CS 539: Natural Language Processing with Deep Learning **(Due: February 10th, 2021)**

Submission Homework 2

Instructor: Prof. Stefan Lee *Name:* Rashmi Jadhav, *Student ID:* 934-069-574

1. Demystifying Recurrent Neural Networks

1. Derivation for the gradient expressions $\frac{\partial L(h_T)}{\partial w_{ix}}, \frac{\partial L(h_T)}{\partial b_i}$:

The basic derivatives of the *sigmoid* and *tanh* functions to be used in the further derivation are as follows:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{0.1}$$

$$tanh'(x) = 1 - tanh^2(x) \tag{0.2}$$

$\frac{\partial L(h_T)}{\partial w_{ix}}, \frac{\partial L(h_T)}{\partial b_i}$ terms would be summations of individual losses over time steps $t = 1$ to $t = T$. Deriving chain rule expression for $\frac{\partial L(h_T)}{\partial w_{ix}}$:

$$\frac{\partial L(h_T)}{\partial w_{ix}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial i_t} . \frac{\partial i_t}{\partial w_{ix}} \tag{0.3}$$

We now derive individual terms in the chain rule expression:

$$\frac{\partial L_t}{\partial i_t} = \frac{\partial L_t}{\partial h_t} . \frac{\partial h_t}{\partial c_t} . \frac{\partial c_t}{\partial i_t} \tag{0.4}$$

Since Loss function is not specified, we leave $\frac{\partial L_t}{\partial h_t}$ as is.

$$\frac{\partial h_t}{\partial c_t} = \frac{\partial}{\partial c_t}(o_t tanh(c_t))$$

Using (0.2),

$$\implies \frac{\partial h_t}{\partial c_t} = o_t.(1 - tanh^2 c_t) \tag{0.5}$$

$$\frac{\partial c_t}{\partial i_t} = \frac{\partial}{\partial c_t}(f_t c_{t-1} + i_t g_t)$$

$$\implies \frac{\partial c_t}{\partial i_t} = g_t \tag{0.6}$$

Substituting (0.5) and (0.6) back in (0.4):

$$\frac{\partial L_t}{\partial i_t} = \frac{\partial L_t}{\partial h_t} . o_t . (1 - tanh^2(c_t)) . g_t \tag{0.7}$$

Now we calculate term $\frac{\partial i_t}{\partial w_{ix}}$ from (0.3)

$$\frac{\partial i_t}{\partial w_{ix}} = \frac{\partial}{\partial w_{ix}}(\sigma(w_{ix} x_t + w_{ih} h_{t-1} + b_i))$$

Using (0.1) for sigmoid derivative:

$$\implies \frac{\partial i_t}{\partial w_{ix}} = \{\sigma(w_{ix} x_t + w_{ih} h_{t-1} + b_i).(1 - \sigma(w_{ix} x_t + w_{ih} h_{t-1} + b_i))\}.x_t$$

$$\implies \frac{\partial i_t}{\partial w_{ix}} = i_t(1-i_t).x_t \tag{0.8}$$

Substitute (0.7) and (0.8) in (0.3),

$$\frac{\partial L(h_T)}{\partial w_{ix}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t}.o_t.(1-tanh^2(c_t)).g_t.i_t(1-i_t)x_t \tag{0.9}$$

Similarly for $\frac{\partial L(h_T)}{\partial b_i}$:

$$\frac{\partial L(h_T)}{\partial b_i} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial i_t}.\frac{\partial i_t}{\partial b_i}$$

$$\frac{\partial L(h_T)}{\partial b_i} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t}.o_t.(1-tanh^2(c_t)).g_t.i_t(1-i_t) \tag{0.10}$$

2. In order to get the LSTM to solve the parity problem, it basically needs to implement the XOR logic.
A XOR B = (A OR B) AND (A NAND B)
As per the LSTM system of equations in 1.1 above, we have:
Input gate: w_{ix}, w_{ih}, b_i and w_{gx}, w_{gh}, b_g
Forget gate: w_{fx}, w_{fh}, b_f and
Output gate: w_{ox}, w_{oh}, b_o
We may map the *OR* and *NAND* operations in the *XOR* expansion to the input gates at each time step and then the collective *AND* can be made to correspond to the final point-wise multiplication as it mimics AND's behavior.

We don't really need the forget gate since we do not want to forget any input and hence we need to obtain a 0 here. Thus, we may have $w_{fx} = -10, w_{fh} = -10, b_f = -10$ as per sigmoidal calculations.

We can make the input gate i_t correspond to $NAND$ gate and with calculations, we obtain: $w_{ix} = -20, w_{ih} = -20, b_i = 30$.

The output gate o_t corresponds to OR gate and $w_{ox} = 20, w_{oh} = 20, b_o = -10$.

At g_t we want to keep the input and hence we try to reach 1. With tanh calculations, we get $w_{gx} = 0, w_{gh} = 0, b_g = 10$.

Finally,

$w_{ix} = -20, w_{ih} = -20, b_i = 30$
$w_{gx} = 0, w_{gh} = 0, b_g = 10$
$w_{fx} = -10, w_{fh} = -10, b_f = -10$
$w_{ox} = 20, w_{oh} = 20, b_o = -10$

2. Learning to Copy Finite State Machines

1. `class ParityLSTM` implementation:

```
######################################################################
# Task 2.2
######################################################################

# Implement a LSTM model for the parity task.

class ParityLSTM(torch.nn.Module):

    # __init__ builds the internal components of the model (presumably an LSTM and linear layer for classification)
    # The LSTM should have hidden dimension equal to hidden_dim

    def __init__(self, hidden_dim=64):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.lstm = LSTM(input_size=1, hidden_size=hidden_dim, num_layers=1, batch_first=True)
        self.linear = Linear(in_features=hidden_dim, out_features=2)

    # forward runs the model on an B x max_length x 1 tensor and outputs a B x 2 tensor representing a score for
    # even/odd parity for each element of the batch
    #
    # Inputs:
    #   x -- a batch_size x max_length x 1 binary tensor. This has been padded with zeros to the max length of
    #        any sequence in the batch.
    #   s -- a batch_size x 1 list of sequence lengths. This is useful for ensuring you get the hidden state at
    #        the end of a sequence, not at the end of the padding
    #
    # Output:
    #   out -- a batch_size x 2 tensor of scores for even/odd parity

    def forward(self, x, s):
        x_unsqueezed = torch.unsqueeze(x, 2)
        x_pack = pack_padded_sequence(input=x_unsqueezed, lengths=s, batch_first=True, enforce_sorted=False).to(dev)
        out_pack, ht = self.lstm(x_pack)
        return self.linear(ht[0][-1])
```

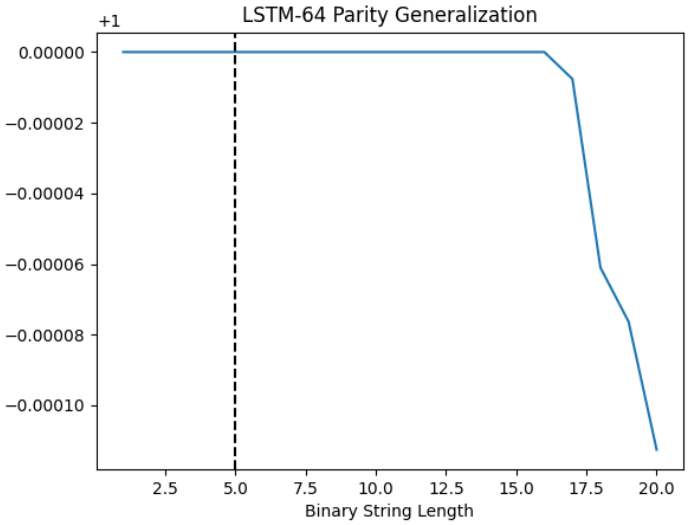
We initialize the model by setting up the hidden dimension, the LSTM function, and a linear layer for classifier in the `__init__()` function of `class ParityLSTM`. The `LSTM()` function applies a multi-layer LSTM RNN to the input sequence and for each element in the input sequence, each layer computes the LSTM system of equations for us. For our univariate LSTM, the input size is 1 and the parameters hidden_size, num_layers are configurable per experiment. The `linear()` function applies a linear transformation to the incoming data.

In forward pass function `forward()`, since `pad_sequence()` was used for converting variable length sequences to same size, we'd have to `pack_padded_sequence()` before feeding to lstm. For outputting a B x 2 tensor representing a score for even/odd parity for each element in the batch, we pass `lstm()`'s last index output to the linear function which in turn gives the output of our forward() function.

After implementing the model and running the experiment, the model successfully approached a 100% accuracy on the training set.

```
epoch 140 train loss 0.263, train acc 0.935
epoch 150 train loss 0.150, train acc 0.984
epoch 160 train loss 0.078, train acc 1.000
epoch 170 train loss 0.040, train acc 1.000
epoch 180 train loss 0.022, train acc 1.000
epoch 190 train loss 0.014, train acc 1.000
```

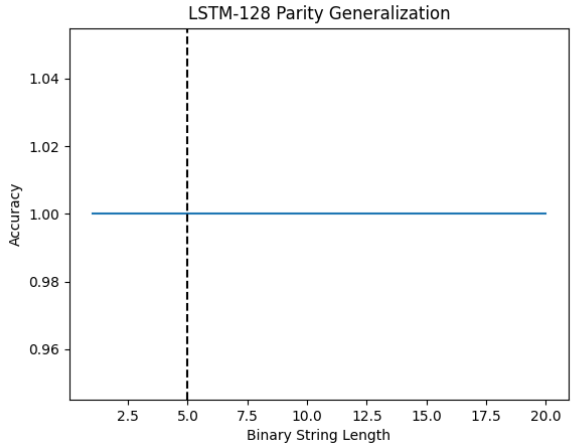
2. Accuracy per binary length plot for LSTM-64:

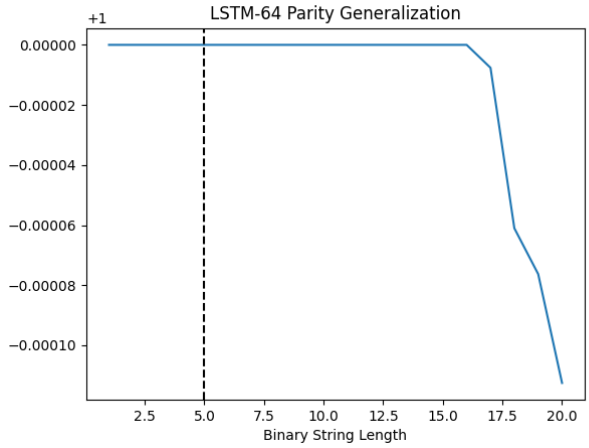


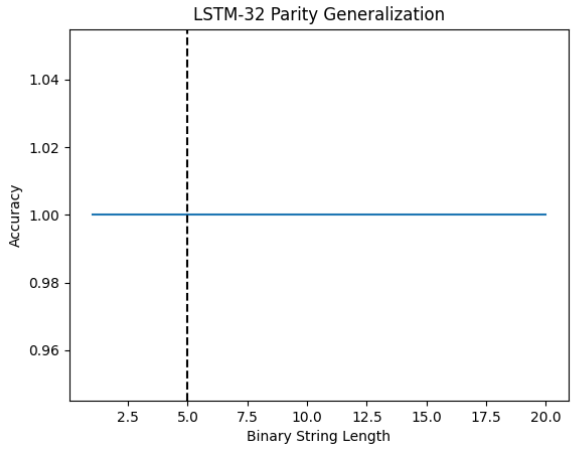
We observe the accuracy to be 1 up-to string length of 16 after which it starts to go down, although very minimally. From string length 1 to 20, the accuracy ranges from 1 to 0.9999. This can be interpreted as the model being overfit to the train dataset such that the error in validation starts to go up leading to val accuracy going down. If this is the trend for training, we can keep increasing the learning rate as the hidden size goes down.

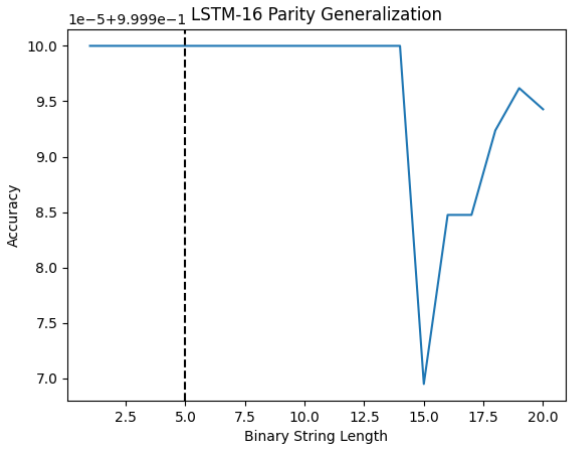
3. The training accuracy reached 1 at the 140th epoch for LSTM-128, 160th epoch for LSTM-64 and LSTM-32, 190th epoch for LSTM-16, 320th epoch for LSTM-8, and 210th epoch for LSTM-4. Lesser the hidden_size, more iterations did the training require to reach the accuracy.

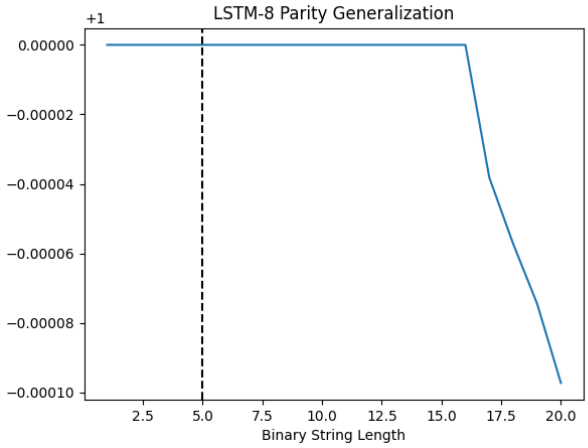
   Following are the parity generalization plots for LSTM-[128, 64, 32, 16, 8, 4]:

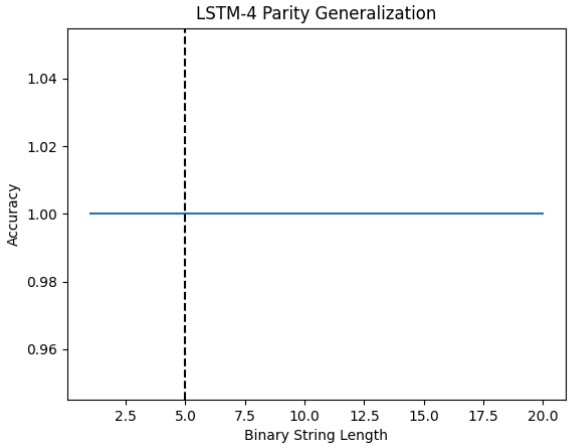












   The smallest hidden size for which this model can still train to fit the dataset is 4.

4. Theoretically, both Vanilla RNNs and LSTMs are Turing complete and they should be able to identify whether a string can be generated by a grammar or not. From the given embedded grammar automaton picture, we notice that the given ERG consists of two separate copies of Reber's grammars. All the strings generated that have T as the second character must have a corresponding T as the second last character and same goes for the P. This adds a long term sequential dependency for an NN. From the structure, `BT BTXSE BTXSE BPVVE BTXXVVE TE` there's quite a bit of sequence between the two T's at the extremes.

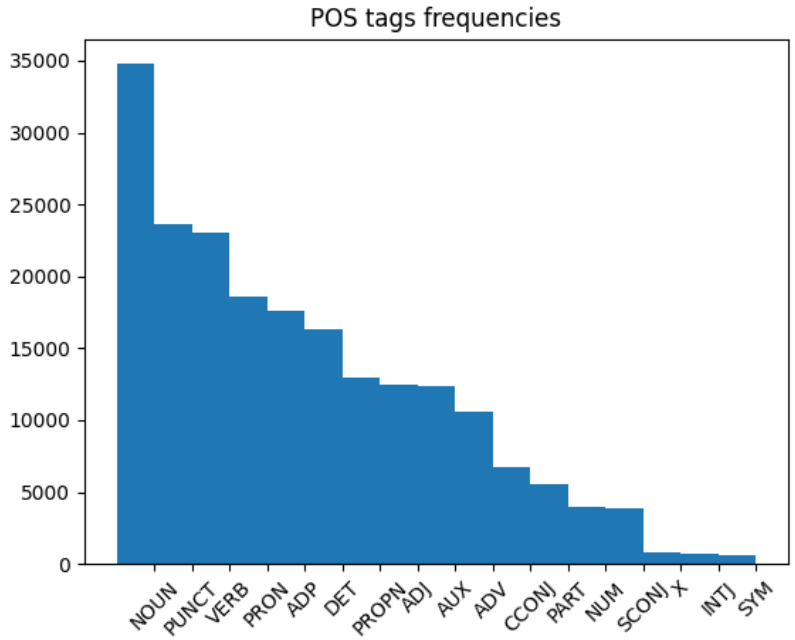
   The hidden state(which holds information of prior data seen by the network) acts as vanilla RNN's memory. If a sequence is long enough, a Vanilla RNN will have difficulty carrying information from earlier time steps to future ones. An LSTM on the other hand has a cell state and gated mechanisms which nicely determine what information to keep and what to discard. This leads to a difference that the Vanilla RNNs tend to have shorter memory and LSTMs have a long memory. Thus, it is difficult for vanilla RNNs to handle longer input strings to identify whether they can be generated by the grammar.

3. Part-of-Speech Tagging

1. The Universal Dependencies English Part-Of-Speech (UDPOS) benchmark dataset seems to have sentences which we use while communicating in our daily lives (there are a lot of pronouns in the way humans address each other). Some of the topics observed in the dataset are financial and medical industries, government/law, email language, etc.

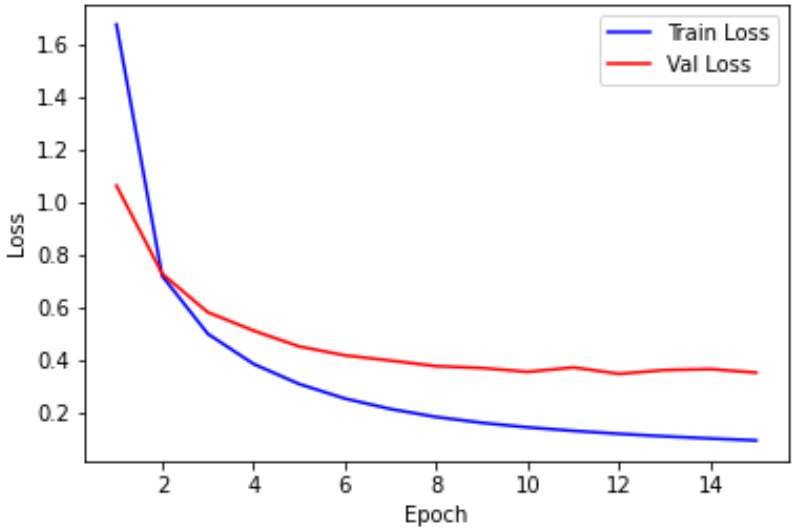
   Looking at the dataset, the sentences have already been converted to their tokenized forms. This tokenization seems pretty good because it has identified emails, phone numbers, physical addresses, dates, timestamps, contractions, and hyphenated words fairly well. Since it has already been tokenized, there's **no need for tokenization before training** here. However, if the input to the model hasn't been tokenized, it would have to be tokenized to handle the cases mentioned. Talking about lemmatization, if we lemmatize for example, `kindly` to `kind` we just lost the token's part of speech by changing it to an `adjective` form from an `adverb` form. Similar case would happen on applying stemming - chopping off from the end of word would lose its form to POS tag on it. Considering these factors, we **shouldn't use lemmatization and stemming** as well.

   Following is the histogram for POS tags in the dataset with NOUN being the most common `(34,781 nouns)` POS tag and SYM being the least common `(599 syms)`. Clearly, it is not balanced between all tags.



   If a simple baseline picked the majority label (NOUN) for any and every given token, it would still achieve a **word-level accuracy** of **0.16999095818**.

2. Bidirectional LSTM model was built to predict POS tags for tokens in a sentence. After trying out the hyper-parameters
`hidden_dim = [32, 64, 128], embedding_dim: [100, 200, 300],`
`learning_rate: [0.01, 0.001, 0.003, 0.005],`
`layers: [1, 2, 3], weight_decay: [1e-5, 1e-4],`
hidden_dim=32, embedding_dim=100, learning_rate=0.001, layers=1, weight_decay=1e-5 were chosen. Following is the plot of training and validation losses over 15 training epochs:




```
epoch=1  tr_loss=1.68    tr_accu=0.55
epoch=1  val_loss=1.07   val_accu=0.69
epoch=2  tr_loss=0.72    tr_accu=0.78
epoch=2  val_loss=0.73   val_accu=0.78
epoch=3  tr_loss=0.50    tr_accu=0.84
epoch=3  val_loss=0.58   val_accu=0.82
epoch=4  tr_loss=0.39    tr_accu=0.88
epoch=4  val_loss=0.52   val_accu=0.83
epoch=5  tr_loss=0.31    tr_accu=0.90
epoch=5  val_loss=0.46   val_accu=0.85
epoch=6  tr_loss=0.26    tr_accu=0.92
epoch=6  val_loss=0.42   val_accu=0.86
epoch=7  tr_loss=0.22    tr_accu=0.93
epoch=7  val_loss=0.40   val_accu=0.87
epoch=8  tr_loss=0.19    tr_accu=0.94
epoch=8  val_loss=0.38   val_accu=0.88
epoch=9  tr_loss=0.17    tr_accu=0.95
epoch=9  val_loss=0.37   val_accu=0.89
epoch=10         tr_loss=0.15    tr_accu=0.96
epoch=10         val_loss=0.36   val_accu=0.89
epoch=11         tr_loss=0.13    tr_accu=0.96
epoch=11         val_loss=0.38   val_accu=0.88
epoch=12         tr_loss=0.12    tr_accu=0.96
epoch=12         val_loss=0.35   val_accu=0.89
epoch=13         tr_loss=0.11    tr_accu=0.97
epoch=13         val_loss=0.37   val_accu=0.88
epoch=14         tr_loss=0.11    tr_accu=0.97
epoch=14         val_loss=0.37   val_accu=0.88
epoch=15         tr_loss=0.10    tr_accu=0.97
epoch=15         val_loss=0.36   val_accu=0.89
```

GloVe 100 dimensional embeddings pre-trained on 6 billion tokens were used to improve the performance. For the purpose of not overfitting to the training set leading to worse val losses, lowest val-loss model was regularly saved. The lowest validation loss obtained was 0.35 at epoch 12. Using this model, a **test loss of 0.36 and a test per-word accuracy of 0.88** was obtained which seems decent.

3. For tagging a sentence, it was first tokenized into a list of tokens using SpaCy's tokenizer since manual one was already tried out in previous homework and builtin was to be explored. This list of tokens was converted to lower case as per the UDPOS dataset. It was then numeralized using previously built vocabulary.The numeralized tokens list was used to form a tensor which was then passed through the model.The model's predicted output for 3 given examples is as follows:

```
---------------------------
the     =>      DET
old     =>      ADJ
man     =>      NOUN
the     =>      DET
boat    =>      PROPN
.       =>      PUNCT
---------------------------
the     =>      DET
complex =>      ADJ
houses  =>      NOUN
married =>      VERB
and     =>      CCONJ
single  =>      ADJ
soldiers        =>      NOUN
and     =>      CCONJ
their   =>      PRON
families        =>      NOUN
.       =>      PUNCT
---------------------------
the     =>      DET
man     =>      NOUN
who     =>      PRON
hunts   =>      ADV
ducks   =>      ADV
out     =>      ADP
on      =>      ADP
weekends        =>      NOUN
.       =>      PUNCT
---------------------------
```

Sentence 1, *boat* is tagged as *proper noun*, I was expecting a noun.
Sentence 2, this seems to be correctly tagged.
Sentence 3, *ducks* is tagged as *adverb* instead of *noun*.