AI539 Natural Language Processing with Deep Learning – Homework 2

Recurrent Neural Networks and Sentence Representations

Overview and Objectives. In this homework, we'll get more familiar with LSTMs by doing a bit of math, interacting with simple versions in toy settings, and then move on to developing a full pipeline for part-of-speech tagging in English with a Bidirectional LSTM.

How to Do This Assignment. The assignment walks you through completing the provided skeleton code and analyzing some of the results. Anything requiring you to do something is marked as a "Task" and has associated points listed with it. You are expected to turn in both your code and a write-up answering any task that requested written responses. Submit a zip file containing your completed skeleton code and a PDF of your write-up to Canvas.

Advice. Start early. Students will need to become familiar with `pytorch` for this and future assignments. Extra time may be needed to get used to working remotely on the GPU cluster here. You can also use GPU-enabled runtimes in Colab `colab.research.google.com`.

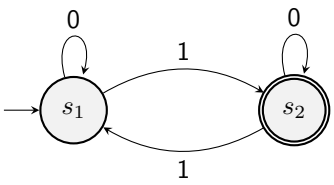
1 Demystifying Recurrent Neural Networks [5pt]

1.1 Hand Designing A Network for Parity

Consider a univariate LSTM defined given the following equations. We refer to this model as univariate because all inputs, outputs, and weights are scalars. $\sigma(\cdot)$ is a sigmoid activation.

$$
\begin{aligned}
i_t &= \sigma(w_{ix}x_t + w_{ih}h_{t-1} + b_i) && (1)\\
f_t &= \sigma(w_{fx}x_t + w_{fh}h_{t-1} + b_f) && (2)\\
o_t &= \sigma(w_{ox}x_t + w_{oh}h_{t-1} + b_o) && (3)\\
g_t &= \tanh(w_{gx}x_t + w_{gh}h_{t-1} + b_g) && (4)\\
c_t &= f_t c_{t-1} + i_t g_t && (5)\\
h_t &= o_t \tanh(c_t) && (6)
\end{aligned}
$$

To ground this complicated looking model, we will consider a simple sequence classification problem – parity of binary strings. Given an arbitrarily-long binary string $b \in \{0,1\}^*$, classify whether it has an even or odd number of ones. For example all these strings have an even parity – `‘‘ ’’`, `0000`, `101101` – and these an odd parity – `1`, `0011001`, `00011100`. Below is a simple DFA[1] that accepts strings with odd parity at the accepting state s_2. As it only has two states, it seems solving the parity classification problem only requires storing 1 bit of information in memory.



Another way to think about this computation is as a recursive application of `XOR` over time. If the parity up to element $t-1$ is 1 (odd) and we see a 1 for element t, the updated parity is 1 XOR $1 = 0$ (even). Likewise, if at time $t+1$ we observe another 1, 0 XOR $1 = 1$. In other words, PARITY$(X,t) = X[t]$ XOR PARITY$(X,t-1)$ and PARITY$(X,-1) = 0$ by definition. This sounds like the sort of operation we can do with an LSTM!

[1] `https://en.wikipedia.org/wiki/Deterministic_finite_automaton`

▶ TASK 1.1 [5 pt] 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.

Through this exercise, we've shown that a 1-dimensional LSTM *is sufficient to solve parity of binary strings of arbitrary length.* In the next section, we'll try to actually train an LSTM for this task. Hint: It may be harder than you think.

2 Learning to Copy Finite State Machines [10pt]

Let's probe LSTMs a bit by making them mimic simple finite state machines. The rest of this assignment will work within the PyTorch deep learning framework. You'll need to install PyTorch in your development environment.

2.1 Parity: Examining Generalization To Longer Sequences

First, lets try to train an LSTM to perform the parity task we just did manually. The `driver_parity.py` file provide skeleton code for these experiments including a parity dataset shown below. During training, it generates all binary strings of length 0 to `max_length`. At inference, it generates all binary strings of length `max_length`.

```
class Parity(Dataset):

    def __init__(self,split="train", max_length=5, max_test_samples=500):
      if split=="train":
        self.data = []
        for i in range(1,max_length+1):
          self.data += [torch.FloatTensor(seq) for seq in itertools.product
    ([0,1], repeat=i)]
      else:
        self.data = [torch.FloatTensor(np.random.randint(2, size=max_length)
    ) for i in range(max_test_samples)]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        x = self.data[idx]
        y = x.sum() % 2
        return x,y
```

Listing 1: Parity Dataset

To train our network, we can build a DataLoader from our Parity dataset (see below). However, our dataset contains strings of variable length so we need to pass a custom function to our data loader describing how to combine elements in a sampled batch together.

```
    train = Parity(split='train', max_length=max_length)
    train_loader = DataLoader(train, batch_size=B, shuffle=True,
                              collate_fn=pad_collate)
```

Listing 2: Data loader

Our `pad_collate` function puts all the elements of a batch in a $B \times T_{\text{max}}$ tensor for inputs and a B tensor for outputs where T_{max} is the largest length in the batch. Shorter sequences are zero-padded. A batch of inputs looks like:

```
    tensor([[1., 1., 0., 0., 0.],
            [0., 1., 0., 1., 0.],
            [0., 0., 0., 0., 0.],
            [0., 1., 1., 0., 1.],
            [0., 1., 1., 0., 0.]], device='cuda:0')
```

Listing 3: Example input for a batch

▶ TASK 2.1 [5 pt] 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 but often generalize more poorly to new sequences. Running `driver_parity.py` will train your model and output per-epoch training loss and accuracy. A correctly-implemented model should approach or achieve 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.

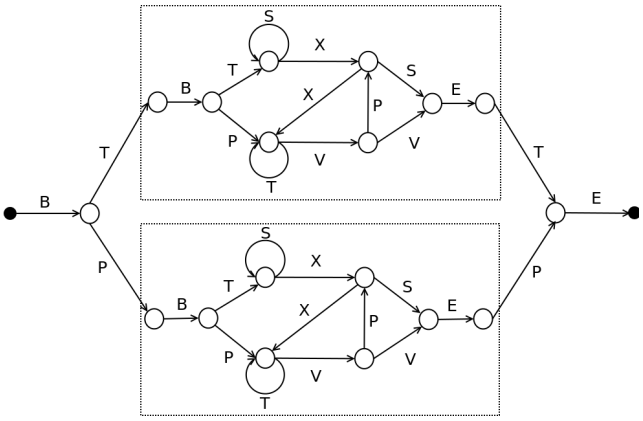
▶ TASK 2.2 [1 pt] `driver_parity.py` also evaluates your trained model on binary sequences of length 1 to 256 (for 500 samples each) and saves a corresponding plot of accuracy vs. length. Include this plot in your write-up and describe the trend you observe. Why might the model behave this way?

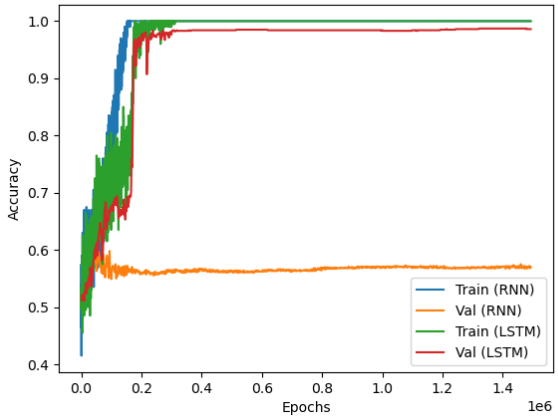
▶ TASK 2.3 [3 pt] We know from 1.1 that even a univariate LSTM (one with a scalar hidden state) can theoretically solve this problem. Run a few (3-4) experiments with different hidden state sizes, what is the smallest size for which you can still train to fit this dataset? Feel free to adjust any of 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.

Note: Don't spend all night tuning this unless you've finished the rest of the assignment and have nothing better to do.

2.2 Embedded Reber Grammars: Thinking About Memory

Next we turn to recognizing an Embedded Reber Grammar (ERG). An ERG generation diagram is shown below. To generate an ERG string, start from the leftmost black node and sample any outgoing edge (only B in this case) and write down the letter of that edge. Repeat this process from the new node until reaching the right-most black node through the edge labelled E. So `BTBTSSXSETE` is a valid ERG string. Notice that the sub-networks in dashed boxes are identical for the upper and lower branch. These are the Reber Grammars that are embedded in this larger grammar. To make longer strings, the dashed boxes can be repeated k times (E edge returns to start) before exiting to the T/P edges headed towards the terminal node on the right.





The plot on the right shows train and validation accuracy over training for an LSTM and vanila RNN on the task of identifying ERG strings with $k = 4$. An example string from this grammar is `BTBTXSEBTXSEBPVVEBTXXVVETE`. Breaking it out, its easy to see the structure – `BT BTXSE BTXSE BPVVE BTXXVVE TE` – as a prefix, four repeated Reber Grammar strings, and a suffix. The dataset contained 50% positives samples, 25% negative with a random perturbation of one character, and 25% negatives with the second-to-last letter perturbed. In this setting, the vanilla RNN model overfit to the training set given sufficient time, but failed to generalize. In contrast, the LSTM model generalized well to the test set.

▶ `TASK 2.4 [1 pt]` 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?

3 Part-of-Speech Tagging [15pt]

Now lets get to work on a real problem – part-of-speech tagging in English. This is a many-to-many sequence problem where each word in an input must be labelled with a corresponding part-of-speech tag. The goal of this section will be to train a bidirectional LSTM model for this task. Unlike in the previous sections, no skeleton code will be provided and students are responsible for building the whole pipeline including text processing, data loading, model design, training, and evaluation. Feel free to use any libraries you wish to accomplish this.

We will be using the Universal Dependencies[2] English Part-Of-Speech (UDPOS) benchmark. The easiest way to load this dataset is through the `torchtext` module. The code below loads the dataset and prints out an example. Note you will need to install the `torchtext` and `portalocker v2.8.2` Python packages.

[2] https://universaldependencies.org/

```python
import torch
from torch.utils.data import DataLoader
from torchtext import datasets
from torch.utils.data.backward_compatibility import worker_init_fn

# Create data pipeline
train_data = datasets.UDPOS(split='train')

# Function to combine data elements from a batch
def pad_collate(batch):
  xx = [b[0] for b in batch]
  yy = [b[1] for b in batch]

  x_lens = [len(x) for x in xx]

  return xx, yy, x_lens

# Make data loader
train_loader = DataLoader(dataset=train_data, batch_size=5,
                          shuffle=True, num_workers=1,
                          worker_init_fn=worker_init_fn,
                          drop_last=True, collate_fn=pad_collate)

# Look at the first batch
xx,yy,xlens = next(iter(train_loader))

# Visualizing POS tagged sentence
def visualizeSentenceWithTags(text,udtags):
  print("Token"+"".join([" "]*(15))+"POS Tag")
  print("-----------------------------------")
  for w, t in zip(text, udtags):
    print(w+"".join([" "]*(20-len(w)))+t)
```

Listing 4: Loading the UD En POS benchmark

```
> visualizeSentenceWithTags(xx[0],yy[0])
Token               POS Tag
---------------------------------
They                PRON
are                 AUX
doing               VERB
it                  PRON
deliberately        ADV
.                   PUNCT
```

Listing 5: Visualizing a sentence and its associated list of tags.

Note that there is a train, validation, and test split of the dataset specified by the `split` argument. We should only evaluate on test once when we are completely done tuning our model on validation and are reporting final results.

▶ TASK 3.1 [2 pt] 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?

You will need to get the dataset into a form that we can use in a model – i.e. numeralized batches of sequences and their labels. Simple examples of dataset classes and dataloaders that rely on our Vocabulary code from last assignment have been demonstrated in lecture. Another option is to look into the `torchtext.data` APIs which can flexibly handle text datasets like this one and take care of tokenization / numeralization with less effort – provided you spend the time learning the API. Either way is fine for this assignment.

The next step is to implement your model and the training routine. The lectures and other skeleton code in this assignment should provide a good starting point for these. If you run into trouble, printing out the shapes of tensors and reading the PyTorch documentation can be very helpful. A useful development strategy is to operate only on a single batch at first – if your model can't quickly fit the batch, something is wrong. Once you get a basic model running, you'll want to tune parameters and network structure to achieve good validation accuracy.

▶ TASK 3.2 [10 pt] 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 [3 pt] 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.