Your Name: João Miguel Coelho

Your Andrew: jmcoelho

Homework 3

1. Training Set Construction (5 pts)

Construct the training set for the amazon review dataset as instructed and report the following statistics.

Statistics	
[1,2]the total number of unique words in T	24473
the total number of training examples in T	2000
the ratio of positive examples to negative examples in T	1
[1]the average length of document in T	187.67
[1]the max length of document in T	3816

[1]The reported values are in terms of tokens as given by a split on white-space, which can be seen as very basic word-level tokenizer. Nothing was removed from this tokenization (i.e., punctuation and special chars were kept). The values above may be different for other tokenization strategies.
[2]Special tokens used by the model (4) were not accounted for in the total count.

2. Performance of deep neural network for classification (20 pts)

Suggested hyperparameters:

1. Data processing:
    a. Word embedding dimension: 100
    b. Word Index: keep the most frequent 10k words
2. CNN

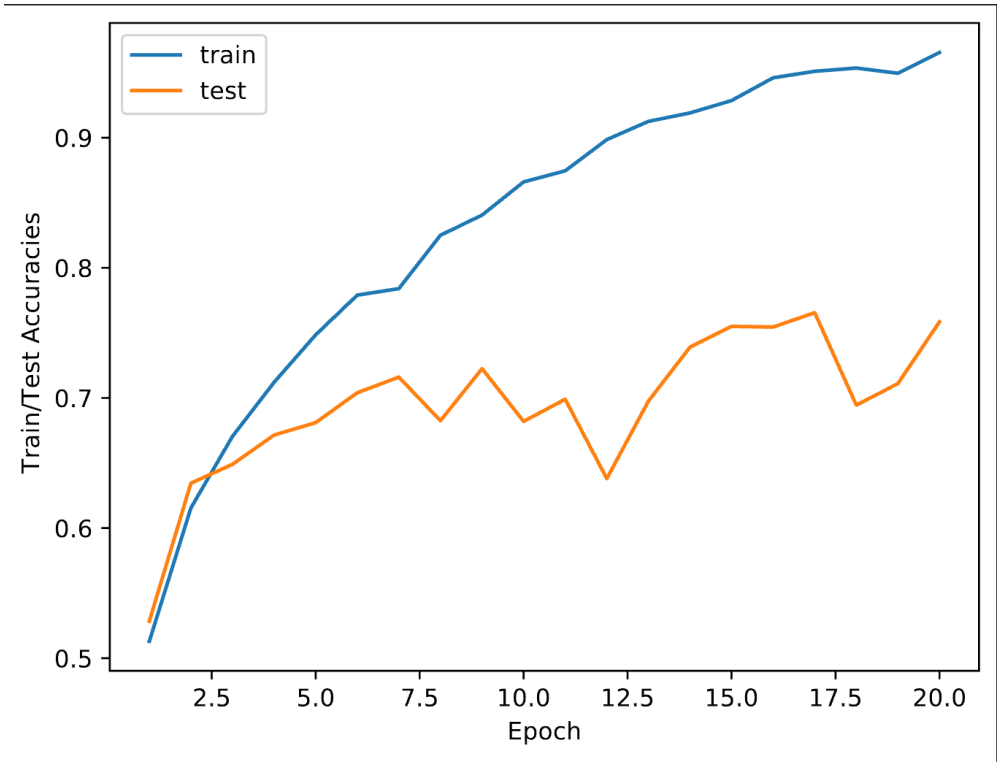
   a. Network: Word embedding lookup layer -> 1D CNN layer -> fully connected layer -> output prediction
   b. Number of filters: 100
   c. Filter length: 3
   d. CNN Activation: Relu
   e. Fully connected layer dimension 100, activation: None (i.e. this layer is linear)
3. RNN:
   a. Network: Word embedding lookup layer -> LSTM layer -> fully connected layer(on the hidden state of the last LSTM cell) -> output prediction
   b. Hidden dimension for LSTM cell: 100
   c. Activation for LSTM cell: tanh
   d. Fully connected layer dimension 100, activation: None (i.e. this layer is linear)

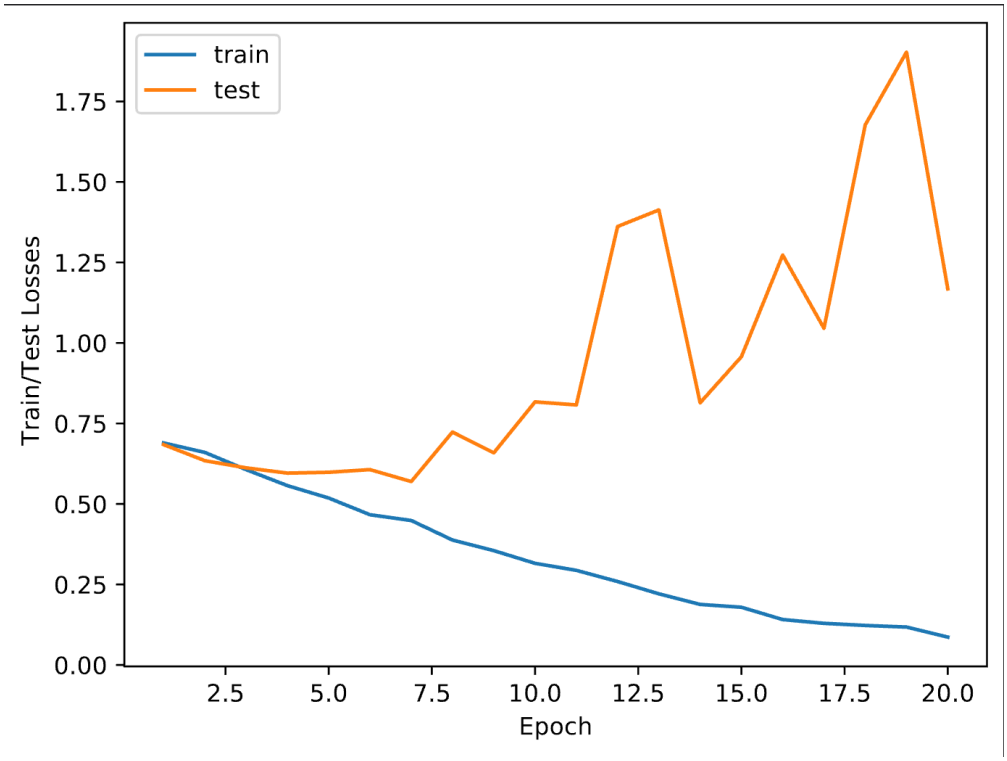
	Accuracy	Training time(in seconds)
RNN w/o pretrained embedding	0.7655	340 (17s/epoch)
RNN w/ pretrained embedding	0.8280	300 (15s/epoch)
CNN w/o pretrained embedding	0.6409	100 (5s/epoch)
CNN w/ pretrained embedding	0.7210	40 (2s/epoch)

3. Training behavior (10 pts)

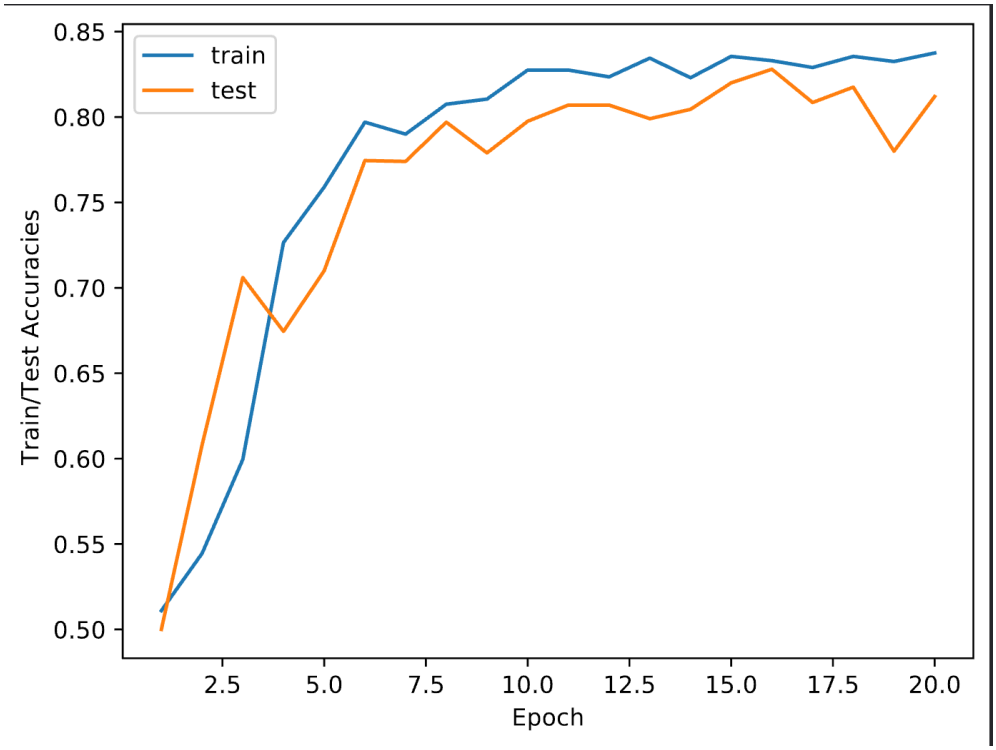
Plot the training/testing objective, training/testing accuracy over time for the 4 model combinations (correspond to 4 rows in the above table). In other word, there should be 2*4=8 graphs in total, each of which contains two curves (training and testing).

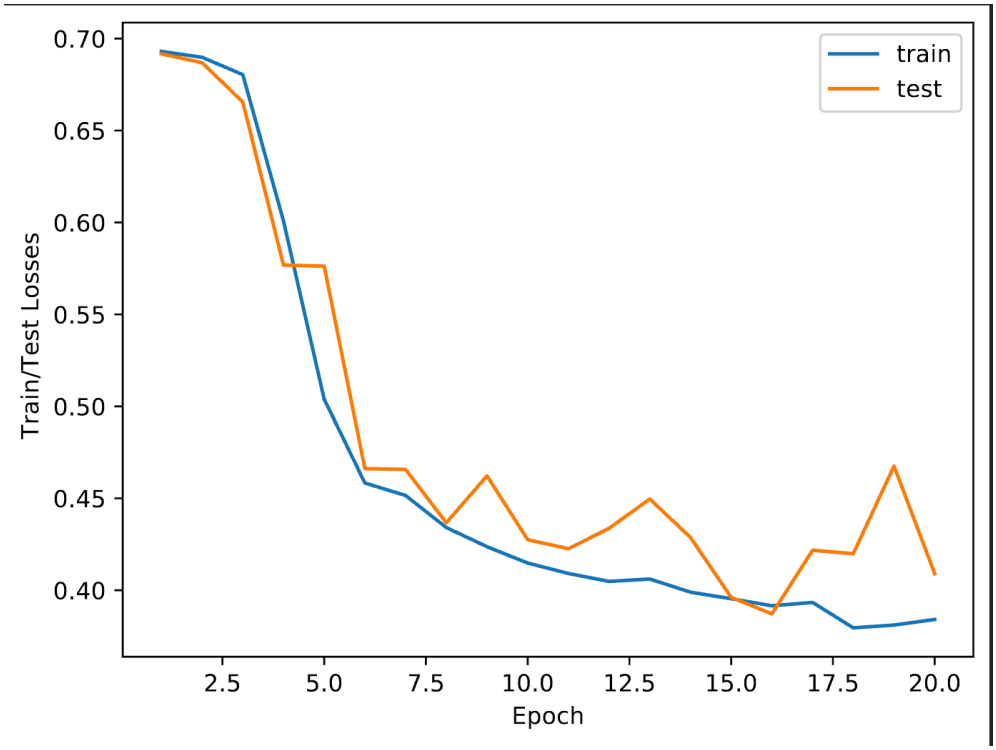
RNN w/o pretrained embedding



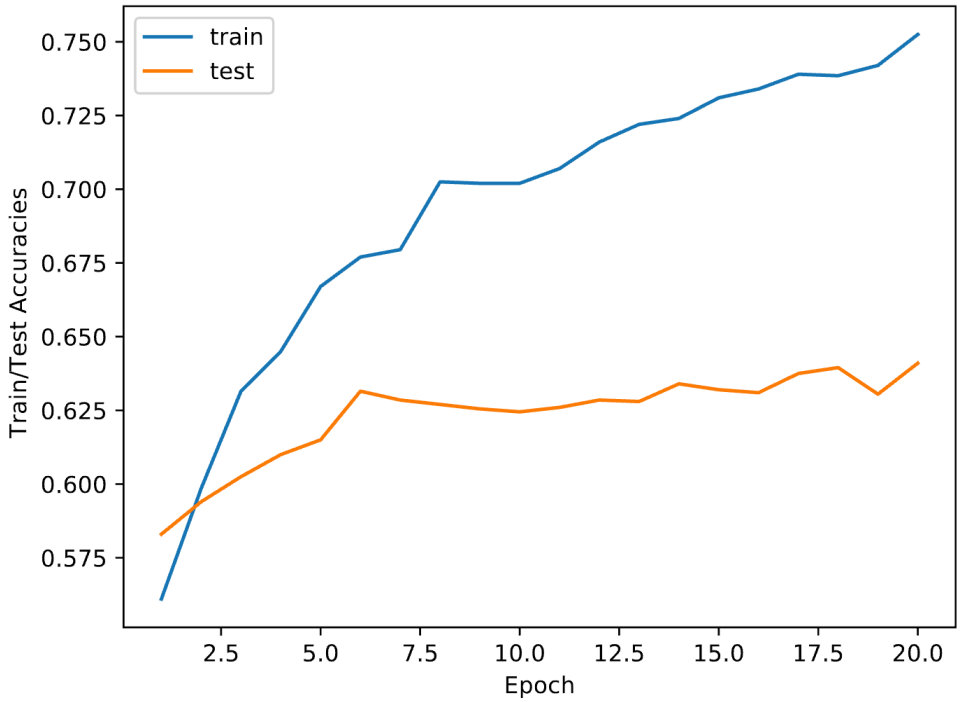


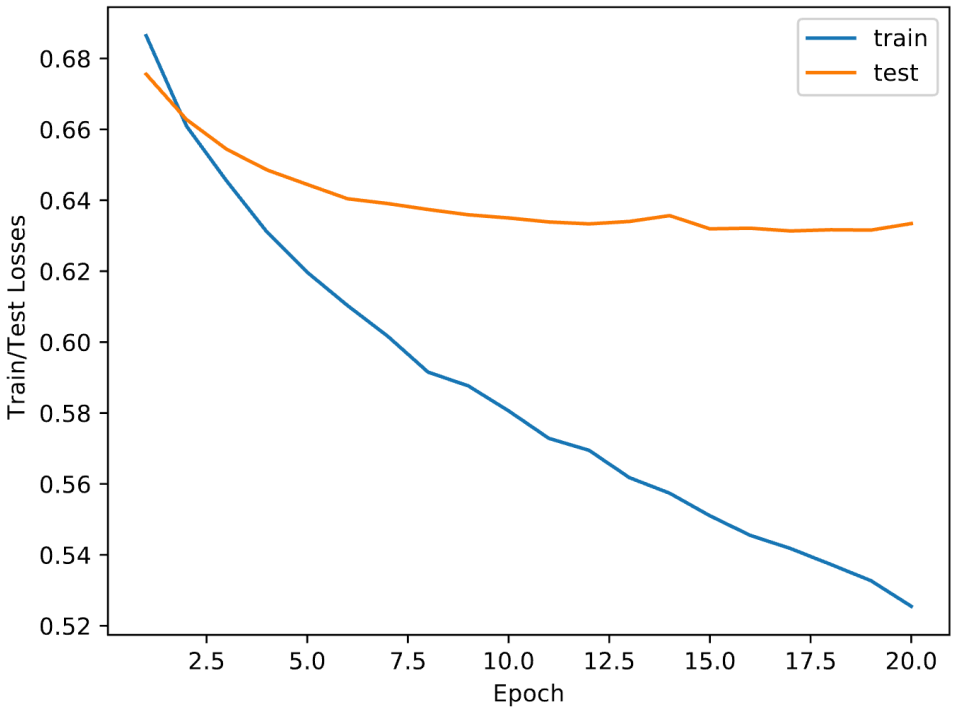
RNN w/ pretrained embedding



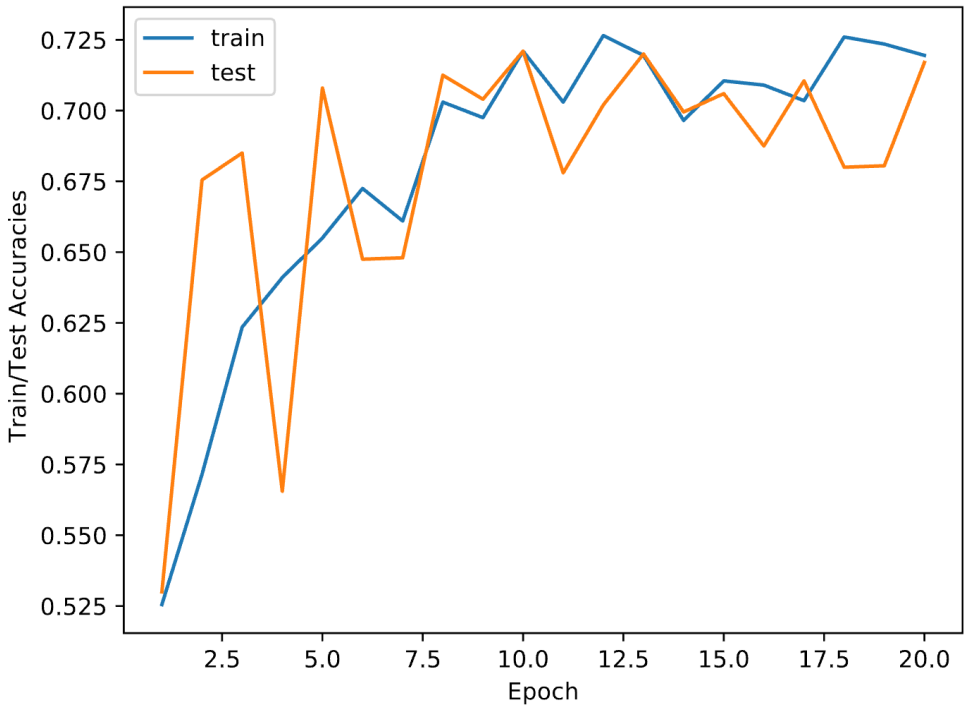


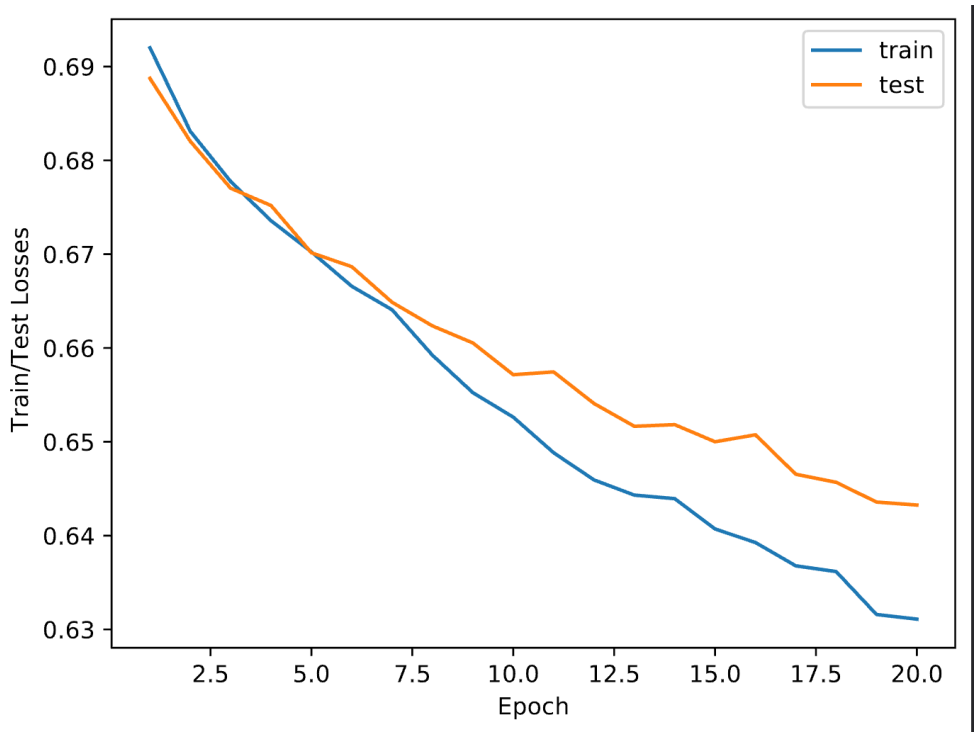
CNN w/o pretrained embedding





CNN w/ pretrained embedding





4. Analysis of results (10 pts)

Discuss the complete set of experimental results, comparing the algorithms to each other. Discuss your observations about the various algorithms, i.e., differences in how they performed, different parameters, what worked well and didn't, patterns/trends you observed across the set of experiments, etc. Try to explain why certain algorithms or approaches behaved the way they did.

RNN models setup:

The architecture for the RNN model is as follows:

- Embedding layer: Maps each token to a vector of size (hidden_dimension).
- Bi-directional LSTM encoder: the representations from the last hidden layer of forward and backward LSTM are concatenated and fed to a linear layer which reduces the dimensionality from (2*hidden_dimension) to (hidden_dimension).
- Linear layer: Maps inputs of shape (hidden_dimension) to the number of classes. No activation is used, as the pytorch nn.CrossEntropyLoss already computes softmax, i.e., the criterion handles raw logits.

By design choice, the number of classes was set to 2. It would also be possible to set it to 1, and use pytorch's binary cross entropy loss function. The code supports both approaches through a flag in the configuration file.

The hyperparameters for both experiments with RNN were slightly different to account for the change between pre-trained vs non-pre-trained embedding layer:

	W/O PRE-TRAINED EMBEDDINGS	**W PRE-TRAINED EMBEDDINGS**
EPOCHS	20	20
BATCH SIZE	20	20
HIDDEN SIZE	100	100
LEARNING RATE (CONST)	0.0002	0.00008
L2 PENALTY	1e-2	4e-3
DROPOUT	0.2	0.15
MAX LENGTH	192	192
OPTIMIZER	AdamW	AdamW

Other setups with higher learning rate and less regularization would start-off much stronger (i.e., 75%> test accuracy on the first epoch), but the remainder of the training would be much less stable, with patterns of increasing validation loss and accuracy at the same time, which after empirical observation I attribute to the model becoming overconfident on its misclassifications. This pattern can be observed for instance in the loss and accuracy curves for the RNN without pre-trained embeddings.

It is also worth noting that the LSTM encoder is bi-directional, as it achieved much better results than the unidirectional LSTM. This is expected, since text is now processed left-to-right and right-to-left, i.e., tokens are contextualized with both past and future terms.

Tokenization is done with torchtext's "basic_english" tokenizer, which is a word-level tokenizer with some extra rules, for instance punctuation removal, verb tense normalization, etc. Maximum sequence length was chosen as the average number of tokens in training examples as given by this tokenization. A word was considered to be part of the vocabulary if it appeared at least once in the training data.

CNN models setup:

The model implementation follows the given paper:

- Filters are of size (x, embedding dimension), hence moving vertically through the feature maps with a stride of 1 and no padding.
- ReLU is used as the activation function.
- Max pooling is applied to each filter output, generating a single scalar per filter.
- The scalars are combined into a vector of shape (1, num filters) and fed to a linear layer.

Differences from the original paper include:

- All filters use the same size.
- A single channel with either learnable or pre-trained frozen embeddings is used.

Again, by design choice, the number of classes was set to 2. It would also be possible to set it to 1, and use pytorch's binary cross entropy loss function. The code supports both approaches through a flag in the configuration file.

The hyperparameters for both experiments with CNN were slightly different to account for the change between pre-trained vs non-pre-trained embedding layer:

	W/O PRE-TRAINED EMBEDDINGS	W PRE-TRAINED EMBEDDINGS
EPOCHS	20	20
BATCH SIZE	20	20
HIDDEN SIZE	100	100
LEARNING RATE (CONST)	0.0001	0.00005
L2 PENALTY	1e-1	1e-2
DROPOUT	0.25	0.1
MAX LENGTH	192	192
OPTIMIZER	Adam	Adam
NUM FILTERS	200	200
FILTER SIZE	(3,100)	(3, 100)
CHANNELS	1	1
STRIDE	1	1

During the experiments, I noticed that CNN benefited from heavier regularization, even though it's a less complex model, when compared to the previous LSTM. This may be due to the fact that the LSTM is a model better equipped to deal with the sequential nature of this textual task. Hence, CNNs, which are better equipped for spatial features, may have limited capacity for the task in hand, potentially being prone to overfitting.

Tokenization, maximum sequence length, and vocabulary construction follow what was used on the LSTM.

RNN vs CNN results:

Some insights on each model are already detailed above. Overall, the LSTM achieved the best results. I attribute this to the higher number of parameters, the bi-directional encoder on the LSTM, and the LSTMs ability of capturing sequential patterns. More in-depth testing on the CNN can be conducted to increase its complexity: multiple filter sizes, more filters, etc. Still this needs to be done carefully, since for the scenario without pre-trained embeddings, the model is already showing signs of overfitting. In terms of run-time, CNN is faster, given the lower amount of parameters. Since the statement did not ask for optimal performance, I did not do a comprehensive hyperparameter tuning. Having said that, I spent more time tuning the LSTM, which may have biased the results.

One problem that is usually tied with RNN-like models is gradient vanishing. LSTM aims to solve that through gating operations which limit the amount of chained non-linear computations. I did not experience gradient vanishing in the LSTM experiments. Moreover, I add gradient clipping during the training of both CNN and LSTM to avoid gradient explosion.

Pre-trained vs Non pre-trained embedding layer:

When using the aforementioned tokenization, some of the tokens in the vocabulary did not have a corresponding entry in the pre-trained embedding file. In this situation, a random initialization was considered.

When training with the pre-trained embeddings, the embedding layer was frozen. Hence, there are two benefits from the start: less parameters to train, and meaningful initial representations. As such, in both RNN and CNN, using pre-trained embeddings made the models converge much faster to optimal results, and are faster to train due to having less trainable parameters. In the end, both models achieved better results by starting with pre-trained embeddings. Also, I note that it was harder to achieve stable training with the non-pretrained embeddings. This difficulty most likely arises from the increased complexity and harder task, since the embeddings need to be trained from scratch. Also, the limited training data available may not be sufficient to produce meaningful embeddings.

As previously stated, I considered a word to be part of the vocabulary if the word appears at least once in the training data. This may make learnable embeddings even harder in this scenario, since the embedding lookup table will have a very high number of parameters (n_vocab * hidden_dim). Using only the top-K words may be a better strategy in this case., but it was not tested.

5. The software implementation (5 pts)

Add detailed descriptions about software implementation & data preprocessing, including:

1. A description of what you did to preprocess the dataset to make your implementations easier or more efficient.

I left the data file structure as is. I parsed this data into a pytorch dataset which can be fed to a dataloader. Steps include:

- Compute a vocabulary: tokenize the examples, find unique tokens, and add special tokens. All tokens that appeared at least once in the train set were used to build the vocabulary. Four special tokens were used: [BOS] – marks the beginning of a sentence; [EOS] – marks the end of a sentence; [PAD] – pad token to make all examples in a single batch of the same size; [UNK] – As only the train data was used to build the vocab, some tokens in the test data to be new, hence the need for this token.
- Define a maximum sequence length: The average document token length in the training set (192) was used as maximum sequence length. Smaller examples are padded, and longer examples truncated.
- Label the examples: Each example is associated with a label, either 0 for negative or 1 for positive.

2. A description of major data structures (if any); any programming tools or libraries that you used;

Code is fully built on top of pytorch for optimization and data handling. Torchtext was also used, but only for the word-level “basic_english” tokenizer. Matplotlib was used for the plots.

3. Strengths and weaknesses of your design, and any problems that your system encountered;

The trained models achieve decent performance on the task, and timing is also OK. Training curves are automatically saved as pdfs after each run. The implementation is clean and has the following components:

- Custom_datasets:
  - Exposes a pytorch dataset which can be fed to a data loader.
- Custom_models:
  - Exposes implementations for the CNN and RNN models. Both implementations are consistent, i.e., the receive the same input format and return the same output format.
  - Also exposes a trainer interface, which, since the models are consistent, works for both models. It implements a training and evaluation steps. It computes accuracy and loss for both train and evaluation. Batching is supported, and training employs gradient clipping to avoid gradient explosions.
- Configuration files can be used to parse parameters for easier training. The submitted code contains the 4 examples here reported.

Possible future improvements:

- No validation: using only a train and a test set impairs hypothesizing about the generalizability of the model. I.e., hyperparameters were tuned directly on the test data, which is not standard. Usually, I would have considered sampling some examples from the test set to use as validation, however, since question 3 asks for training and testing over time, I limited the experiments to that.
- Constant learning rate: With deep learning models, a constant learning rate may hinder generalization. E.g., a decaying learning rate (exponential, linear, co-sine annealing, …) would probably be more appropriate, allowing the model to explore the complex loss landscape of the non-convex objective in the initial steps, and restricting this behavior by smoothly decreasing the learning rate in later steps.
- Global padding: all examples are padded to the maximum sequence length. Batch padding could be implemented to save some computation, i.e., pad to the maximum size in the batch which can be lower than the maximum sequence length.
- Word-level tokenization: Word level tokenization was used since the provided pre-trained embeddings are also on word level. However sub-word tokenization techniques have shown better performance in the literature for this type of task.
- Non-exhaustive hyperparameter tuning: The statement did not ask for optimal performance, so the hyperparameter tuning was not very comprehensive.
- No GPU support: The code provided runs on CPU. Adding GPU support will make it faster.