HW4-11763

Submission Deadline:17th December 2015

1 Introduction

In this homework, you will perform Named Entity Recognition by using sequential neural networks. This assignment is open-ended with a few compulsory deliverables for minimum credit. Recurrent Neural Networks (RNNs) and Long Short Term Memory (LSTMs) networks are widely used sequential neural models. For the mandatory part, we will work with RNNs and the general architecture and procedure for tagging with RNNs is described in the following sections. This description is fairly restricted and variations on almost every modelling decision can be incorporated.

2 Recurrent Neural Networks

RNNs are neural networks that compute representations of sequences of inputs. That is, at time step t they are presented with an input $\mathbf{x}_t \in R^n$ and they compute a representation of that input together with all the previous inputs. They differ from standard feed-forward neural network primarily because they work with inputs of arbitrary length rather than a fixed size. Hence, at an abstract level, RNNs take as **input**, a sequence $\boldsymbol{x}_{1:n}$ and initial internal state $\mathbf{s}_0$ (which is usually optimized as a parameter), and generate a **output** sequence of states $\boldsymbol{s}_{1:n}$ representing the history of events. Although $\mathbf{s}_t$ represents an unbounded history, it is defined recursively as a function of a single previous RNN state $\mathbf{s}_{i-1}$ and the current input vector $\mathbf{x}_i$:

$$\mathbf{s}_t = \mathbf{S}(\mathbf{x}_t, \mathbf{s}_{t-1})$$

where $\mathbf{S}$ is a vector-valued function. Commonly,

$$\mathbf{s}_i = \phi_s(\mathbf{W}_h \mathbf{s}_{i-1} + \mathbf{W}_x \mathbf{x}_i + \mathbf{b}_s).$$

Where the parameters are $\mathbf{W}$'s which are matrices and $\mathbf{b}$ which is a bias vector. ϕ is nonlinear activation function, usually tanh. The initial state value $\mathbf{s}_0$ is treated as a parameter. All parameters are estimated to minimize loss on some prediction task, as discussed in class.[1]

How are RNNs used to make predictions? Typically, RNNs make a prediction at each time step t. In language modeling, this will be a prediction of the next word in a sequence (here $\mathbf{s}_t$ represents the entire historical context, making RNN LMs very powerful). In tagging problems, $\mathbf{s}_t$ will be used to predict the tag y_t. The loss function is up to the user, but in discrete sequence models, it is usually the cross-entropy at the next position (we discuss cross-entropy in the next section).

[1] As an alternative to this definition of $\mathbf{S}$, one can use long short-term memories (LSTMs). These are a variant of RNNs, but the transformation $\mathbf{S}$ more complicated, but these often have an easier time modeling long-range dependencies. However, RNNs are simpler to implement and understand, so the mandatory deliverables will have you work with RNNs.

3 NER with Simple RNNs

In this homework we focus on using RNNs to make named entity recognition (NER) predictions. That is, given the sequence *John Smith visited Pittsburgh*, we wish to predict `B-PER I-PER O B-LOC` indicating that token 1 "begins" a segment labeled with `PER`, token 2 continues it, token 3 is not a named entity, and token 4 is the beginning of a segment labeled as `LOC`.

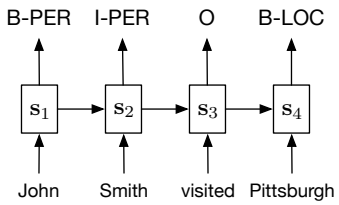

Figure 1: RNN-based NER tagging model.

Figure 1 depicts the RNN-based tagging model graphically. In this figure, an arrow represents an affine transformation followed by a nonlinearity.

There are two questions to be discussed: (i) how to obtain a representation $\mathbf{x}_t$ for each word w_t and (ii) how to make a prediction $y_t \in Y$. To represent words, we will use a projection of a one-hot embedding of a word into a d-dimensional vector space. That is each word type will be associated with a vector representation $\mathbf{x}_t$. To make a prediction we will project each $\mathbf{s}_t$ onto a $|Y|$-dimensional vector space, creating a vector $\mathbf{r}_t$. That is,

$$\mathbf{r}_t = \mathbf{W}_{out}\mathbf{s}_t + \mathbf{c}$$

where $\mathbf{W}_{out}$ is a $|Y| \times d$ dimensional matrix and $\mathbf{c}$ is a $|Y|$-dimensional vector. Next, we use a softmax operation to normalize this vector of scores into a probability distribution over tages.

$$p(y_t = y \mid \mathbf{r}_t) = \frac{\exp r_{y,t}}{\sum_{y' \in Y} \exp r_{y',t}}$$

Training criterion. For training, we can minimize the cross entropy, CE of the estimated model p relative to an empirical distribution $\tilde{p}$ each time step:

$$CE_t = \sum_{y' \in Y} \tilde{p}(y_t = y') \log \left(\frac{\tilde{p}(y_t = y')}{p(y_t = y' \mid \mathbf{r}_t)} \right)$$

Generally, for NER training, there is just one correct tag so the true distribution $\tilde{p}$ is the true distribution over tags at each position (that is, it will be one for the correct tag and 0 elsewhere). p is the probability distribution over the tags, estimated by the RNN. Hence, the loss at position i becomes:

$$CE_t = -\log p(y_t \mid \mathbf{r}_t)$$

For the whole sequence, we simply add up the CE at all the positions in the sequence.

Prediction. To make predictions at test time, we simply take the arg max from the prediction vector for each position t in the sequence. Since each prediction is independent of the others in this model, this greedy decoding algorithm is guaranteed to be correct.

Deliverable 1. Implement a simple RNN tagger and report the results you obtain on the provided development data. (You should hold out the development data from training, but you can use it to validate your model.) Report both the per-token accuracy and the F-measure. Submit the code and the tagging output on the test data.

4 Bi-directional RNNs

A simple extension to the tagger above is to use two RNNs which process a sequence in opposite directions. The rationale behind doing this is that this enables the tagger to be aware of both the left and right contexts when making predictions. The computation is very similar to the unidirectional RNN case. For each position t, we obtain states $\overrightarrow{\mathbf{s}_t}$ and $\overleftarrow{\mathbf{s}_t}$ and combine these as follows:

$$\mathbf{s}_t = \phi\left(\mathbf{W}_c[\overrightarrow{\mathbf{s}_t}; \overleftarrow{\mathbf{s}_t}] + \mathbf{b}_c\right)$$

where $[\mathbf{a}; \mathbf{b}]$ is the concatenation of vectors $\mathbf{a}$ and $\mathbf{b}$, and $\mathbf{W}_c$ and $\mathbf{b}_c$ are parameters. This state $\mathbf{s}_t$ is used as input to compute $\mathbf{r}_t$ as above.

Figure 2 (left side) shows the architecture.

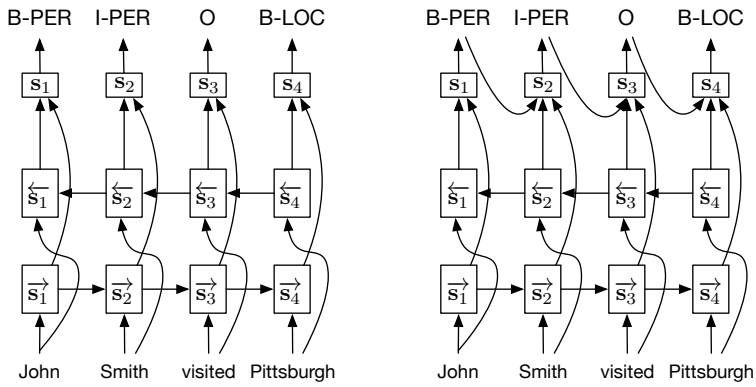

Figure 2: Two neural NER models based on bidirectional RNNs: independent classification (left); structured model (right).

Training and prediction. Although this model is more sophisticated than the previous one, both the training criterion and the prediction algorithm stay the same.

Deliverable 2. Implement bi-directional RNN. Compute the performance on the development data, measured in terms of per-token accuracy and F measure. Submit the code and the tagging output on the test data.

5 Adding structure

In the previous two models, when we predict a tag y_t at time t, we only need the state $\mathbf{s}_t$, which depends only on the inputs $\boldsymbol{x}_{1:n}$. That is, the tag probabilities are conditionally independent given $\boldsymbol{x}_{1:n}$.

In this section, we add statistical dependencies from y_t onto y_{t-1}. Since tags, like words, are discrete objects, and we are working with vector-based models, we will need to introduce a **tag**

embedding. For each tag $y \in Y$, refer to $\mathbf{e}_y$ as the tag embedding. Thus, the embedding of the tag predicted at y_t will be written $\mathbf{e}_{y_t}$.

The model you should implement is illustrated in Figure 2 on the right side. It is similar to the previous model except we define

$$\mathbf{s}_t = \phi\left(\mathbf{W}'_c[\overrightarrow{\mathbf{s}_t}; \overleftarrow{\mathbf{s}_t}; \mathbf{e}_{y_{t-1}}] + \mathbf{b}'_c\right)$$

Use an auxiliary START tag for y_0 as the first tag of the sequence.

Training and prediction. The training criterion for this model stays the same as in the previous two tagging models(Cross entropy). However, since there are now statistical dependencies between adjacent tags, finding the best tag sequence requires using the Viterbi algorithm.

Deliverable 3. Implement code to train the *structured* bi-directional RNN to perform NER on the given data. Submit the code and the tagging output on the test data.

Deliverable 4 . Submit the curves with training loss on the y axis vs the number of epochs during training. Optimize for 20 epochs.[2] Compare the curves for the normal bi-RNN and the structured bi-RNN.

6 Questions

1. Compare the structured and unstructured bi-RNNs in terms of their performance and learning behavior during training.
2. The unstructured BiRNN does not depend on the previously generated tags. Then, why does it do so well?
3. In the structured model, we just had you look at tag $\mathbf{y}_{t-1}$ when predicting $\mathbf{y}_t$. Would it be possible to condition on $\mathbf{y}_{t+1}$? How would this complicate training and/or prediction?

7 Make it better!

This is an open ended assignment and there are many relevant extensions that can potentially improve the performance of your tagger. The top performing systems will get extra credit. Some extensions to get you thinking about ideas:

- Try LSTMs instead of simple RNNs.
- Add cost to prevent the model from predicting too many `O`'s using softmax margin. For hints, refer to:
  `http://homes.cs.washington.edu/~nasmith/papers/gimpel+smith.naacl10.pdf`
- Add recurrent connections between the $\mathbf{s}_t$'s in the structured mode. Note that you won't be able to have a bidirectional model at the top layer, but you will be able to have a forward RNN. This will complicate your decoder (Viterbi will no longer work!). You will need to use either a greedy approximation or beam search using a k-best list. This will involve maintaining k-best partial tagging sequences and k RNNs for each of the partial sequences.

[2] 'epoch' is a complete pass through the training data.

- Try conditioning on a larger tag history for the structured BiRNN.
- Experiment with the architecture of your network. You can add multiple RNN layers to generate several stacked state vectors. You can also make the **O** function more complex by adding more layers and non linearities. Experiment with different sizes of hidden layers and different non linearities.
- Try incorporating informative features related to the word types at each position into the raw representation for each word which gets projected to a lower dimensional space using ϕ_{lookup} to construct the input vector for the RNNs.
- Try using different optimization schemes.

Deliverable 5 : Submit a **detailed report** containing the description of approaches you tried to improve the performance of your tagger. Also, discus how those approaches affected the learning of parameters, performance on development set etc. Describe approaches that worked and that didn't work.

8 Implementation recommendation

We recommend using existing neural net libraries to avoid having to code back propagation through computation DAGs/trees from scratch. Any NN library is fine to use but we will be able to best help you if you use the C++ library *cnn* (https://github.com/clab/cnn) or its python wrapper *pycnn* (https://github.com/clab/cnn/tree/master/pycnn).
We strongly recommend you to look at the example RNN and bi-LSTM implementations in ***examples/*** and the ***pyexamples/*** directories.
Also, it id advisable to go through the source code in the ***cnn/*** directory to understand how the computation happens. In particular these files should be helpful:

```
--> /cnn/expr.* -- For checking the supported functions for implementing
non linearities and other mathematical operations.
--> /cnn/rnn* -- For inderstanding how the
RNN state computations are implemented.
--> /cnn/cnn.* -- For understanding the general
organization of the library.
--> /cnn/training.* -- For understanding implementation of various online
optimizers like SGD, adagrad etc.
```

9 Submission instructions

Email your best output, deliverable 1 output, deliverable 2 output, deliverable 3 output, deliverable 4 output, the relevant code, performance on the development set, the report as deliverable 5 and the answers to the questions as a .zip or .tgz submission with the subject 'assignment four - andrew id' to cmu-instructors- 11763-fa2015@googlegroups.com by 11:59pm on the due date.