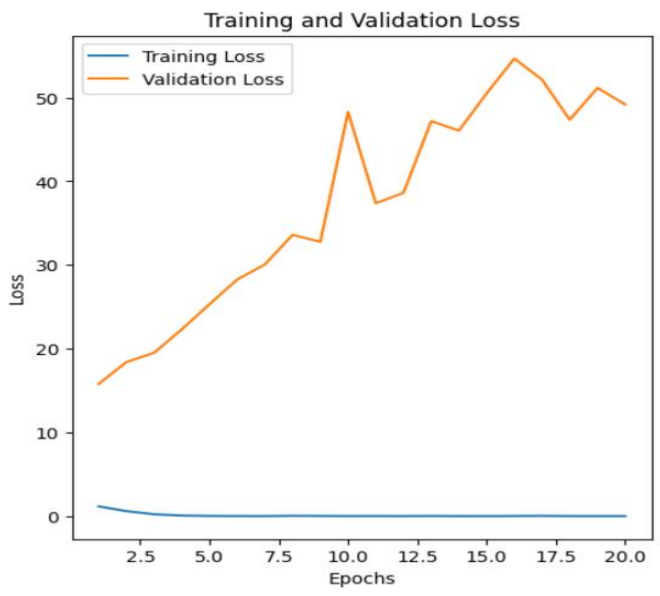


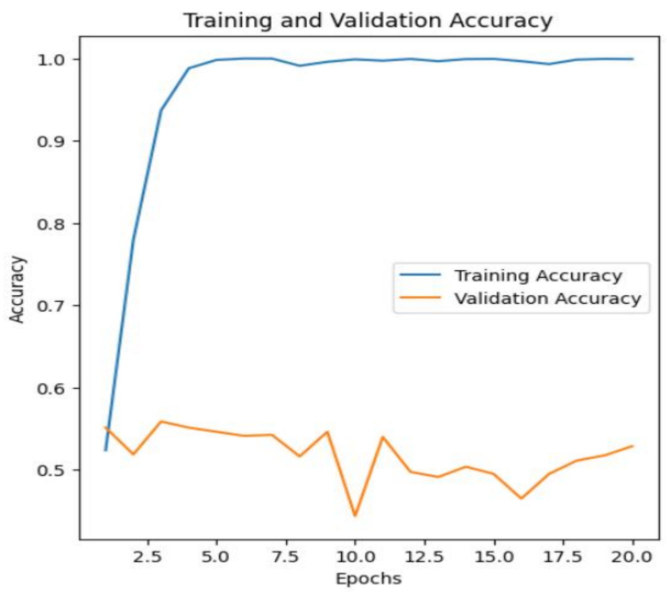
```
# Testing loop: compute accuracy for test data
print("========== Testing ==========")
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for input_vector, gold_label in test_data:
        input_vector = input_vector.float()

        predicted_vector = model(input_vector)
        predicted_label = torch.argmax(predicted_vector)

        correct += int(predicted_label == gold_label)
        total += 1
test_accuracy = correct / total
print(f"Test accuracy: {test_accuracy}")
```

Results (30pt) **TODO:** Apart from the default hyperparameters, try multiple variations (between 1-2 for FFNN and RNN each) of models by changing hidden unit sizes. **TODO:** Summarize the performance of your system and Put the results into tables or diagramsand include your observations and analysis.

FFNN Hyperparameter Variations:

Model 1: hyperparameters: Adam optimizer, 128 hidden dimensions, 20 epochs, learning rate of 0.001, batch size of 32, drop out of 0.3 without early stopping.



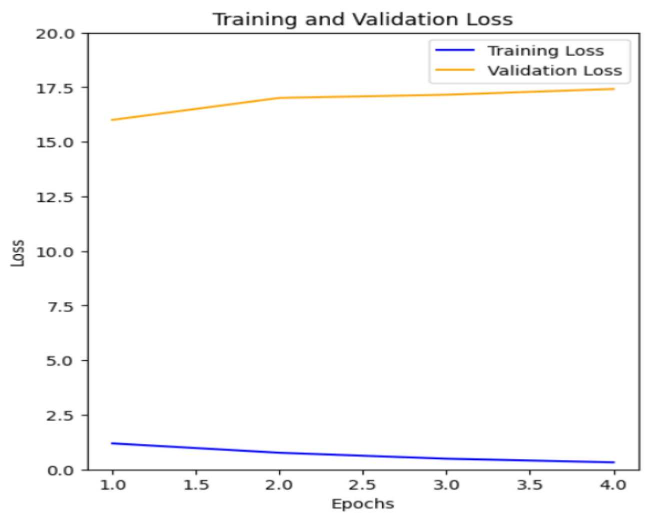


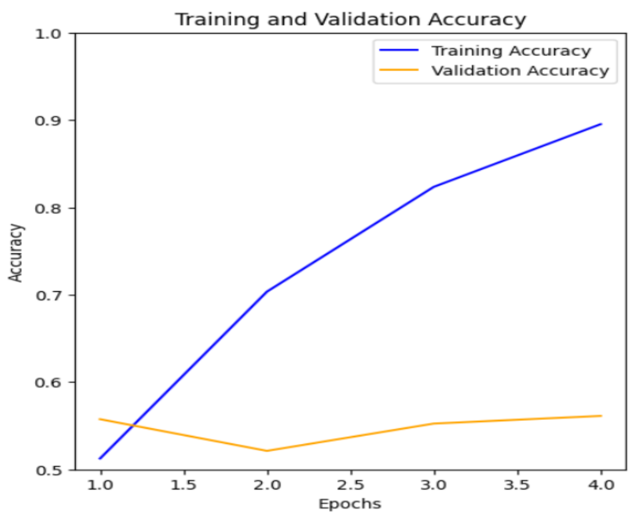
Test Accuracy: 0.545

Observations:

- Training loss decreased steadily over 20 epochs, but validation accuracy fluctuated significantly, indicating overfitting.
- No early stopping led to an overfitted model.

Model 2: hyperparameters: optim.Adam, 256 hidden dimensions, 20 epochs , learning rate of 0.001, weight_decay=1e-4, patience = 3, batch size of 32, also using early stopping patience of 3.





Test Accuracy: 0.5325

Observations:

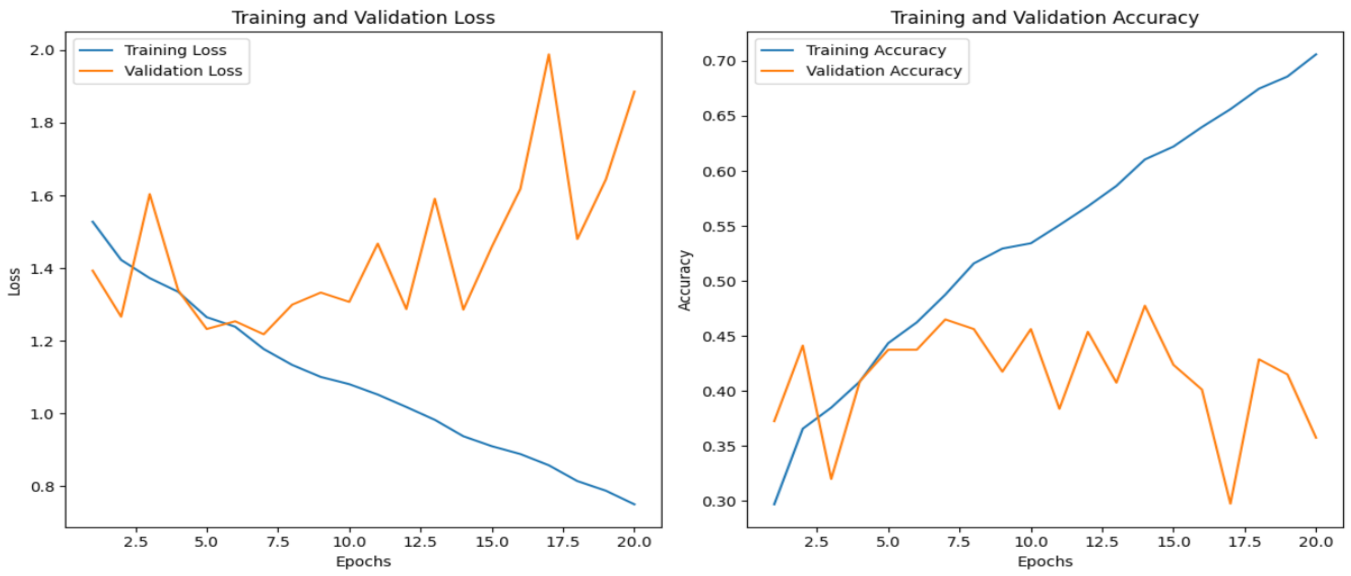
- Early stopping was triggered at epoch 4, leading to a more stable model with better generalization.
- Despite a lower number of training epochs, the test accuracy (0.5325) was only slightly lower, but this approach helped prevent overfitting.

Summary of FFNN:

Increasing hidden units (128 to 256) and using early stopping reduced overfitting and stabilized validation accuracy. Model 1 overfitted due to no early stopping, while Model 2 achieved similar results in fewer epochs with early stopping.

RNN Hyperparameter Variations:

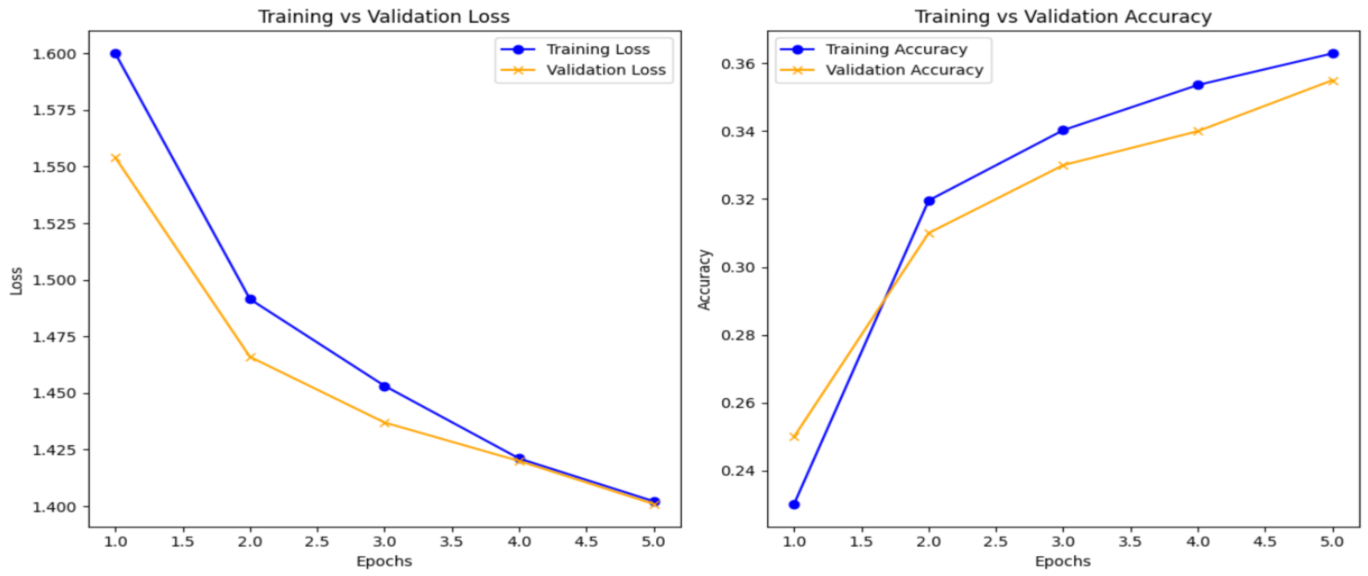
Model 1: hyperparameters: Adam optimizer, 128 hidden dimensions, 20 epochs, learning rate of 0.0005, batch size of 32, drop out of 0.3, Gradient Clipping: 1.0.



Test Accuracy: 0.466

Observations: The RNN model is trained with 256 hidden units, a dropout rate of 0.3, and the Adam optimizer with a learning rate of 0.0005 over 20 epochs. Despite achieving around 70% training accuracy, the validation accuracy fluctuates between 40-55%, indicating potential overfitting.

Model 2: hyperparameters: optim.Adam, 256 hidden dimensions, 10 epochs , learning rate of 0.0001, Adam (with weight_decay=1e-5 for L2 regularization), Gradient clipping value- 1.0, patience = 3, batch size of 64, also using early stopping patience of 3.



Test Accuracy: 0.4163

Observations:

- The training loss decreases steadily, and the validation loss decreases initially but fluctuates slightly, indicating some overfitting, though less severe than the previous model.
- Training accuracy improves consistently, while validation accuracy shows slight fluctuations, suggesting reasonable generalization.

Summary of RNN:

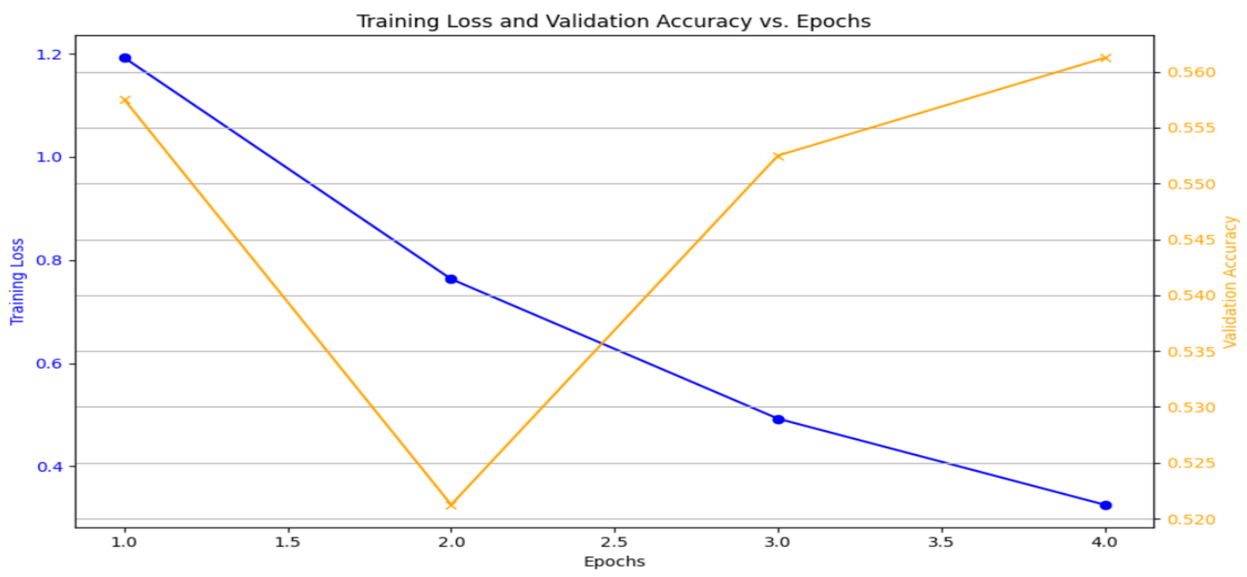
This model benefits from better regularization due to the higher dropout rate (0.5) and weight decay (1e-5), which help reduce overfitting and improve generalization. The lower learning rate (0.0001) ensures more stable learning, and early stopping prevents overtraining when validation performance plateaus. As a result, this model generalizes more effectively compared to the previous RNN.

4 Analysis (bonus: 10pt)

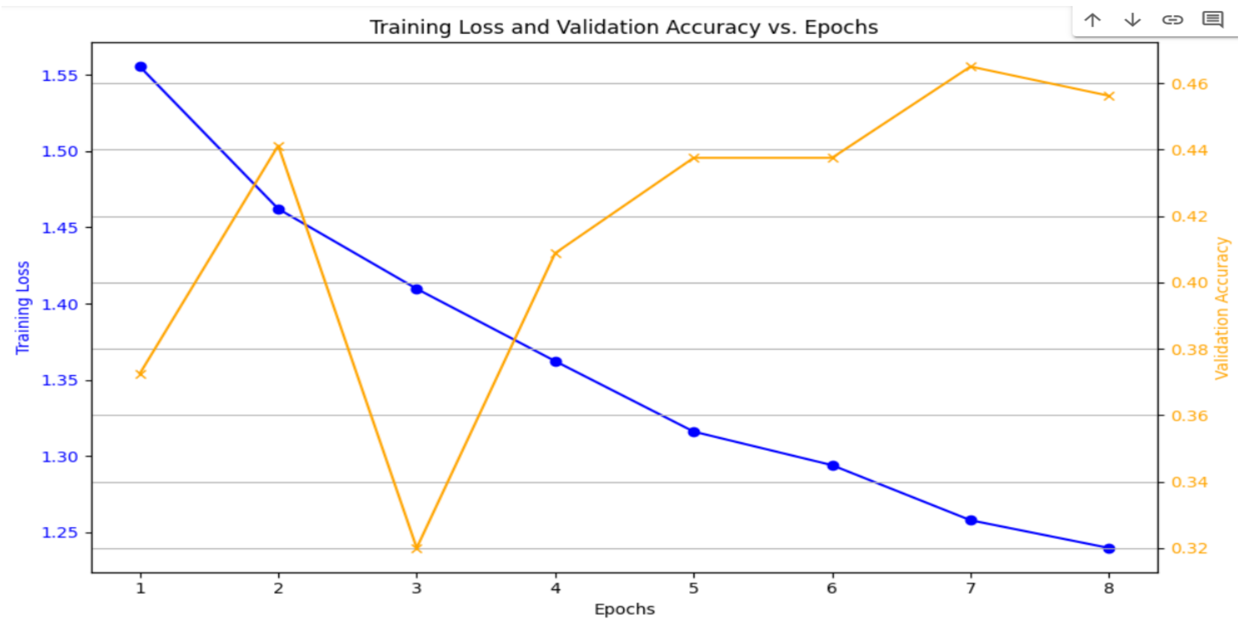
TODO:

- (5pt) Plot the learning curve of your best system. The curve should include the trainingloss and development set accuracy by epoch.

FFNN:



RNN:



- (5pt) Error analysis. List some (one or more than one) error examples and provide some analysis. How might you improve the system?

Further variations in hyperparameters such as dropout rates or learning rates can be tested for better generalization. The **early stopping** mechanism proved effective in maintaining validation stability.

Conclusion and Others (5pt)

TODO:

- Individual member contribution.

Successfully implemented different aspects of the project, including data preprocessing, model design, implementation of RNN and FFNN architectures, and the evaluation of models. I have collaborated closely to troubleshoot issues, especially during model training and optimization.

- Feedback for the assignment. e.g., time spent, difficulty, and how we can improve.

Despite the significant time invested, the learnings achieved were minimal compared to the effort required. A large portion of the time was spent dealing with runtime execution errors, even when using Google Colab with GPU support. The assignment felt more focused on technical troubleshooting rather than deepening conceptual understanding of model training. Reducing such technical barriers or providing more guided resources could help improve the balance between effort and learning outcomes.