ECE 661: Homework #3
Understand and Implement Sequence Models

Frankie Willard

ECE Department, Duke University — September 15, 2022

1 True/False Questions (30 pts)

Problem 1.1 (3 pts) In the self-attention layer of Transformer models, we compute three core variables—key, value and query.

True. As mentioned in lecture 10 slide 12, we need key, value, and query to calculate attention, as we take the dot product of the query and key, softmax it to get attention, and then use the values to take the weighted sum of attention for every key j for the output.

Problem 1.2 (3 pts) In the self-attention layer of Transformer models, the attention is denoted by the cosine similarity between the key and value.

False. As mentioned before, the attention is denoted by the softmax of the dot product between the query and key. Cosine similarity is the dot product of unit vectors, which is not the attention calculation.

Problem 1.3 (3 pts) In the self-attention layer of Transformer models, after obtaining the attention matrix, we need to further apply a normalization on it (e.g., layer normalization or batch normalization).

True. As mentioned in lecture slide 20, layer normalization is a key trick to help these Transformers train faster.

Problem 1.4 (3 pts) The decoder of Transformer learns auto-regressively.

True. As discussed in lecture 10, decoders such as GPT are autoregressive. Basically, decoders use information from the previous time steps of the decoder to generate the desired output at a given time step. Meanwhile, autoencoders are seeking to learn a representation of the data, whereas the autoregressor may not need to understand in order to predict (or may not be able to complete tasks that require understanding).

Problem 1.5 (3 pts) BERT's architecture is a Transformer encoder while GPT's architecture is a Transformer decoder

True. This is in lecture 10, slide 26. BERT is a pre-trained encoder that learns the representation of the data and GPT is a pre-trained decoder that is used for generating current time step words/language/other tasks based on previous data.

Problem 1.6 (3 pts) BERT's pre-training objectives include (a) masked token prediction (masked language modeling) and (b) sentence order prediction.

True. As discussed in lecture 10 slide 27, BERT's pre-training objective are masked language modeling (predict masked words) and next sentence prediction (which is the same as sentence order prediction- the idea of predicting whether two sentences are consecutive)

Problem 1.7 (3 pts) GPT is a zero-shot learner (can be transferred to unseen domain and tasks) while BERT is not.

True. GPT has been proven successful in many few shot learning and zero shot learning applications (https://arxiv.org/abs/2005.14165). Due to the nature of BERT, it is simply not able to perform well on such tasks (people are looking into ways but have not found much success).

Problem 1.8 (3 pts) Gradient clipping can be used to alleviate gradient vanishing problem

False. This is because gradient clipping is meant to alleviate the exploding gradient problem by setting a maximum possible gradient to backpropagate. Setting this maximum absolute value gradient will not affect a gradient that is converging to 0.

Problem 1.9 (3 pts) Word embeddings can only contain positive values.

False. This can be seen in lecture 9. A word embedding is merely a representation of a word. While one can do an embedding that is only positive, Word2Vec, the most ubiquitous word embedder, includes negative values. These negative values have no particular meaning, but rather the meaning of the embedding comes from its relativity/proximity to other word vectors.

Problem 1.10 (3 pts) The memory cell of an LSTM is computed by a weighted average of previous memory state and current memory state where the sum of weights is 1.

False. Based on the slides for lecture 9, the memory cell is the Hadamard product of the forget gate and previous memory state plus the Hadamard product of the input gate and the tanh activation of learned weights multiplied by the input and previous hidden state plus a learned bias.

2 Lab 1: Recurrent Neural Network for Sentiment Analysis (45 pts)

Lab 1 (45 points)

(a) (5 pts) Implement your own data loader function. First, read the data from the dataset file on the local disk. Then split the dataset into three sets: train, validation, and test by 7 : 1 : 2 ratio. Finally return $x\_train, x\_valid, x\_test, y\_train, y\_valid$ and y_test, where x represents reviews and y represent labels.

(b) (5 pts) Implement your own data loader function. First, read the data from the dataset file on the local disk. Then split the dataset into three sets: train, validation, and test by 7 : 1 : 2 ratio. Finally return $x\_train, x\_valid, x\_test, y\_train, y\_valid$ and y_test, where x represents reviews and y represent labels.

(c) (5 pts) Implement the $build\_vocab$ function to build a vocabulary based on the training corpus. You should first compute the frequency of all the words in the training corpus. Remove the words that are in the $STOP\_WORDS$. Then filter the words by their frequency ($\geq min\_freq$) and finally generate a corpus variable that contains a list of words.

(d) (5 pts) Implement the tokenization function. For each word, find its index in the vocabulary. Return a list of integers that represents the indices of words in the example.

(e) (5 pts) Implement the $\_\_getitem\_\_$ function in the IMDB class. Given an index i, you should return the i-th review and label. The review is originally a string. Please tokenize it into a sequence of token indices. Use the max_length parameter to truncate the sequence so that it contains at most max_length tokens. Convert the label string ('positive' / 'negative') to a binary index, such as 'positive' is 1 and 'negative' is 0. Return a dictionary containing three keys: 'ids', 'length', 'label' which represent the list of token ids, the length of the sequence, the binary label.

(f) (10 pts) Implement the LSTM model for sentiment analysis.

- (5 pts) In $\_\_init\_\_$, a LSTM model contains an embedding layer, an lstm cell, a linear layer, and a dropout layer. You can call functions from Pytorch's nn library. For example, nn.Embedding, nn.LSTM, nn.Linear.
- (5 pts) In forward, decide where to apply dropout. The sequences in the batch have different lengths. Write/call a function to pad the sequences into the same length. Apply a fullyconnected (fc) layer to the output of the LSTM layer. Return the output features which is of size [batch size, output dim].
- (5 pts) As a sanity check, train the LSTM model for 5 epochs with SGD optimizer. Do you observe a steady and consistent decrease of the loss value as training progresses? Report your observation on the learning dynamics of training loss, and validation loss on the IMDB dataset. Do they meet your expectations and why? (Hint: trust what you have observed and report as is.)

  No, I do not observe a steady and consistent decrease of the loss value as training progresses. After one epoch of training, the training accuracy is 0.496 (training loss: 0.693) and validation accuracy is 0.496 (validation loss: 0.694). After all 5 epochs of training, the training accuracy is 0.496 (training loss: 0.694) and validation accuracy is 0.496 (validation loss: 0.694). It is struggling to even reduce the train loss function. This meets my expectation, as SGD has a very slow convergence rate, and given the high dimensionality of data and high number of parameters, I would expect it to take much longer to learn. Additionally, this is using vanilla SGD, without momentum, which is very susceptible to becoming stuck in a local maxima as it trains (which is why it has a high learning rate). Additionally, as mentioned in Lecture 9, LSTM needs a larger dataset to train and LSTM takes a longer time to train and usually has a very large model size. The default model size being used is fairly small, there is a decent amount of data, but only a few epoch are being run and with a slower optimizer (SGD). While the vanishing gradient problem can be tough for RNNs, the LSTM is made to combat it such that this is likely not the issue.

(g) (10 pts) Gated Recurrent Unit (GRU) is a simplified version of LSTM that performs good on language tasks. Implement the GRU model similar instructions as (e), as follows:

- (5 pts) In $\_\_init\_\_$, a GRU model includes an embedding layer, an gated recurrent unit, a linear layer, and a dropout layer. You can call functions from torch.nn library. For example, nn.Embedding, nn.GRU, nn.Linear.
- (5 pts) In forward, decide where to apply dropout. The sequences in the batch have different lengths. Write/call a function to pad the sequences into the same length. Apply a FullyConnected (FC) layer to the output of the LSTM layer. Return the output features which is of size [batch size, output dim].

Lab 2

1. (Bonus, 5 pts) Repeat (f) for GRU model and report your observations on the training curve. What do you hypothesize is the issue that results in the current training curve? (Hint: Check slides of Lecture 9 for the recommendations on training RNNs.)

   I do not observe a steady and consistent decrease of the loss value as training progresses when using the GRU, as well. After one epoch of training, the training accuracy is 0.498 (training loss: 0.694) and validation accuracy is 0.496 (validation loss: 0.694). After all 5 epochs of training, the training accuracy is 0.501 (training loss: 0.694) and validation accuracy is 0.496 (validation loss: 0.695). It is struggling to even reduce the train loss function, and the validation loss marginally increased (negligible). This is the same exact performance as the LSTM. As mentioned in Lecture 9, LSTM needs a larger dataset to train and LSTM takes a longer time to train and usually has a very large model size. The default model size being used is fairly small, there is a decent amount of data, but only a few epoch are being run and with a slower optimizer (SGD). These reasons are likely similar to why the GRU is failing to improve on its training curve as well.

3 Lab 2: Training and Improving Recurrent Neural Network(25 pts)

Lab 3

(a) (5 pts) We start to look at the optimizers in training RNN models. As SGD might not be good enough, we switch to advanced optimizers (i.e., Adagrad, RMSprop, and Adam) with adaptive learning rate schedule. We start with exploring these optimizers on training a LSTM. You are asked to employ each of the optimizer on LSTM and report your observations. Which optimizer gives the best training results, and why?

Adam works best as expected. This is because Adam combines the second moment information used by AdaGrad and RMS Prop (also uses decay rate) with first moment information. Adam is the best of both worlds scenario. It combines the advantages of SGD with momentum and RMSProp Optimizers (and RMSProp was an improvement of AdaGrad). It is able to leverage its extra information to converge fast and create the best learning rate at a given epoch, helping it reach a strong accuracy in 5 epochs here. RMS Prop is generally able to perform similarly well.

Model	OPTIM	LR	Test Accuracy
LSTM	Adam	0.001	0.868
LSTM	AdaGrad	0.001	0.808
LSTM	RMSProp	0.001	0.857
LSTM	SGD (default)	0.01	0.496

(b) (5 pts) Repeat (a) on training your GRU model, and provide an analysis on the performance of different optimizers on a GRU model. How does a GRU compare with a LSTM, in terms of model performance and model efficiency? Up to this stage, you should expect at least 86% accuracy on IMDB dataset. You may find that optimizers solve most of the existing dilemmas in RNN training. Next, we will start playing with the structures of RNN models and see if we can design a better RNN model. Starting here, you may change any of the hyperparameters except random seed. Yet, our recommendation of hyperparameter change is limited to learning rate, number of layers in RNN models, and the dimension of embedding/hidden units.

(c) The GRU has approximately 90 percent of the parameters of the LSTM and is able to perform similarly to the LSTM, and even better for the AdaGrad optimizer, helping overcome some of its shortcomings. It is more efficient and has a shorter runtime, while maintaining a similar level of performance. Additionally, the GRU appeared to be somewhat less dependent on the optimizer, as Adam, AdaGrad, and RMS Prop all performed incredibly similarly (while SGD did not, it is in a difference class of optimizer that does not adapt). It was able to significantly improve the AdaGrad model.

(d)

Model	OPTIM	LR	Test Accuracy
GRU	Adam	0.001	0.858
GRU	AdaGrad	0.001	0.854
GRU	RMSProp	0.001	0.864
GRU	SGD (default)	0.01	0.498

(e) (5 pts) Try to make your RNN model deeper by changing the number of layers. Is your RNN model achieving a better accuracy on IDMB classification? You may use LSTM as an example. (Hint: you do not need to explore more than 4 recurrent layers in a RNN model.).

My RNN is not seeing much improvement from making it deeper. This is likely due to the lack of DropOut as a regularizer in the model. I was able to see a slight performance gain in the appendix when using DropOut that improving the depth of the network could provide marginal performance gain (the dropout regularization likely helped accomodate the increase in parameters).

Model	OPTIM	LR	N_LAYERS	Drop Out	Test Accuracy
LSTM	Adam	0.001	1 (default)	N	0.868
LSTM	Adam	0.001	2	N	0.852
LSTM	Adam	0.001	3	N	0.852
LSTM	Adam	0.001	4	N	0.864

Lab 4

(a) (5 pts) Try to make your RNN model wider by changing the number of hidden units. Is your RNN model achieving a better accuracy on IDMB classification? You may use LSTM as an example. (Hint: you do not need to explore a hidden dimension of more than 320 on IMDB).

Model	OPTIM	LR	N_LAYERS	$HIDDEN\_DIM$	Test Accuracy
LSTM	Adam	0.001	1	25	0.872
LSTM	Adam	0.001	1	50	0.819
LSTM	Adam	0.001	1	75	0.863
LSTM	Adam	0.001	1	100 (default)	0.868
LSTM	Adam	0.001	1	125	0.864
LSTM	Adam	0.001	1	150	0.872
LSTM	Adam	0.001	1	175	0.874
LSTM	Adam	0.001	1	200	0.872
LSTM	Adam	0.001	1	225	0.859
LSTM	Adam	0.001	1	250	0.870
LSTM	Adam	0.001	1	275	0.870
LSTM	Adam	0.001	1	300	0.862
LSTM	Adam	0.001	1	320	0.870

Yes, changing the number of hidden dimensions was able to increase RNN accuracy (with several different values) on this dataset. Both decreasing to 25 (may help against overfitting) and increasing to values such as 175 (may be able to encode information) accounted for a marginal performance boost.

(b) (5 pts) Embedding tables contain rich information of the input words and help build a more powerful representation with word vectors. Try to increase the dimension of embeddings. Is your RNN model achieving a better accuracy on IMDB classification? You may use LSTM as an example. (Hint: you do not need to explore an embedding dimension of larger than 256).

I am exploring this hyperparameters based on the previous optimal hyperparameter, as that is how I am interpreting this question. Ultimately, in hyperparameter selection, one can not generally try every combination of hyperparameters (although they grid search the numerical ones they can), such that some architectural/optimizer choices are made in a greedy way by the optimizing the most important variable first (e.g. learning rate/optimizer). I am doing this here, at the risk of the greedy approach providing a suboptimal hyperparameter combination. An additional exploration of the hyperparameters for this variable (using default hidden states) is in the appendix.

Model	OPTIM	LR	$LAYERS$	$HIDDEN\_DIM$	$EMBED\_DIM$	Test Accuracy
LSTM	Adam	0.001	1	175	1 (default)	0.874
LSTM	Adam	0.001	1	175	2	0.862
LSTM	Adam	0.001	1	175	4	0.863
LSTM	Adam	0.001	1	175	8	0.853
LSTM	Adam	0.001	1	175	16	0.853
LSTM	Adam	0.001	1	175	32	0.856
LSTM	Adam	0.001	1	175	64	0.834
LSTM	Adam	0.001	1	175	128	0.830
LSTM	Adam	0.001	1	175	256	0.848

No, changing the number of embedding dimensions was able to increase RNN accuracy (with several different values) on this dataset. The dataset is relatively small (may be able to embed information easier), and DropOut is not being used (not directly related but could have some kind of impact due to its regularization abilities). Additionally, we used the high value of hidden states here such that there are more parameters to learn (those extra parameters help better model the embeddings)- optimizing the embedding dimensions while optimizing the hidden dimensions would work better as they are directly related.

Lab 5

1. (Bonus, 5 pts) A better way to scale up RNN models is simultaneously scaling up the number of hidden units, number of layers, and embedding dimension. This is called compound scaling, which is widely adopted in emerging ML research. You are asked to make no more than 50 trials to perform a compound scaling on your implemented RNN models. Is the model crafted via compound scaling performing better than the models you obtain in (d), (e), and (f)? You may use LSTM as an example. (Hint: You may use the accuracy-parameter trade-off as a criterion.)

   In doing my compound scaling, it was difficult to do it in the true nature of the EfficientNetpaper, which suggests compound scaling with depth coefficient, width coefficient, and resolution coefficient such that the depth coefficient times the width coefficient squared times the resolution coefficient squared is equal to 2. The embedding dimension starts at 1 and can range greatly, such that scaling it up would require greater coefficients or more compounding (and more compounding). There is not much instruction into creating starting values (in order to scale them well in relation to each other- should you start from optimal found in early testing or lowest values that perform okay, or other values), how much to compound for (especially given we are limited to 50 total trials), and how to best go about finding the optimal coefficients. Compounding too much could lead to too many layers or dimensions for a given hyperparameter, wasting computational resources on models that don't have great hyperparameters. Ultimately, I believe this technique would have worked better with a more flexible but similarly motivated hyperparameter search that involved continously scaling from different values at different rates while controlling the growth rate and max/min parameter (a form of non-exhaustive grid search- grid values in the same tier of scale for each parameter).

   In any case, I explored with a few ranges of values for the coefficients before applying one for 10 trials and seeing how the models performed.

   Additionally, I believe the model would have done better with DropOut implemented, as that would help it regularize such that it works for networks with more parameters.

   In any case, in trying to stay true to the paper and using the compound scaling coefficients, we were able to explore the search space and find some marginally better models, but not to the extent that I would have liked (88 percent). More trials and a better idea of how to start (or more trials to explore) and more flexibility in scaling coefficients could have led to finding a better hyperparameter combination.

2. (Bonus, 5 pts) RNNs can be bidirectional. Use the best model discovered in (f) and make it bidirectional. Do you observe better accuracy on IMDB dataset and why?

   The model received a small performance boost by introducing bidirectionality to 0.88 (although with a full compound scaling would have been better). Bidirectional RNN should do better as they have more context. Bidirectional LSTMs are complex such that they sometimes do require more training (and we are limited to 5 epochs). I suspect that additional training could benefit the bidirectional LSTM, as well as a better architecture. Additionally, I do not think I have the optimal parameters, such that the bidirectional may have provided the boost needed to get in the ideal performance range had I approached compound scaling better (better starting values, scaling coefficients, more runs).

4 Appendix 1: Values Obtained for Embedding Tuning When Default Hidden States Used

These models were run in a different notebook (I can submit that too if desired).

Lab 6

Model	OPTIM	LR	$LAYERS$	$HIDDEN\_DIM$	$EMBED\_DIM$	Test Accuracy
LSTM	Adam	0.001	1	100	1 (default)	0.868
LSTM	Adam	0.001	1	100	2	0.863
LSTM	Adam	0.001	1	100	4	0.858
LSTM	Adam	0.001	1	100	8	0.865
LSTM	Adam	0.001	1	100	16	0.858
LSTM	Adam	0.001	1	100	32	0.849
LSTM	Adam	0.001	1	100	64	0.854
LSTM	Adam	0.001	1	100	128	0.857
LSTM	Adam	0.001	1	100	256	0.850

5 Appendix 2: Values Obtained When Using Drop-Out

These models were run in a different notebook (I can submit that too if desired).

Lab 7

N_LAYERS Experiment:

Model	OPTIM	LR	N_LAYERS	Drop Out	Test Accuracy
LSTM	Adam	0.001	1 (default)	Y	0.860
LSTM	Adam	0.001	2	Y	0.856
LSTM	Adam	0.001	3	Y	0.871
LSTM	Adam	0.001	4	Y	0.871

$HIDDEN\_DIM$ Experiment:

Model	OPTIM	LR	N_LAYERS	$HIDDEN\_DIM$	Test Accuracy
LSTM	Adam	0.001	3	25	0.869
LSTM	Adam	0.001	3	50	0.863
LSTM	Adam	0.001	3	75	0.864
LSTM	Adam	0.001	3	100 (default)	0.871
LSTM	Adam	0.001	3	125	0.832
LSTM	Adam	0.001	3	150	0.502
LSTM	Adam	0.001	3	175	0.827
LSTM	Adam	0.001	3	200	0.853
LSTM	Adam	0.001	3	225	0.741
LSTM	Adam	0.001	3	250	0.852
LSTM	Adam	0.001	3	275	0.858
LSTM	Adam	0.001	3	300	0.858
LSTM	Adam	0.001	3	320	0.860

$EMBED\_DIM$ Experiment:

Model	OPTIM	LR	$LAYERS$	$HIDDEN\_DIM$	$EMBED\_DIM$	Test Accuracy
LSTM	Adam	0.001	3	100	1 (default)	0.871
LSTM	Adam	0.001	3	100	2	0.857
LSTM	Adam	0.001	3	100	4	0.505
LSTM	Adam	0.001	3	100	8	0.857
LSTM	Adam	0.001	3	100	16	0.858
LSTM	Adam	0.001	3	100	32	0.847
LSTM	Adam	0.001	3	100	64	0.849
LSTM	Adam	0.001	3	100	128	0.855
LSTM	Adam	0.001	3	100	256	0.855