ECE 661: Homework #3

Elisa Chen (Duke ID: eyc11)

2023-10-14

Q1 True / False Questions

Problem 1.1: **True** - The self-attention layer of the Transformer will generate new representations of the score, attention and output of the input. These are crucial components for capturing and modeling dependencies between words in the input sequence. To calculate the representations, we will need queries, keys and values that are computed during the self-attention layer.

Problem 1.2: **False** - In the self-attention layer, attention is denoted by the normalization of the score through softmax, and not by the cosine similarity between the key and query.

Problem 1.3: **True** - In the self-attention layer of Transformer models, it is common practice to apply normalization to the attention scores. Typically, a layer normalization is applied to the matrix to stabilize the training process.

Problem 1.4: **False** - Only the decoder of the Transformer learns auto-regressively where the model predicts future values or sequences based on previously generated values.

Problem 1.5: **False** - While GPT's architecture is a Transformer decoder, BERT on the other hand is autoencoding or a pre-trained encoder.

Problem 1.6: **True** - BERT has two pre-training objectives: 1) Masked language modeling in which the model attempts to predict words that are masked in the sequence, and 2) next sentence prediction where it is trying to predict the likelihood of two sentences between contiguous to each other.

Problem 1.7: **True** - Both GPT and BERT are capable of zero-shot prediction. However, the extent they can be transferred to unseen domains and tasks depends on the specific model and the quality of the data.

Problem 1.8: **False** - Gradient clipping is used to alleviate the gradient exploding problem instead of the gradient vanishing problem. Gradient clipping is a technique used to prevent the gradients from becoming too large.

Problem 1.9: **True** - Word embeddings are data vectors constructed from the context surrounding the word. Therefore, it is completely reasonable for a word embedding to contain both positive and negative numbers.

Problem 1.10: **False** - The memory cell of an LSTM is not computed as a weighted average of the previous memory state and the current memory state where the sum of weight equals to 1. Instead, the memory cell in an LSTM is updated through a series of operations involving the current input, previous memory state, and input and forget gates.

Q2 Lab 1

a): Please refer to the `LabRNN.ipynb` notebook for more details about the code implementation. The data loader function is called `load_imdb`. The dataset was split into three data sets: train, validation, and test by 7:1:2 ratio respectively. In total, we had 35k training samples, 10k test samples and 5k validation samples.

b): Please refer to the `build_vocab` function in the `LabRNN.ipynb` notebook for more details on the code implementation. In this code implementation we calculated the frequency of all the words in the training corpus and then removed any words that were part of the `STOP_WORDS` defined as part of the `HyperParams` object. All the words in the corpus were also lower-cased to avoid miscounts between capitalized and non-capitalized words. Lastly, words that didn't meet a certain frequency threshold were also removed from the vocabulary. The length of the vocabulary for this particular training dataset was 56,492.

c): Please refer to the `tokenize` function in the `LabRNN.ipynb` notebook for more details on the code implementation. This function returns a list of integers that represents the indices of words for a given string.

d): Please refer to the `IMDB` class in the `LabRNN.ipynb` notebook for more details on the code implementation. This function will tokenize the sentences in the dataset and appropriately decode the binary labels into a binary index.

e): Please refer to the `LSTM` class in the `LabRNN.ipynb` notebook for more details on the code implementation. The architecture contains an embedding layer, a LSTM cell that can be bidirectional, a dropout layer and a linear layer. If the LSTM is bidirectional then the hidden dimensions must be multiplied by 2 to account for the forward and backward hidden states. In the `forward` function of the model, we first pass the input through the embedding layer and pad the sequences into the same length so that we can pass the sequences through the lstm layer. We then take the hidden inputs and pass them through the dropout layer and lastly the linear layer to obtain our outputs. The dropout layer was applied prior to the linear layer to prevent overfitting and improve the model's generalization performance.

f): After training the LSTM model for 5 epochs with SGD Optimizer, we observe that the model doesn't seem to learn the data very effectively. The training and validation loss seem to oscillate around ~0.69 without any signs of decreasing. This is not the behavior we would expect if the model was learning after each epoch since we would expect the training loss to decrease over time. Please refer to Figure 1 for more details.

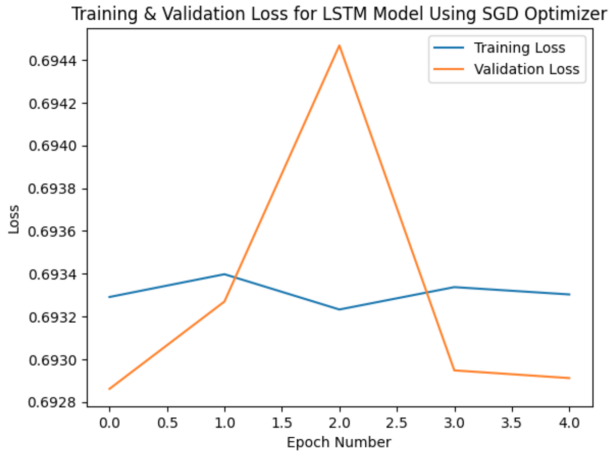

Figure 1: Training and Validation Loss for LSTM Using SGD Optimizer

g): Please refer to the `GRU` class in the `LabRNN.ipynb` notebook for more details on the code implementation. The architecture contains an embedding layer, a gated recurrent unit (GRU) that can be bidirectional, a dropout layer and a linear layer. If the GRU is bidirectional then the hidden dimensions must be multiplied

by 2 to account for the forward and backward hidden states. In the `forward` function of the model, we first pass the input through the embedding layer and pad the sequences into the same length so that we can pass the sequences through the GRU layer. We then take the hidden inputs and pass them through the dropout layer and lastly the linear layer to obtain our outputs. The dropout layer was applied prior to the linear layer to prevent overfitting and improve the model's generalization performance.

h): After training the GRU model for 5 epochs with SGD Optimizer, we observe that similar to the LSTM model, the GRU model doesn't learn the data very effectively. The training and validation loss seem to oscillate around ~0.69 as well without any signs of decreasing. This is not the behavior we would expect if the model was learning after each epoch since we would expect the training loss to decrease over time. Please refer to Figure 2 for more details. I hypothesize that this happens because we run into gradient explosion or vanishing problems. Due to the properties of certain activation functions, RNNs easily experience gradient vanishing or explosion problems that could occur even after a small number of timesteps.

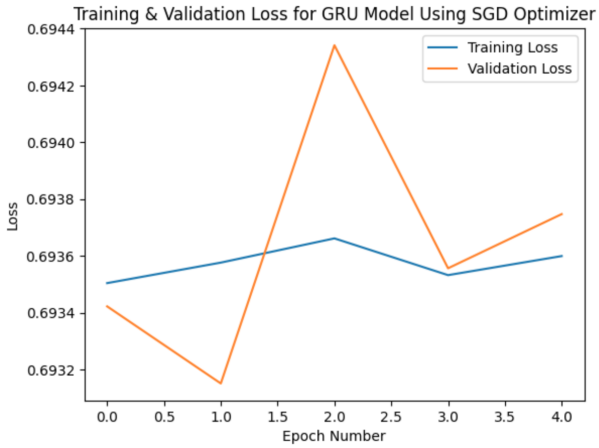

Figure 2: Training and Validation Loss for GRU Using SGD Optimizer

Q3 Lab 2

For details about each experiment, please refer to the Lab 2 portion in the `LabRNN.ipynb` notebook.

a): To address the potential vanishing / exploding gradient problem, we'll want to experiment with alternative optimizers with an adaptive learning rate schedule. Based on the observations, it's clear that rmsprop resulted in the lowest training and validation losses. Both rmsprop and adam optimizers were significantly better at training the model that was illustrated with the decreasing training (and validation) loss over time. Please refer to Figure 3 for more details. We'd expect rmsprop to be an efficient optimizer as it uses a moving average of incoming gradients as the denominator which implies that parameters with small gradients receive larger updates, which would help prevent the vanishing gradient problem.

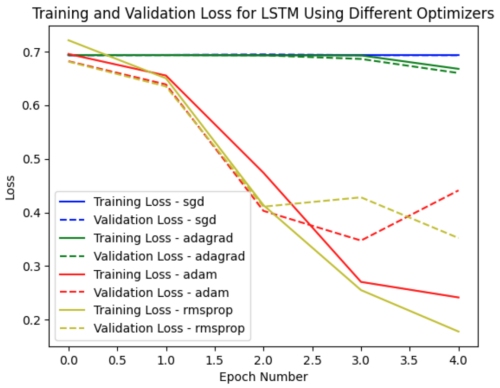

Figure 3: Training and Validation Loss for LSTM Using Different Optimizer

b): The results for GRU are fairly similar to that of LSTM in the sense that rmsprop achieved the lowest training loss. However, we see that with rmsprop (and adam) optimizer, the validation loss curve starts to increase a lot sooner than it does for LSTM. This could indicate that the GRU model learns faster than the LSTM model as we're observing potential overfitting of the GRU model during the later epochs. As opposed to the LSTM model, the adagrad optimizer worked relatively better on the GRU model. Please refer to Figure 4 for more details. In terms of model performance, the GRU model obtained a test accuracy of 86.2% whereas the LSTM model obtained a test accuracy of 85.7% when using the rmsprop optimizer. Overall, the GRU model seems to be learning faster and achieving better performance compared to the LSTM model.

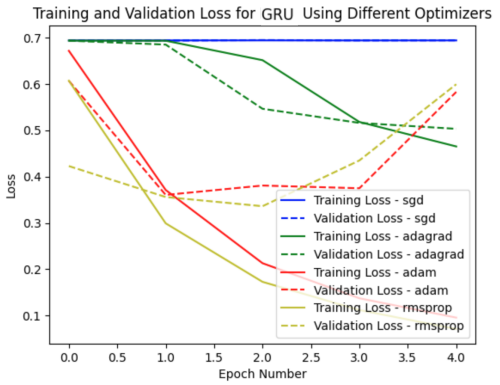

Figure 4: Training and Validation Loss for GRU Using Different Optimizer

c): To experiment how the number of layers would impact model performance, we compared the results on a LSTM model with 1, 2, 3 and 4 layers. It appears that the number of layers doesn't have a significant effect on model performance in either direction. We do observe a slight improvement to the baseline model when we use 2 layers instead of 1 achieving a test accuracy of 85.9% vs. 85.2%. Interestingly, the performance decreases by over an percentage when we increase the number of layers to 3, but achieves a reasonable performance once we increase it further to 4 layers. Please refer to figure 5 for more details.

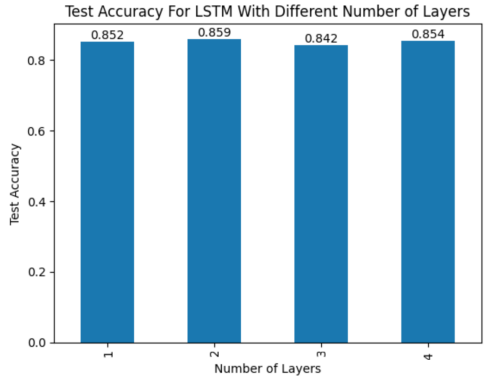

Figure 5: Test Accuracy for LSTM With Different Number of Layers

d): To experiment how the number of hidden units would impact model performance, we compared the results on a LSTM model with 50, 100, 150, 200, 250, and 300 hidden units. It appears that the number of hidden units does have a significant impact on model performance: increasing the number of hidden units above 100 seems to negatively impact the test accuracy by reducing it 30%+ in the most severe instances. The ideal number of hidden units is likely between 0 - 100. Please refer to figure 6 for more details.

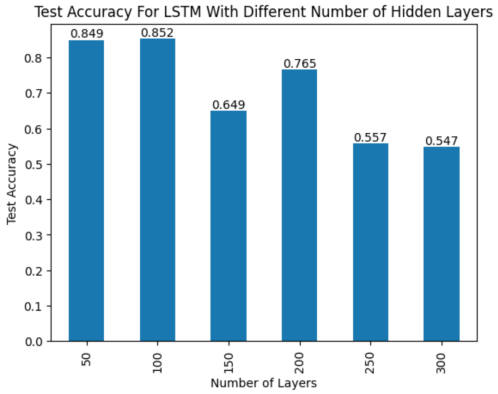

Figure 6: Test Accuracy for LSTM With Different Number of Hidden Units

e): To experiment how the embedding size of the input would impact model performance, we compared the results on a LSTM model with 16, 32, 64, 128, and 256 embedding dimensions. From the experiment we learned that smaller embedding sizes like 16, 32 and 64 performed relatively better than the baseline model with an embedding size of 1 or when the embedding size was larger than 64. The highest test accuracy of 86.4% was achieved when embedding size was 32. Please refer to figure 7 for more details.

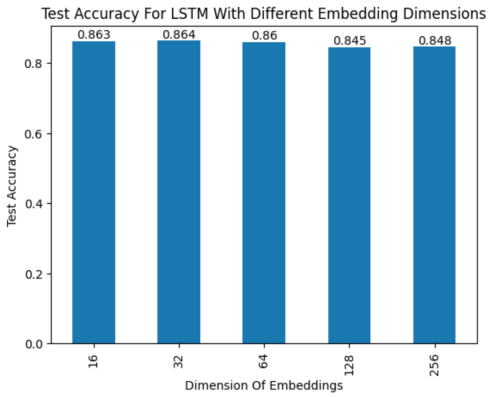

Figure 7: Test Accuracy for LSTM With Different Embedding Sizes

f): To experiment how simultaneously scaling up the number of hidden units, number of layers, and embedding dimensions would impact the model performance, we ran the model 32 times using an exhaustive grid search against the following parameter values:

- `num_layers = [1, 2, 3, 4]`
- `hidden_units = [50, 100]`
- `embedding_sizes = [16, 32]`
- `learning_rates = [0.001, 0.0001]`

Learning rate was also considered as part of the experimentation. The most optimal model achieved a test accuracy of ~87% with the following parameters: {'num_layer': 4, 'hidden_unit': 100, 'embedding_size': 32, 'learning_rate': 0.001}. The parameters obtained are mostly aligned with our expectations observed in parts c) - e).

g): Please refer to the `LSTM` class in the `LabRNN.ipynb` notebook for more details on the code implementation for a bidirectional layer. To account for bidirectional LSTMs, we have to ensure that the forward and backward hidden states are concatenated prior to passing them into the linear layer. After running the most optimal model achieved in part f) bidirectionally, we achieve a test accuracy of 86.8%, which is slightly lower than the baseline test accuracy of ~87%. We most likely do not observe better performance with a bidirectional model, because we might be overfitting to our limited training data. This is further supported by the training and validation loss curves illustrated in figure 8 where the validation loss starts increasing rapidly after the second epoch.

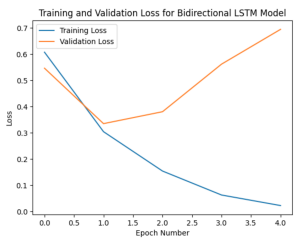

Figure 8: Training and Validation Loss For Bidirectional LSTM Model