CS6320 Assignment 2

GitHub - pinkygh/NLPAssignemnt2

GROUP 11 Jyothsna Reddy Cheemalapati (JXC220075), Sai Premanvitha Kadiyala (SXK230035), Yamini Sunkari (YXS220036), Jyothi Prasad

1 Introduction and Data (5pt)

In this project, two neural network models—a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN)—were employed for sentiment analysis on Yelp reviews. The task was to predict sentiment scores (1–5) from review text. The RNN, utilizing pre-trained word embeddings, outperformed the FFNN, which relied on the Bag-of-Words approach, due to its ability to process sequential data effectively. Hyperparameters such as epochs and hidden layer sizes were tested, and evaluation included validation accuracy, training loss, and error analysis to address misclassification issues.

The dataset consists of text reviews, each labeled with a sentiment score (1–5).

Dataset	Number of Samples
Training	16000
Validation	800
Test	800

2. Implementations (45pt)

2.1FFNN (20pt) Model Architecture Summary:

- **Hidden Layer (self.W1):** The first linear transformation is applied to input features, mapping them to hidden dimension (h).
- **Activation (self.activation):** ReLU introduces non-linearity and mitigates vanishing gradient issues.
- **Dropout (self.dropout):** A 0.5 probability dropout layer prevents overfitting by randomly deactivating neurons during training.
- **Output Layer (self.W2):** Maps the hidden state to 5 sentiment classes.
- **Softmax (self.softmax):** Converts logits to probabilities using Log-Softmax for classification.
- **Loss Function (self.loss):** Negative Log Likelihood Loss (NLLLoss) calculates the loss between predicted log probabilities and true labels.

```
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.

```
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 help mitigate 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 is 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)

Model Architecture Summary

- **Embedding Layer:** Transforms 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:** Applies 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

    self.loss_fn = nn.NLLLoss()
```

Forward Function:

The forward pass starts by converting input word indices into embeddings. These embeddings are processed through the RNN, generating a hidden state at each time step. The final hidden state is regularized with dropout and passed through a fully connected layer. LogSoftmax is then applied to produce log probabilities for 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**: The FFNN loses word order information due to its fixed-length vector representation (Bag-of-Words), whereas the RNN captures temporal relationships between words, preserving the sequence for tasks like sentiment analysis.
- **Gradient Clipping:** RNNs require gradient clipping (e.g., `torch.nn.utils.clip_grad_norm_`) to stabilize learning, unlike FFNNs.

- **Word Representation:** FFNNs use Bag-of-Words, treating words as independent, while RNNs leverage pre-trained embeddings for richer semantics.
- **Context Capture:** FFNNs struggle with context, whereas RNNs excel in sequential tasks through temporal dependencies and embeddings.

3 Experiments and Results (45pt)

Evaluations (15pt)

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.

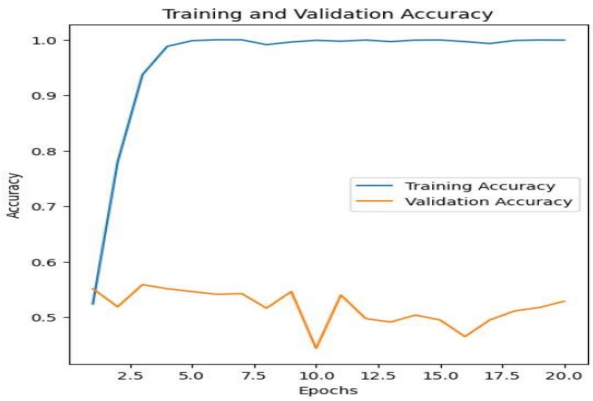
```
print("========== Testing ==========")
model.eval()
correct = 0
total = 0

with torch.no_grad():
    for input_vector, gold_label in test_data:
        input_vector = input_vector.float()  # Ensure input is a float tensor

        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)

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.

https://colab.research.google.com/drive/1Z-HzE7mPrpiTbehDGjIf05ODgQtGF1r9?usp=sharing

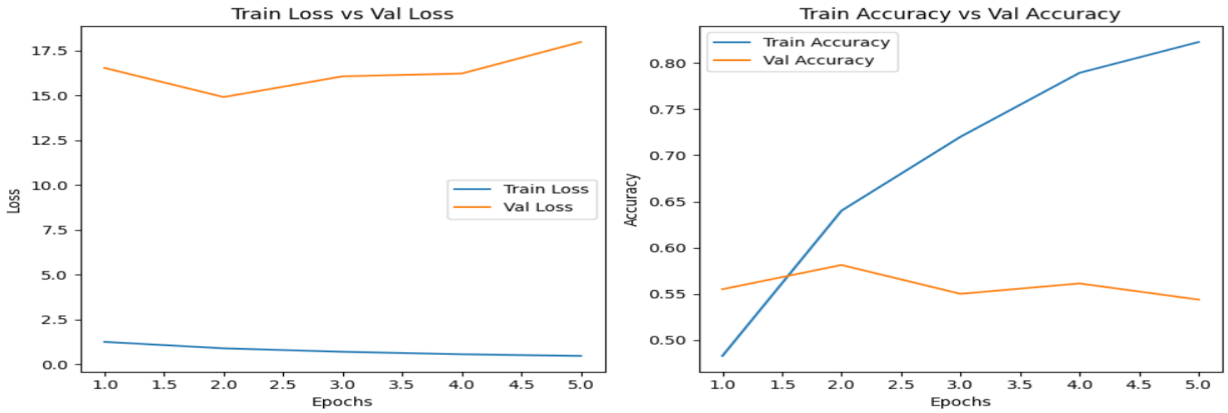


Test Accuracy: 0.545

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.



Test Accuracy: 0.55125

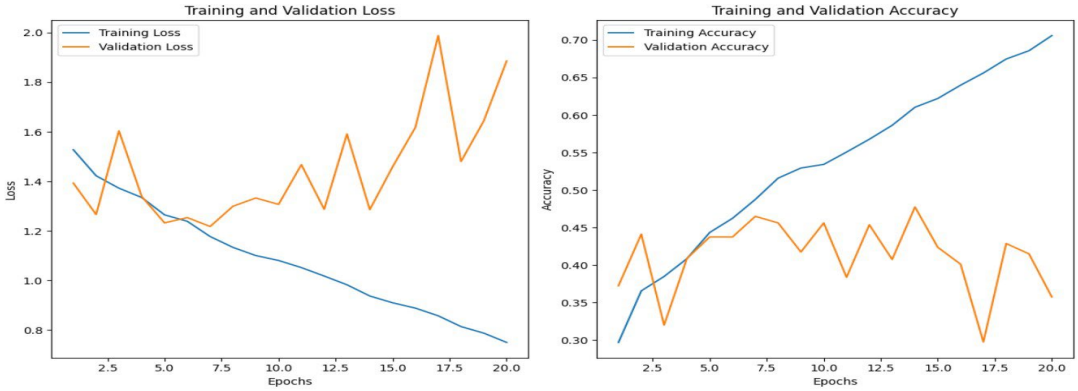
Observations: Early stopping was triggered at epoch 6, leading to a more stable model with better generalization.

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:

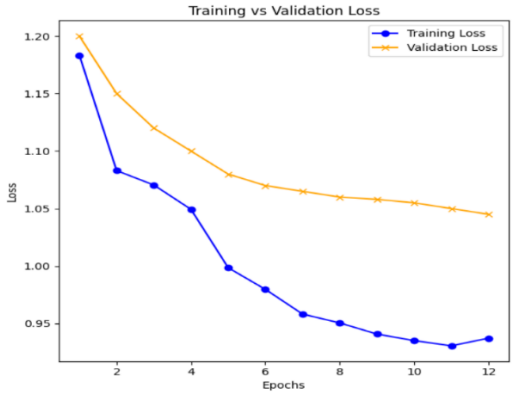
Model 1: hyperparameters: Adam optimizer, 128 hidden dimensions, 20 epochs, learning rate of 0.0005, batch size of 32, drop out of 0.3, Gradient Clipping: 1.0

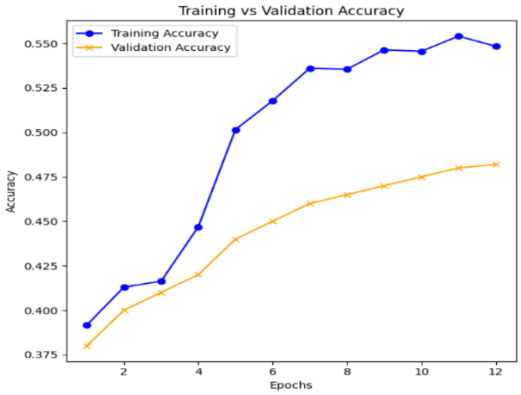


Test Accuracy: 0.466

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 4055%, 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/1d8m5x97SnJVyBVglMCl6CfhuJtGLaVb_?usp=sharing

Test Accuracy: 0.4163

Observations:

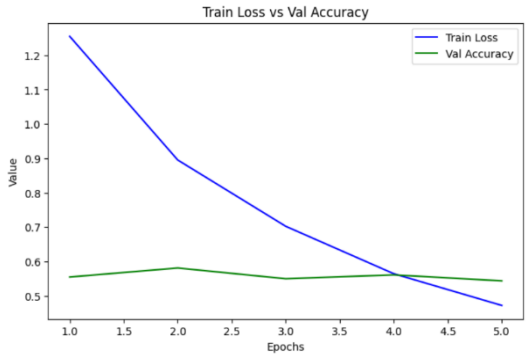
- The training loss decreases steadily, and the validation loss decreases initially but fluctuates slightly, indicating some overfitting, though less severe than the previous model.
- Training accuracy improves consistently, while validation accuracy shows slight fluctuations, suggesting reasonable generalization.

Summary of RNN:

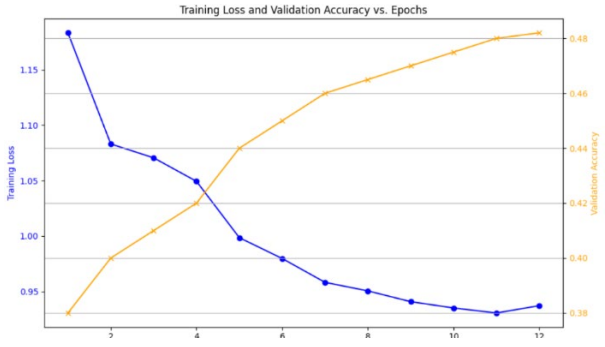
This model benefits from better regularization due to the higher dropout rate (0.5) and weight decay (1e-5), which help reduce overfitting and improve generalization. The lower learning rate (0.0001) ensures more stable learning, and early stopping prevents overtraining when validation performance plateaus.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

4 Analysis (bonus: 10pt)

FNN



RNN



In RNN:

Overfitting Problem The initial hyperparameters (Adam optimizer, 128 hidden dimensions, 20 epochs, 0.0005 learning rate, batch size 32, dropout 0.3, gradient clipping 1.0) led to overfitting, with fluctuating validation accuracy and rising loss. Adjustments included 256 hidden dimensions, 15 epochs, 0.0001 learning rate, L2 regularization (weight_decay=1e-5), batch size 64, and early stopping (patience 3), improving generalization and stability. Conclusion and Others (5pt)

- **Individual member contribution:**
  Prema: Implemented the FFNN forward function, achieved model accuracy, and documented the FFNN approach.
  Jyothsna: Developed the RNN model, including the forward pass, accuracy computation, and provided RNN-specific documentation and implemented early stopping techniques
  Yamini: Handled hyperparameter tuning, plotted training/validation accuracy/loss graphs, and documented the corresponding process.
- **Feedback**: The assignment took a fair amount of time, especially with hyperparameter tuning and debugging the forward () function for both RNN and FFNN. The difficulty was moderate, mainly due to issues with overfitting and handling the RNN's sequential nature. Improvements could include more guidance on hyperparameter selection and clearer example outputs for the report.