RNN Acceptors and BiRNN Transducers

Yoav Goldberg

This assignment has three parts, each for a third of the weight of the assignment.

- In part 1, you will implement an RNN acceptor and train it on a specific language.
- In part 2, you will explore the capabilities of the RNN acceptor.
- In part 3, you will implement a bi-LSTM tagger.

It is recommended (in terms of computation time) to do the assignment in DyNet, but as before the choice of toolkit is up to you.

Part 1: RNN Acceptor

(the actual steps are detailed in 1.1, 1.2, 1.3)

As discussed in class, a recurrent neural-network *acceptor* is a recurrent network (RNN) that reads an input sequence one word at a time, and returns a vector representation of the entire sequence. Then, the output vector of the RNN is used as an input for a classification network (for example an MLP) to classify the entire sequence.

A bit more formally, if your input is a sequence of n vectors $x_1, ..., x_n$, then the output of the network is $MLP(RNN(x_1, ..., x_n))$ where the RNN is mapping the sequence into a vector in R^{d_1} and the MLP is used to classify this d_1 dimensional vector.

In part one of the assignment, you need to implement an neural network that will use the Long-Short-Term-Memory (LSTM) kind of RNN, followed by an MLP with one hidden layer, in order to perform binary classification of input sequences. That is, the network needs to learn to distinguish good sequences from bad ones. Try your code on an artificial language. Your positive examples should be of the form:

```
[1-9]+a+[1-9]+b+[1-9]+c+[1-9]+d+[1-9]+
```

That is, a sequence of random digits, followed by a sequence of `a`, followed by another sequence of random digits, followed by a sequence of `b` followed by another sequence of digits, then a sequence of `c` and a another sequence of random digits, then a sequence of d followed by a final sequence of random digits.

The negative examples should be of the form:

`[1-9]+a+[1-9]+c+[1-9]+b+[1-9]+d+[1-9]+`

That is, just like before, but this time the sequence of `c` appear before the sequence of `b`.

1.1 Understanding the Challenge

- Can the two languages be distinguished using a bag-of-words approach? Explain why.
- Can the two languages be distinguished using a bigram or trigram based approach? Explain why.
- Can the two languages be distinguished using a convolutional neural network? Explain why.

Write your answers in a PDF file named `challenge.pdf`. Make sure to include your name and ID number in this file.

1.2 Generating the Data

Write code for generating positive and negative examples for this task. Submit the code in a file named `gen_examples.py`. Submit a sample of 500 positive examples and 500 negative examples in files named `pos_examples` and `neg_examples`.

Generate a training and test sets for this problem.

1.3 Writing the RNN acceptor network

Write code for conducting the neural network experiment: training on the training set and testing on the test set. Submit your code in a file named `experiment.py`, as well as explanation on how to run it in a text file named `README.acceptor.txt`.

Provide a PDF file named `report1.pdf`, in which you provide a summary of the experiment: how large were the training and test sets, did your network succeed in distinguishing the two languages (it should)? how long did it take (both wall-clock time (i.e., number of seconds), and number of iterations)? did it succeed only on the train and not on the test? what you did in order to make it work, etc.

Part 2: Acceptor Capabilities: try to make it fail

In this part, your job is to try and make the network fail. Try to devise two classes of sequences that the RNN cannot distinguish, even after seeing many training examples, and running for many training iterations.

You can think of it in terms of formal language theory: you define a language over an alphabet, and the RNN needs to learn to decide if a string is in the language or not.

Provide a report (in a file named `report2.pdf`) in which you describe the 3 most challenging cases you came up with. For each one, provide:

- A description of the languages.
- Why did you think the language will be hard to distinguish?
- Did you manage to fail the LSTM acceptor? (including, train and test set sizes, how many iterations did you train for, did it manage to learn the train but did not generalize well to the test, or did it fail also on train?)

Part 3: BiLSTM Tagger

In this part, you will implement a 2-layer biLSTM tagger with different forms of input representations.

Use the same datasets (NER and POS) from assignment 2. You will implement the following model(s):

For an input sequence $w_1, ..., w_n$, represent each item as a vector $\mathbf{x}_i = repr(w_i)$. Then, feed these representations through a biLSTM, resulting in n vectors $\mathbf{b}_1, ..., \mathbf{b}_n$ where:

$$\mathbf{b}_i = biLSTM(\mathbf{x}_1, ..., \mathbf{x}_n; i) = LSTM_F(\mathbf{x}_1, ..., \mathbf{x}_i) \circ LSTM_B(\mathbf{x}_n, ..., \mathbf{x}_i)$$

These will be fed to another layer of biLSTM, resulting in vectors $\mathbf{b'}_1, ..., \mathbf{b'}_n$ where $\mathbf{b'}_i = biLSTM(\mathbf{b}_1, ..., \mathbf{b}_n; i)$.

Each vector $\mathbf{b'}_i$ will be fed into a linear layer followed by a softmax for predicting the label y_i. Use the cross-entropy loss.

Each word will be represented in one of the following options:

(a) an embedding vector: $repr(w_i) = \mathbf{E}_{[w_i]}$
(b) a character-level LSTM: $repr(w_i) = repr(c_1, c_2, ..., c_{m_i}) = LSTM_C(\mathbf{E}_{[c_1]}, ..., \mathbf{E}_{[c_{m_i}]})$.
(c) the embeddings+subword representation used in assignment 2.
(d) a concatenation of (a) and (b) followed by a linear layer.

(you need to implement all of them)

Submit 2 files: `bilstmTrain.py`, `bilstmPredict.py`.
They should be run as:

```
bilstmTrain.py repr trainFile modelFile [options]
bilstmPredict.py repr modelFile inputFile [options]
```

Where `repr` is one of `a,b,c,d`, `trainFile` is the input file to train on, `modelFile` is the file to save/load the model, and `inputFile` is the blind input file to tag. You can also add whatever additional options you may need. Document your options in a `README.bilstm.txt` file.

The tagger should train for 5 iterations over the training data. Every 500 sentences, compute your dev-set accuracy.

Submit a file called `bilstm.pdf`. It should include the parameters you used for train your models (layer sizes, learning rate, optimizers, dropout if you used it, etc), as well as 2 graphs.

Graph 1 is the learning curves for the POS data (the dev-set accuracies). It should have 4 lines, corresponding to input representations (a), (b), (c), (d) above. Graph 2 is the learning curves for the NER data, again with 4 lines.

Each graph should be be accuracy (y-axis) vs. (number of sentences seen / 100) (x-axis).

Indicate clearly which graph is which condition, and which line is which input representation.

Submit also the predictions of your best model on the blind test sets, in files called `test4.pos` and `test4.ner`.