AI539 - Natural Language Processing with Deep Learning

Homework 2 Report

Recurrent Neural Networks and Sentence Representations

Author: Vy Bui
OSUID: 934370552

Instructor: Professor Stefan Lee

The School of Electrical Engineering and Computer Science
Oregon State University

Task 1.1

Manually find weights and biases for the univariate LSTM defined above such that the final hidden state will be greater than or equal to 0.5 for odd parity strings and less than 0.5 for even parity. The parameters you must provide values for are w_{ix}, w_{ih}, b_i, w_{fx}, w_{fh}, b_f, w_{ox}, w_{oh}, b_o, w_{gx}, w_{gh}, b_g and are all scalars. The LSTM will take one bit of the string (0 or 1) as input x at each time step. A tester is set up in `univariate_tester.py` where you can enter your weights and check performance.

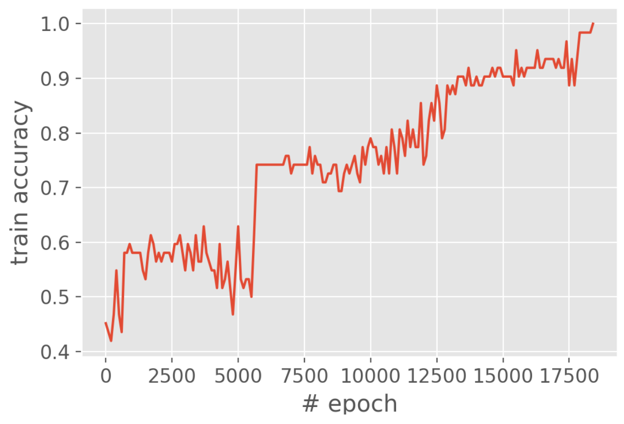
Hints: Some of the weights will likely be zero. Others may be large. Scale matters, larger weights will saturate the activation functions. Also note that `A XOR B` *can be written as* `(A OR B) AND (A NAND B)`*. Work backwards and try to find where this sort of two-term structure could be implemented.*

Task 2.1
Implement the `ParityLSTM` class in `driver_parity.py`. Your model's `forward` function should process the batch of binary input strings and output a $B \times 2$ tensor y where $y_{b,0}$ is the score for the b^{th} element of the batch having an even parity and $y_{b,1}$ for odd parity. You may use any PyTorch-defined LSTM functions. Larger hidden state sizes will make for easier training in my experiments. Running `driver_parity.py` will train your model and output per-epoch training loss and accuracy. A correctly-implemented model should approach 100% accuracy on the training set. In your write-up for this question, describe any architectural choices you made.

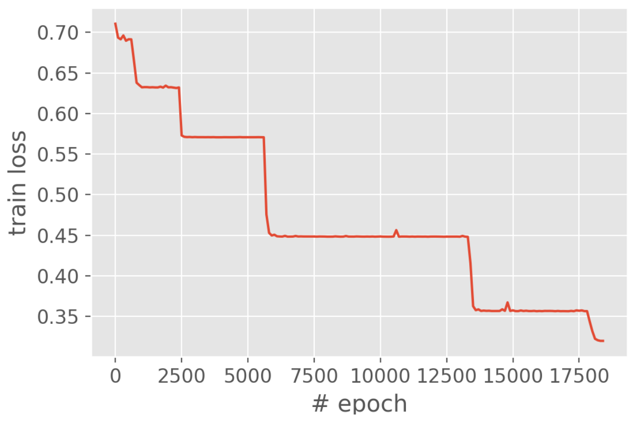
Hint: Efficiently processing batches with variable input lengths requires some bookkeeping or else the LSTM will continue to process the padding for shorter sequences along with the content of longer ones. See pack_padded_sequence and pad_packed_sequence documentation in PyTorch.

The model takes one input feature at a time, including one layer of LSTM with 64 hidden units. The final states of these units are fed to a 2 back-to-back Linear layers with the shape of 128×128 and 128×2, respectively. Their outputs then go to a sigmoid function to generate probabilities for even and odd parity.

Various learning rates were examined and the best learning rate appeared to be 0.0005. The model's training process is depicted by the Figure 1. It took around 18000 epochs to drive the train accuracy up to 1.



(a) train accuracy



(b) train loss

Figure 1: LSTM-128 training with the learning rate of 0.0005

Task 2.2
`driver_parity.py` also evaluates your trained model on binary sequences of length 0 to 20 and saves a corresponding plot of accuracy vs. length. Include this plot in your write-up and describe the trend you observe. Why might the model behave this way?

The model validation accuracy is shown in Figure 2. The model seemed to perform very well with strings of length less than or equal to 5. However, its performance started to degrade significantly with longer sequences until reaching the same accuracy as random guess with sequences longer than 13. The perfect accuracies for short sequences and the fact that we only trained the model with such sequences might imply over-fitting. One possible reason might be that the training data did not represent the population. Another reason might be the high complexity of the model.

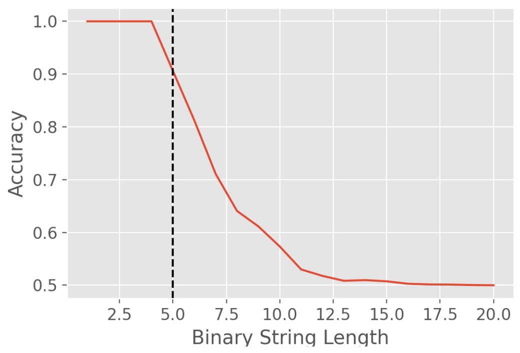
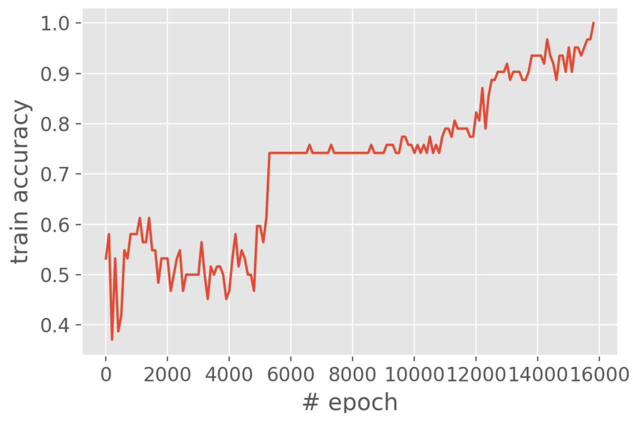


Figure 2: LSTM-128 parity generalization

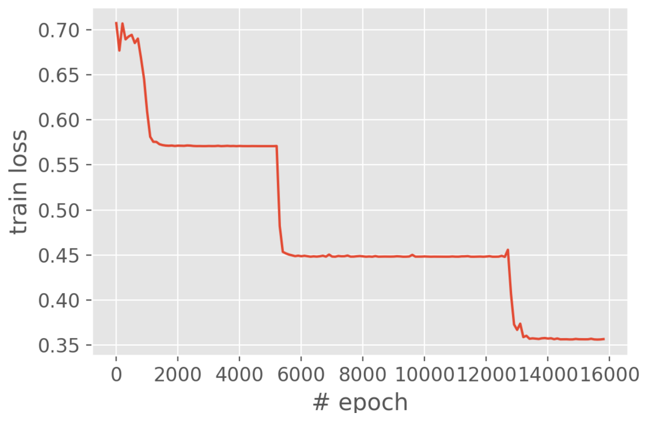
Task 2.3
We know from 1.1 that even a univariate LSTM (one with a scalar hidden state) can theoretically solve this problem. Run a few (3-4) experiments with different hidden state sizes, what is the smallest size for which you can still train to fit this dataset? Feel free to adjust any of the hyper-parameters in the optimization in the `train_model` function if you want. Describe any trends you saw in training or the generalization experiment as you reduced the model capacity.

Models with hidden dimension of 64, 16, 4, and 2 were experimented with training for as many as 80000 epochs at the learning rate of 0.0005, as shown in the figure 3, 4, 5, and 6. Unlike theory suggests, the smallest size LSTM that can fit the data set seemed to be 4.

The model with 2 hidden units seemed not to improve much after reaching about 0.98 training accuracy. However, figure 6 shows that it took about 60000 epochs before a surge in accuracy occurs, which implies that the model might be able to escape the plateau if it is given more training time. Another 2-hidden-unit model was trained with 200000 epochs. As demonstrated by figure 7, the performance was even worse than the one trained with 80000 epochs, possibly due to random initialization.

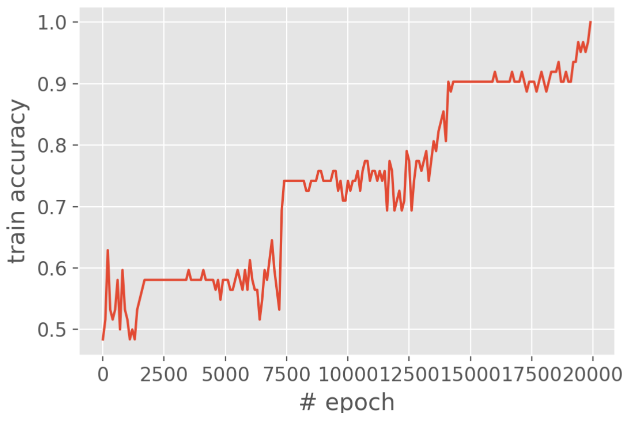


(a) train accuracy

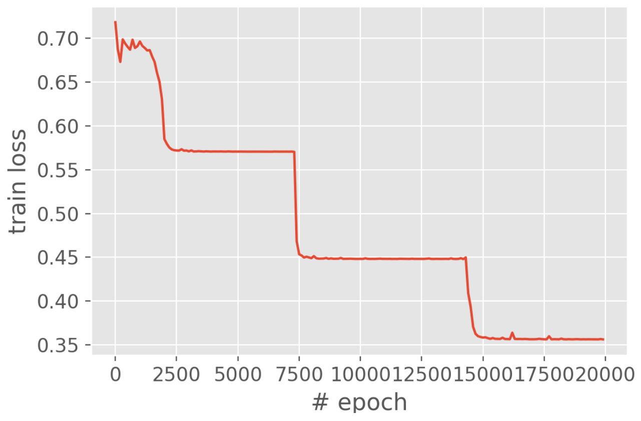


(b) train loss

Figure 3: LSTM-64 training with the learning rate of 0.0005

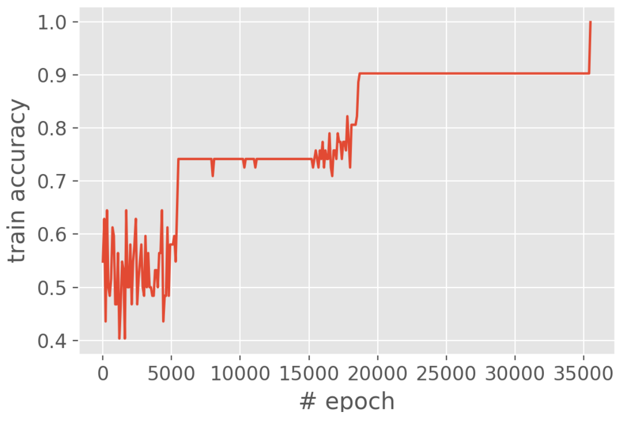


(a) train accuracy

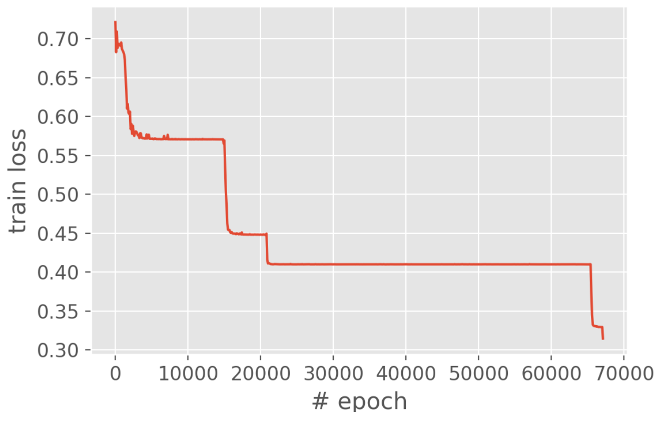


(b) train loss

Figure 4: LSTM-16 training with the learning rate of 0.0005

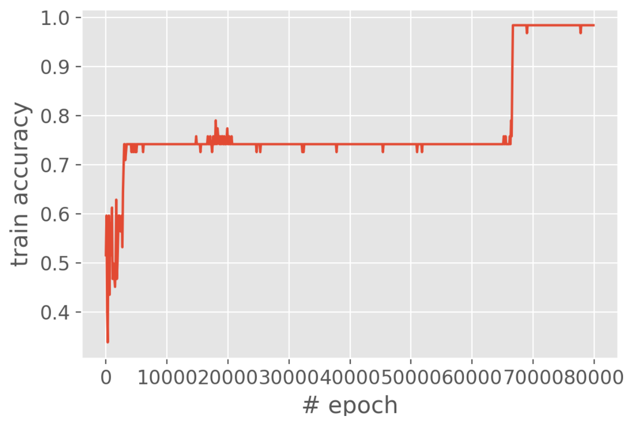


(a) train accuracy

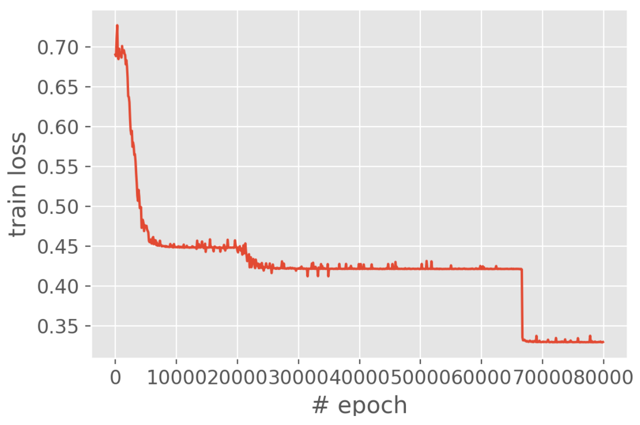


(b) train loss

Figure 5: LSTM-4 training with the learning rate of 0.0005

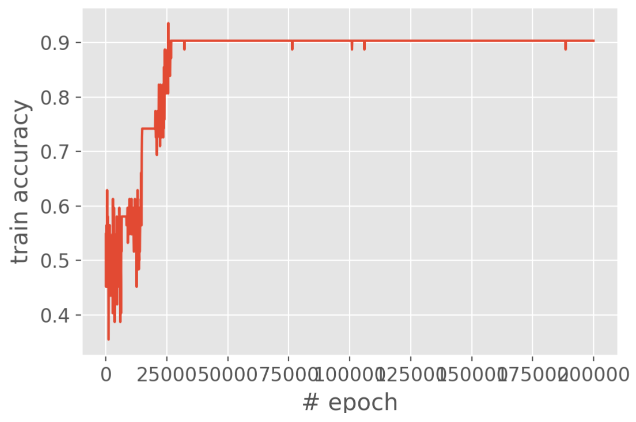


(a) train accuracy

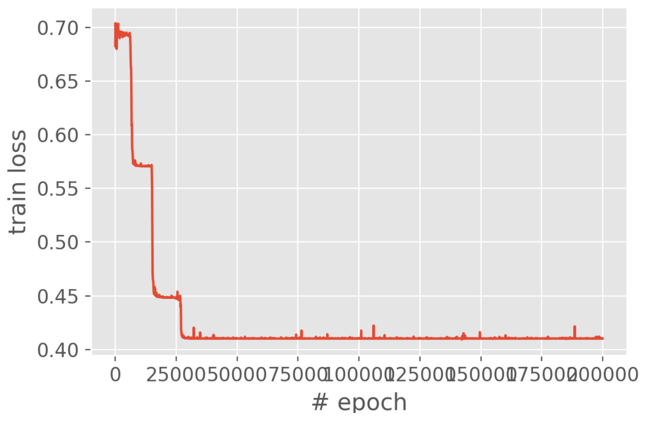


(b) train loss

Figure 6: LSTM-2 training with 80000 epochs



(a) train accuracy



(b) train loss

Figure 7: LSTM-2 training with 200000 epochs

Task 2.4
It has been demonstrated that vanilla RNNs have a hard time learning to classify whether a string was generated by an ERG or not. LSTMs on the other hand seem to work fine. Based on the structure of the problem and what you know about recurrent networks, why might this be the case?

The advantage of LSTM or other networks based on the gated recurrent unit is the ability to forget old state that was already ”learned” by the model. This makes LSTM insusceptible to vanishing or exploding gradients. ERG strings with k = 4 have the lengths of at least 24, which are long enough to cause vanishing or exploding gradients. This might be the reason why RNN could not learn much, whereas LSTM performed quite well in this task.

Task 3.1
The first step for any machine learning problem is to get familiar with the dataset. Read through random samples of the dataset and summarize what topics it seems to cover. Also look at the relationship between words and part-of-speech tags – what text preprocessing would be appropriate or inappropriate for this dataset? Produce a histogram of part-of-speech tags in the dataset – is it balanced between all tags? What word-level accuracy would a simple baseline that picked the majority label achieve?

The texts in the data seem to cover from and wide range of topics from macro ones such as economics and politics to daily life stuff like service complaints. For text preprocessing, lower casing might be appropriate because it does change the grammatical meaning of words, but helps eliminate duplicates. On the other hand, removing stopwords might lose some valuable information about the surrounding words. For example, 'the' and 'a' are usually followed by a noun phrase. Likewise, removing punctuations is not a good idea. Punctuations informs the model about the start and end of sentences, which are important for guessing grammatical roles of words. Last, emojis can be useful for the model too. For example, in the sentence "I ♥ this song so much.", ♥ is used as a verb. If the model could recognize ♥ as a verb, the predictions for the following words would be more certain.

As shown by Figure 8, the distribution of tags is not uniform. The data set has 34781 nouns but only 6707 interjection. If we guess every word as a noun, we get an accuracy of 17%.

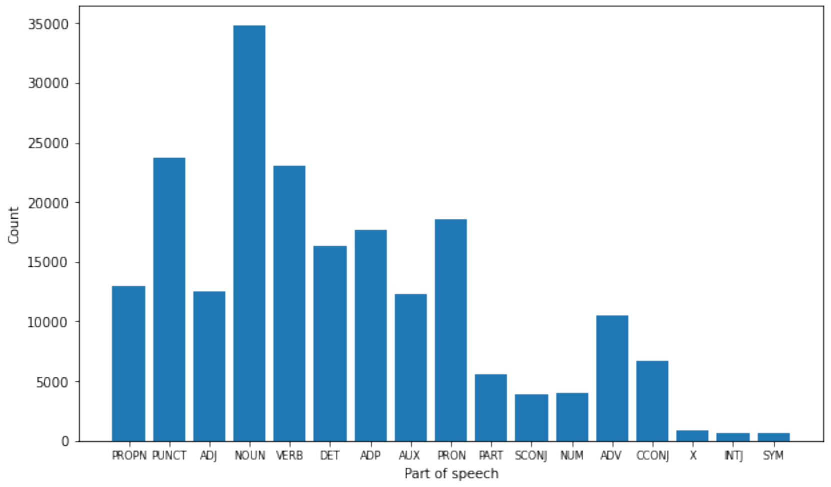


Figure 8: Tags distribution

Task 3.2
Create a file `driver_udpos.py` that implements and trains a bidirectional LSTM model on this dataset with cross entropy loss. The BiLSTM should predict an output distribution over the POS tags for each token in a sentence. In your written report, produce a graph of training and validation loss over the course of training. Your model should be able to achieve > 70% per-word accuracy fairly easily.

To achieve stronger performance, you will likely need to tune hyper-parameters or model architecture to achieve lower validation loss. Using pretrained word vectors will likely help as well. You may also wish to employ early-stopping – regularly saving the weights of your model during training and then selecting the saved model with the lowest validation loss. In your report, describe any impactful decisions during this process. Importantly – `DO NOT EVALUATE ON TEST DURING THIS TUNING PROCESS`.

Once you are done finetuning, evaluate on the test split of the data and report the per-word accuracy.

Task 3.3
Implement a function `tag_sentence(sentence, model)` that processes an input sentence (a string) into a sequence of POS tokens. This will require you to tokenize/numeralize the sentence, pass it through your network, and then print the result. Use this function to tag the following sentences:

The old man the boat.

The complex houses married and single soldiers and their families.

The man who hunts ducks out on weekends.