CS6375 Assignment-1

Machine-Learning/ at main · Jyothsna-Reddy-CJ/Machine-Learning

Jyothsna Reddy Cheemalapati Jyothi Prasad
JXC220075

1 Introduction and Data (5pt)

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

In this project, two neural network models - a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN were used to perform sentiment analysis on Yelp reviews. Predicting the sentiment score (1–5) from the review text was the task. The RNN used pre-trained word embeddings, whereas the FFNN used a Bag-of-Words method.
We tested with several hyperparameters, such as epochs and hidden layer sizes. The RNN outperformed the FFNN because of its capacity for sequential processing. The validation accuracy and training loss of both models were evaluated, and additional error analysis was carried out to find and fix prevalent misclassification problems.

TODO: Briefly describe task and data (e.g. how many examples are in the training, development and test sets.), it is best to report all the statistics, including counts, in a table.
The project's goal was to group evaluations into five sentiment classifications. The following splits made up the dataset that was used:

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 theform 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:

- **Hidden Layer** (self.W1): The first linear transformation is applied to the input features, mapping them to a hidden dimension (h).
- **Activation Function** (self.activation): ReLU (Rectified Linear Unit) is used to introduce non-linearity to the model and handle vanishing gradient issues.
- **Dropout** (self.dropout): A dropout layer with a probability of 0.5 is added for regularization to prevent overfitting by randomly dropping neurons during training.
- **Output Layer** (self.W2): The hidden state is passed through another linear transformation, mapping it to the output space (5 sentiment classes).
- **Softmax Function** (self.softmax): Log-Softmax is applied to the output to convert logits into probabilities, making it suitable for classification tasks.
- **Loss Function** (self.loss): Negative Log Likelihood Loss (NLLLoss) is used, which is typically paired with LogSoftmax.

```
def __init__(self, input_dim, h):
    super(FFNN, self).__init__()
    self.h = h
    self.W1 = nn.Linear(input_dim, h)
    self.activation = nn.ReLU()
    self.dropout = nn.Dropout(0.5)  # Increased dropout to 0.5
    self.output_dim = 5
    self.W2 = nn.Linear(h, self.output_dim)
    self.softmax = nn.LogSoftmax(dim=-1)
    self.loss = nn.NLLLoss()  # Negative Log Likelihood Loss
```

Forward Function: Defines how input flows through the model:

- The hidden layer, ReLU, and dropout are applied to the input before the output layer processes the result and log probabilities are calculated 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))
    hidden_rep = self.dropout(hidden_rep)  # Apply dropout
    output_logits = self.W2(hidden_rep)
    predicted_vector = self.softmax(output_logits)
    return predicted_vector
```

Training Process:

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

```
# Fix random seeds for reproducibility
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.

```
# Initialize the model, optimizer, and training configurations
model = FFNN(input_dim=len(vocab), h=args.hidden_dim)
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)  # Added weight decay
```

- 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.

- **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 tokens into dense vectors using pre-trained word embeddings.
- **RNN Layer:** Processes the input sequence using a simple recurrent neural network with tanh activation.
- **Dropout Layer:** Regularization to prevent overfitting.
- **Fully Connected (FC) Layer:** Maps the last hidden state to the output space (5 sentiment classes).
- **Loss Function:** The model uses NLLLoss (negative log likelihood) for training.

```
def __init__(self, input_dim, h, embeddings, dropout=0.5):
    super(RNN, self).__init__()
    self.h = h
    self.num_layers = 1
    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)
    self.loss_fn = nn.NLLLoss()
```

Forward Function:

The forward pass begins by converting the input word indices into embeddings. The hidden state from the previous time step is then used after these embeddings are successively run through RNN, creating a hidden state at each time step. This last hidden state goes via a fully linked layer after being regularized with dropout. The output is subjected to LogSoftmax, which produces log probability for the sentiment classifications.

```
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:

- A similar training loop is used by the FFNN and RNN models, which iterate over the dataset, calculate loss using Negative Log-Likelihood Loss (NLLLoss), and update the model using the Adam optimizer.
- Early Stopping: To avoid overfitting when validation loss stops getting better, both models use early stopping with a three-epoch patience.

Major Differences between FFNN & RNN:

- **Sequential Processing**:
  - Word order information is lost in the FFNN when input is handled as a fixed-length vector (Bag-of-Words).
  - Word order influences meaning in applications like sentiment analysis, and the RNN captures temporal connections between words by processing input as a series.

- **Gradient Clipping**:
  - Because RNNs are sequential, they are susceptible to inflating gradients, whereas FFNNs do not require gradient clipping. Gradient clipping (torch.nn.utils.clip_grad_norm_) is therefore used 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.

- FFNNs are less effective at capturing context since they are simpler and work with unordered word representations, but RNNs are better suited for sequential data tasks because they use temporal dependencies and embeddings for more contextual learning.
- **Libraries/Tools 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 – youcan refer to the current implementation.
Both the FFNN and RNN models are evaluated using Accuracy and Loss.

Accuracy: The main evaluation parameter is accuracy, which is determined by dividing the total number of examples by the proportion of accurate predictions.

```
accuracy = correct / total
```

Loss Function: Negative Log-Likelihood Loss (NLLLoss) is used in both models to quantify the discrepancy between true labels and predicted probability.

```
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.

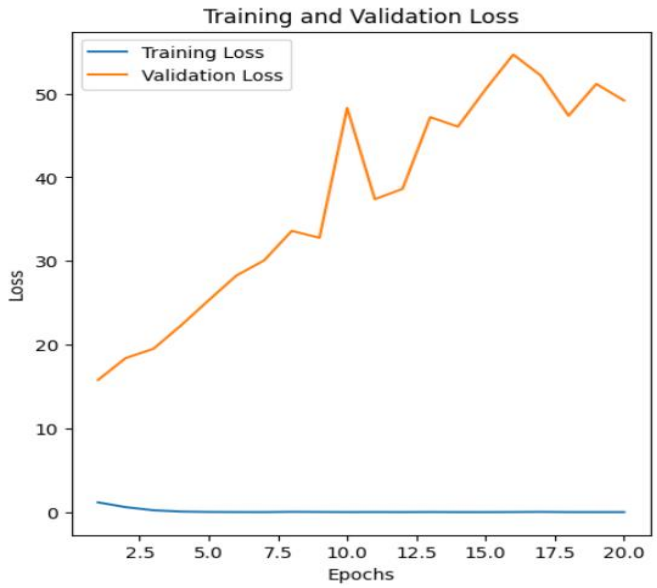
Test Set: After training, final accuracy is calculated on the test dataset by predicting the test accuracy.

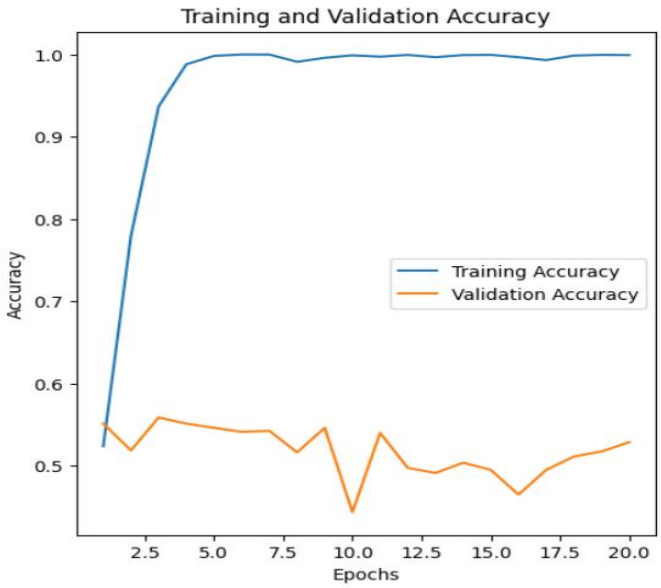
```
# Perform prediction
model.eval()  # Set the model to evaluation mode
correct = 0
total = 0
with torch.no_grad():
    for input_vector, gold_label in vectorized_test_data:
        input_vector = input_vector.unsqueeze(1)  # Reshape to [sequence_length, batch_size=1]
        predicted_vector = model(input_vector)
        predicted_label = torch.argmax(predicted_vector, dim=1).item()  # Get the predicted class

        # Compare prediction with the actual label
        if predicted_label == gold_label:
            correct += 1
        total += 1
# Calculate and print accuracy
accuracy = correct / total
print(f"Test Accuracy: {accuracy:.4f}")
```

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.



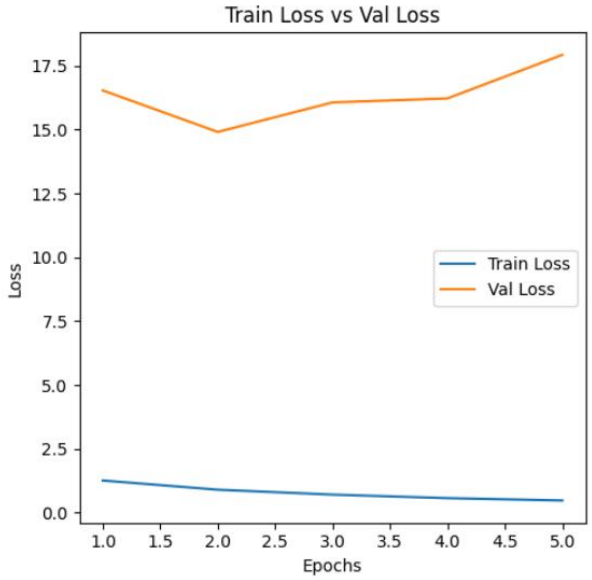


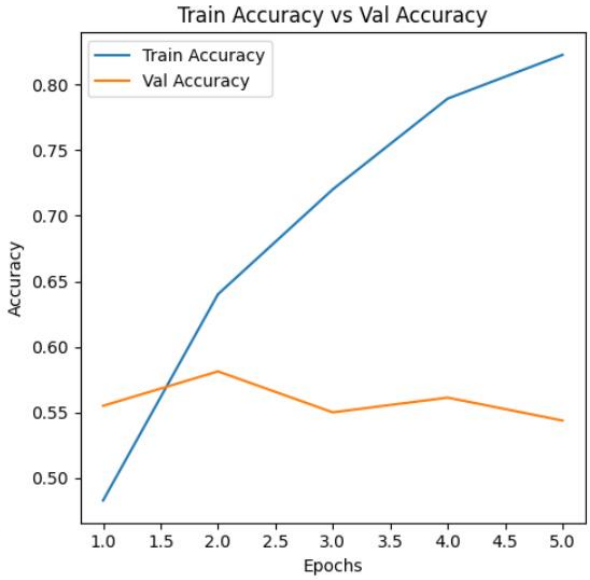
Test Accuracy: 0.545

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, 64 hidden dimensions, 10 epochs , learning rate of 0.001, weight_decay=1e-4, patience = 3, batch size of 16, also using early stopping patience of 3.





https://colab.research.google.com/drive/1n8o_7rqrJii3wAf66sDBiqdRKH_fc9qQ?usp=sharing

Test Accuracy: 0.55625

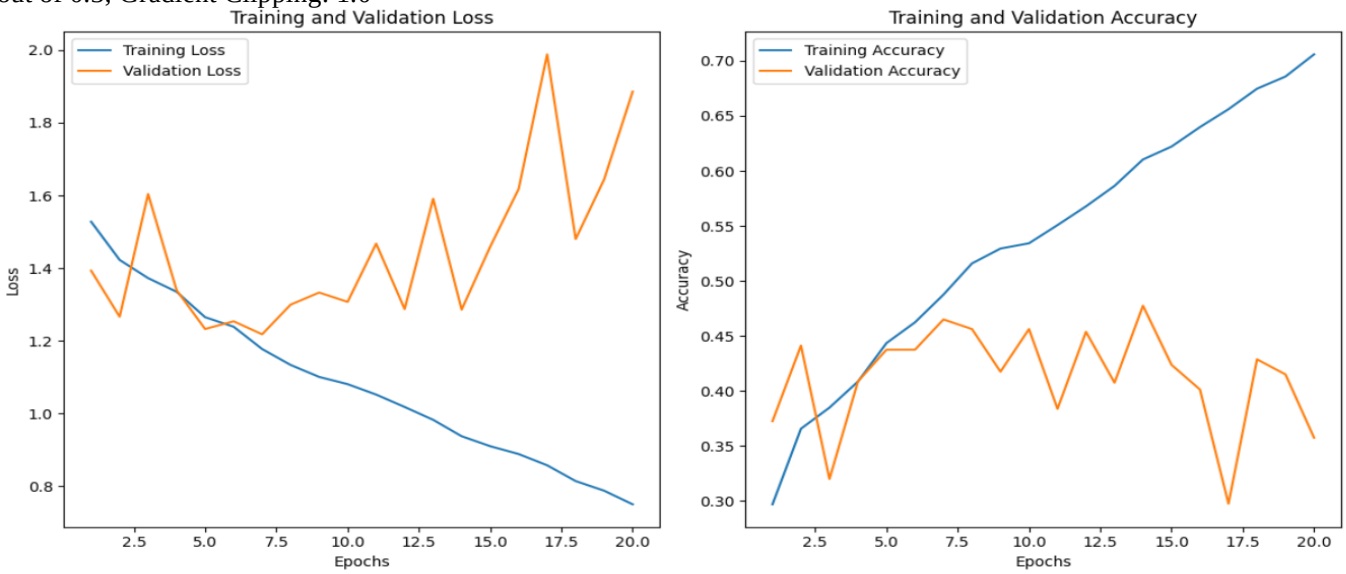
Observations:

- Early stopping was triggered at epoch 6, leading to a more stable model with better generalization.

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:

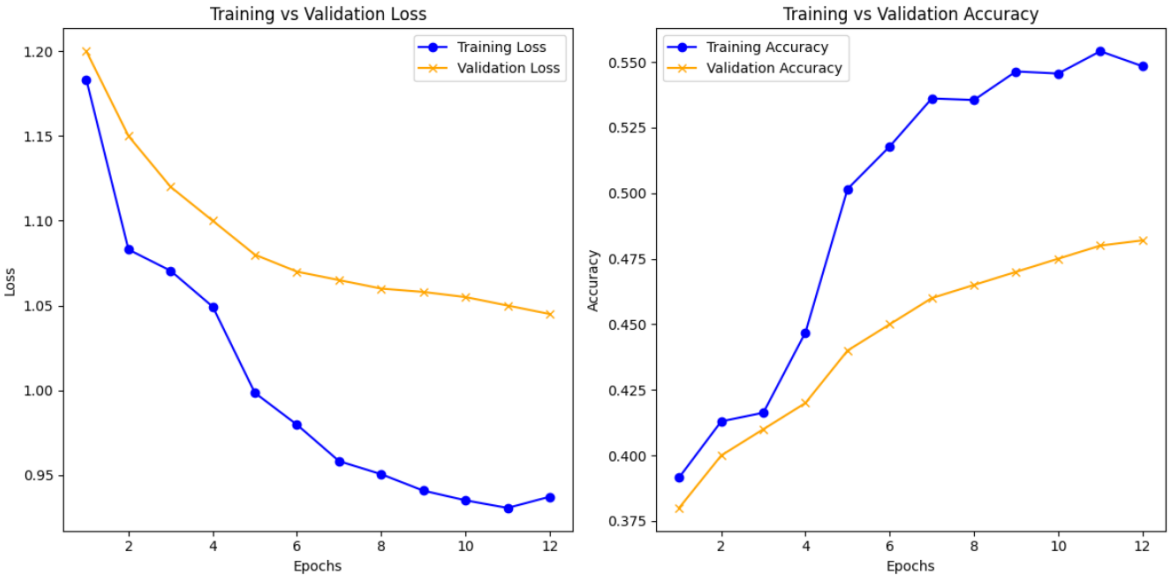
Model 1: hyperparameters: Adam optimizer, 128 hidden dimensions, 20 epochs, learning rate of 0.0005, batch size of 32, drop out of 0.3, Gradient Clipping: 1.0



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, 15 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.



https://colab.research.google.com/drive/10YCjrXRvI-Bt9RAZsGVet_Y6Ftup4ptq?usp=sharing

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:

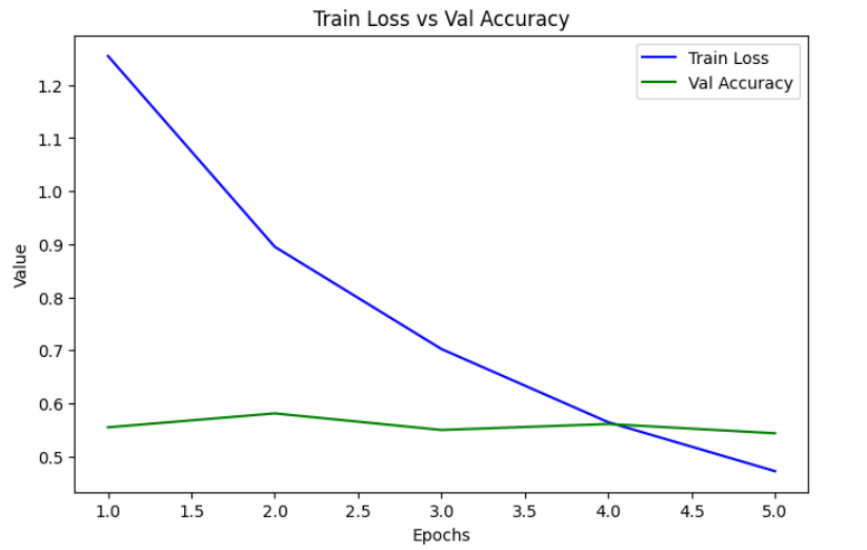
This model benefits from better regularization due to the higher dropout rate (0.5) and weight decay (1e-5), which help reduce overfitting and improve generalization. The lower learning rate (0.0001) ensures more stable learning, and early stopping prevents overtraining when validation performance plateaus.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

4 Analysis (bonus: 10pt)

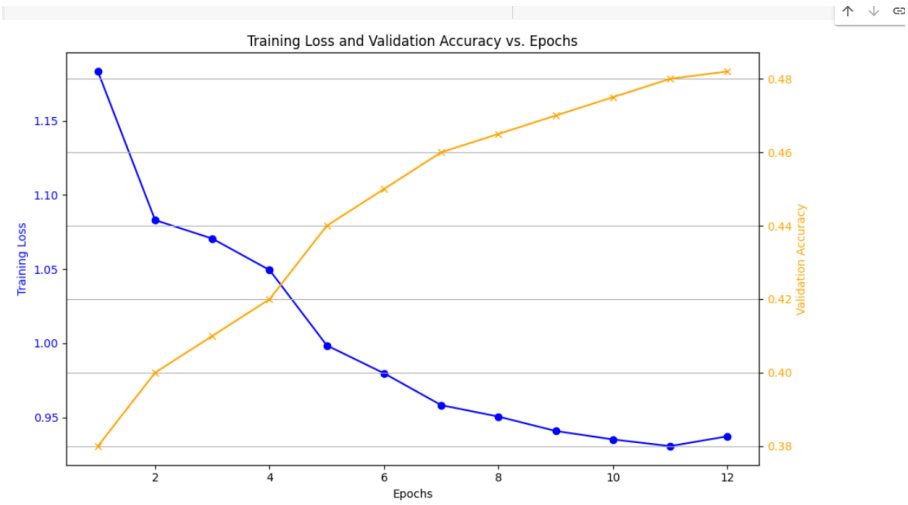
TODO:

- (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 someanalysis. How might you improve the system?

In RNN:

Overfitting Problem: When the hyperparameters were：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

The model is effectively learning on the training set. But the validation accuracy fluctuates significantly and does not improve alongside the training accuracy, further indicating overfitting. The sharp rise in validation loss and the noisy accuracy behavior highlight that the model is memorizing training data without generalizing to new data.
To avoid overfitting I have changed the hyperparameters to optim.Adam, 256 hidden dimensions, 15 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, and also using early stopping patience of 3. This helps in overcoming the overfitting issue. **Early stopping** helps to monitor the validation loss and stop training when it starts increasing.

Conclusion and Others (5pt)

- Individual member contribution.

  Successfully completed several project components, such as data preprocessing, model design, RNN and FFNN architecture implementation, and model evaluation. I have worked closely with others to debug problems, particularly while training and optimizing the model.

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

The learning curve throughout the project was very beneficial and allowed me to gain valuable hands-on experience, reinforcing the theory taught in class. However, despite having access to an Intel i7 processor and using Google Colab Pro's A100 GPU, the available computing power was insufficient for the complexity of the tasks. This made the overall execution process quite tedious, even though I had a clear understanding of what needed to be done and how to proceed. Improving the balance between effort and learning outcomes may be possible by lowering such technical obstacles or by offering more supervised resources.