CS6320 Assignment 2

https://github.com/LeoLuo0115/CS6320-NLP

Group 18	Peter Driscolll	Jiyuan Luo	Huiwen Xue
	Pwd170030	Jxl1220083	Hxx171530

1 Introduction and Data (5pt)

In this project the data is a short paragraph of text followed by a label. The goal is to be able to correctly classify the text. The data has already been cleaned and implementations for converting the text to input for the neural networks were already provided. The two processing methods provided were **bag-of-words** (BoW) for the FFNN and **word embedding** for the RNN

- **Bag-of-Words:** Reads in the input and converts it to vector of size vocab. Each word has a specific index and the value at that index in the vector is the total count of that word in the document.
- **Word embedding:** Using a pre-trained model (word_embedding.pkl) tokens aka words are converted to a vector of fixed size which indicates their meaning these vectors are usually much smaller than the vocab size and are denser than bag of words vectors.

2 Implementations (45pt)

Both implementations had minor modifications outside the forward section to move the model and training validation data over to a GPU for faster training

2.1 FFNN (20pt)

2.1.1 Forward Portion

This FFNN has a structure of **input layer →hidden layer →output layer** with only 1 hidden layer. The only portion we implemented was the forward function where we applied the weights W1 to the input vector, applied the ReLu activation function to add some non linearity, then applied W2 to the output of the hidden layer and passed it to the output layer of size 5. These 5 neurons indicate which class the FFNN is most likely predicting and a softmax must be applied to the output layer so the total probabilities over the 5 output nodes sums to 1.

```
def forward(self, input_vector):
    # First layer: input to hidden
    hidden = self.W1(input_vector)

    # Apply ReLU activation function
    hidden = self.activation(hidden)

    # Second layer: hidden to output
    output = self.W2(hidden)

    # Apply softmax to get probability distribution
    predicted_vector = self.softmax(output)

    return predicted_vector
```

2.1.2 Preprocessing

For the provided pre-processing code the functions **make_vocab, make_indices, load_data, convert_to_vector_representation** all are used to preprocess and prepare the data into BoW format basically reading in the file getting the vocab size between all the different documents then reading in the documents along with their associated label. Afterwards the documents are converted into a tensor where the first portion is a sparse vector of size vocab with the associated BoW counts and the second portion is the correct label for that document.

2.1.3 Training

In the training process the data is loaded up using the previously mentioned functions and the training of the model is run for however many number of epochs specified. Here the training data is broken up into mini batches training the data on a small subset at a time. This is beneficial because we don't train on only 1 item at a time causing many updates that may be seemly random (if that individual data item is bad) or update weights on the entire set at once which would lead to over-fitting for training. Data is input into the model via forward then the model makes a prediction. These are then compared to the actual labels to know if predictions are correct and to calculate loss. For the loss portion in each prediction in the mini batch is compared to its golden label incorrect predictions count as loss and this is aggregated over all items in the mini batch. This aggregated mini batch loss is then used to update the weights via gradient descent. The loss is put through a backwards pass where it calculates the gradients for the loss function in respect to the weights W1 and W2. By using these weight gradients we can correctly shift W1 and W2 in the opposite direction of their gradient which will reduce loss making our model correctly predict classifications for documents.

```
for minibatch_index in tqdm(range(N // minibatch_size)):
    optimizer.zero_grad()
    loss = None
    for example_index in range(minibatch_size):
        input_vector, gold_label = train_data[minibatch_index *
            minibatch_size + example_index]

        # Move data to device
        input_vector = input_vector.to(device)
        gold_label = torch.tensor([gold_label], device=device)

        predicted_vector = model(input_vector)
        predicted_label = torch.argmax(predicted_vector)
        correct += int(predicted_label == gold_label.item())
        total += 1
        example_loss = model.compute_Loss(predicted_vector.view(1, -1),
            gold_label)
        if loss is None:
            loss = example_loss
        else:
            loss += example_loss

    loss = loss / minibatch_size
    loss.backward()
    optimizer.step()
```

2.2 RNN (25pt)

The structure of our RNN is **Input Layer →RNN Layer →Output Layer**. The training through gradient descent is essentially the same in concept to the FFNN where you calculate the loss gradients in respect to the weights over mini batches then attempt to update the weights to reduce loss. The RNN layer is the major difference which attempts to capture context between words. This ability to store context is done through hidden and observed state in the RNN where the observed states are the word embeddings and the hidden states store the previous contextual information. An activiation function in this case tahn is introduced to create non linearity and this is repeated for every word in the document until the last hidden state which in theory has combined all the relevant information from previous words and hidden states. The last hidden state is then output to a 5 node linear layer which has softmax applied to break it into a probability distribution over the 5 different classes.

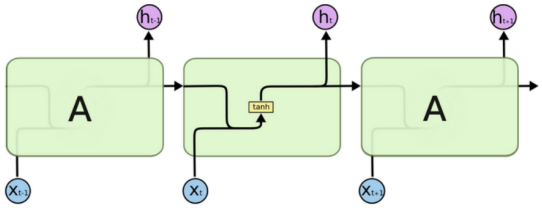

Figure 1: RNN Structure. Source: CS 6320 08-RNN slides

3 Experiments and Results (45pt)

3.1 FFNN (experiment documentation URL - click to view)

During our initial experiments, we observed that simply increasing the hidden dimensions from 32 to 128 only marginally improved the model's performance, with accuracy hovering around 58-59%. This prompted us to investigate potential overfitting issues.

We introduced dropout regularization with various rates (0.1, 0.3, and 0.5) to see if it would help. After testing different combinations, we found that a hidden dimension of 64 with a dropout rate of 0.3 gave us a slight improvement, pushing the validation accuracy to about 61%. While higher dropout rates of 0.5 seemed to stabilize the validation performance, they also limited the model's ability to learn effectively.

After 4 hours of training with various configurations, 61% was the highest validation accuracy we could achieve. This suggests that the limited performance might be due to either insufficient training data or the simple single-layer architecture of our network.

3.2 RNN (experiment documentation URL - click to view)

During our RNN training experiments, we found that increasing the hidden dimensions initially led to minor gains, yet required significantly longer training times. For example, using 128 hidden units produced only a small improvement in accuracy, which seemed disproportionate to the additional processing time required. However, when we adjusted the hidden dimensions to 64, we observed a steadier convergence pattern, suggesting that this configuration balanced complexity and training efficiency more effectively.

Dropout regularization was another key factor in our tests. We experimented with various dropout rates and found that moderate rates, such as 0.3, maintained model stability while also achieving slight gains in validation accuracy.

Overall, the model's validation accuracy showed signs of plateauing with the current architecture, hinting that more substantial gains may require either additional training data or adjustments to the model's structure. These insights suggest a promising direction for future experiments, potentially involving deeper architectures or techniques that enhance generalization without compromising training efficiency.

4 Analysis (bonus: 5pt)

4.1 (5pt) Plot the learning curve of your best system

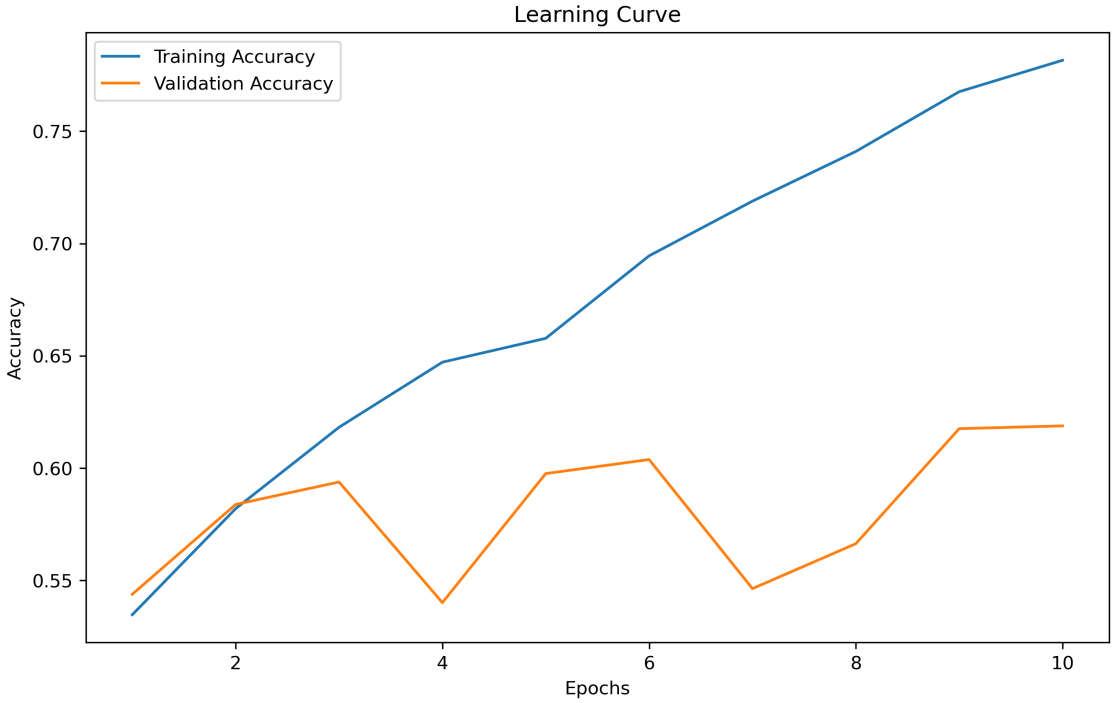

Figure 2: Learning curve of the FFNN model

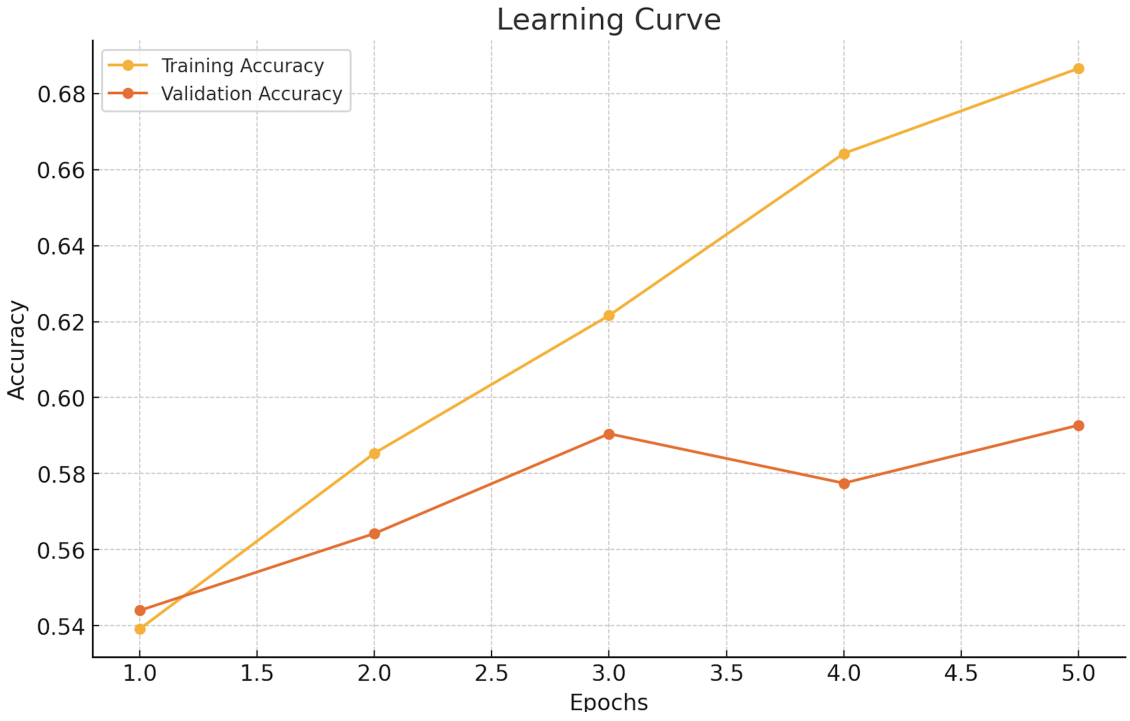

Figure 3: Learning curve of the RNN model

4.2 (5pt) Error

A likely source of error for the RNN would be a poorly trained word embedding model. If a word embedding is incorrect, it would likely lead to misclassification of the document. An example being if the word "shiver" was embedded as a dance move but we were trying to classify weather documents.

5 Conclusion and Others (5pt)

Each team member contributed significantly to the project, with well-defined roles and effective communication throughout the process. One member implemented the FFNN model, while another focused on the RNN architecture and data preprocessing. The remaining members assisted with the training loop and optimization tasks. As for feedback, the training time, especially for the RNN model, was longer than expected. In future work, exploring optimizations such as early stopping or reducing model complexity could help speed up training without sacrificing performance. Overall, the project was a valuable learning experience, and the team collaborated effectively to complete the assignment successfully.