Structured Prediction
HW4

Submission Deadline: May 4 2017

1 Introduction

In this homework, you will perform Named Entity Recognition by using a sequential neural CRF and perform a task of your choice using a structured neural CRF of your own design. Sections 2-4 of this assignment first walk you through a baseline neural model for NER that does not use sequential structure. You do not have to implement this baseline; it is presented here for reference and clarity. Section 5 describes a sequential neural CRF that is parmaeterized using an RNN. This model you will implement. Section 6 describes the last part of the assignment where you define your own task and model.
Note: While this project does have an open-ended component, it is not intended to be a 'course project' and should require considerably less effort. You are only required to extend your neural CRF implementation to consider a new task.

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 $\boldsymbol{x}_t \in \mathbb{R}^n$ 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, RNNs take a sequence of vectors $\boldsymbol{x}_{1:n}$ as input and generates a sequence of vectors $\boldsymbol{s}_{1:n}$ as output. Each state vector $\boldsymbol{s}_t$ is a representation of the input sequence up to timestep t. (The initial state vector $\boldsymbol{s}_0$ is part of the RNN's parameters.) State vector $\boldsymbol{s}_t$ is defined recursively as a function of the previous RNN state $\boldsymbol{s}_{t-1}$ and the current input vector $\boldsymbol{x}_t$:

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

where $\boldsymbol{S}$ is a vector-valued function. A common paramterization is as follows:

$$\boldsymbol{s}_t = \phi(\boldsymbol{W}_s \boldsymbol{s}_{t-1} + \boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{b}_s)$$

where matrices $\boldsymbol{W}_s$ and $\boldsymbol{W}_x$, bias vector $\boldsymbol{b}$, and the initial state vector $\boldsymbol{s}_0$ are parameters, and ϕ is a nonlinear activation function (e.g. tanh or logistic).[1]

How are RNNs used to make predictions? Typically, a classifier makes a prediction (independently) at each timestep t based on the RNNs state vector $\boldsymbol{s}_t$. In a language model, at each timestep,

[1]The LSTM is a variant that uses a more complex recurrence function $\boldsymbol{S}$ and in some cases can more easily learn long-range dependencies. In contrast, basic RNNs are easier implement and understand. You are free to work with either.

a logistic regression classifier predicts the next word in the sequence conditioned on the state vector $\boldsymbol{s}_t$. For tagging problems, each tag *boldsymboly*$_t$ is predicted by a classifier conditioned on $\boldsymbol{s}_t$. The choice of classification model determines the loss function used for training. Logistic regression yields the standard cross-entropy training objective (we discuss cross-entropy in the next section).

3 NER with Simple RNNs

In the first part of this assignment, we focus on using RNNs to perform named entity recognition (NER). 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`. Let the set of possible tags be denoted Y and let $\boldsymbol{y}_t$ denote the tag at position t.

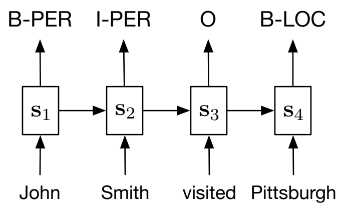

Figure 1: RNN-based NER tagging model.

To apply an RNN-based tagger to this problem, we let input vector $\boldsymbol{x}_t$ be a d-dimensional embedding of the word at position t. The word embeddings may either be pretrained or learned with the rest of the RNN parameters. To predict the tag at position t, we use a logistic regression classifier that treats $\boldsymbol{s}_t$ as an input feature vector. Specifically, to compute the probabilities of each of the possible tags we compute:

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

where $\boldsymbol{W}_o$ is a $|Y| \times d$ dimensional parameter matrix and $\boldsymbol{c}$ is a $|Y|$-dimensional parameter vector. A softmax operation maps the vector of scores to a probability distribution over tags.

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

4 Bi-directional RNNs

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

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

where $[\boldsymbol{a}; \boldsymbol{b}]$ is the concatenation of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, and $\boldsymbol{W}_c$ and $\boldsymbol{b}_c$ are parameters. The combined state $\boldsymbol{s}_t$ is used as input to the classifier (i.e. to compute $\boldsymbol{r}_t$ as above.) This architecture is depicted in the left half of Figure 2.

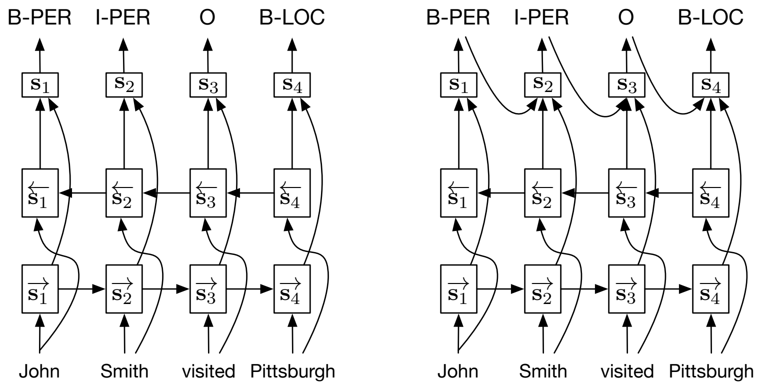
B-PER
I-PER
O
B-LOC
$\mathbf{s}_1$
$\mathbf{s}_2$
$\mathbf{s}_3$
$\mathbf{s}_4$
$\overleftarrow{\mathbf{s}}_1$
$\overleftarrow{\mathbf{s}}_2$
$\overleftarrow{\mathbf{s}}_3$
$\overleftarrow{\mathbf{s}}_4$
$\overrightarrow{\mathbf{s}}_1$
$\overrightarrow{\mathbf{s}}_2$
$\overrightarrow{\mathbf{s}}_3$
$\overrightarrow{\mathbf{s}}_4$
John
Smith
visited
Pittsburgh

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

Learning: Setting the parameters of the model to maximize conditional training likelihood is equivalent to setting the parameters to minimize the cross entropy (CE) between the model's conditional distribution p and the training data's empirical distribution $\tilde{p}$ at each time step:

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

For our NER dataset there is just one correct tag for each position of each training sentence. Thus, the empirical distribution is a pointmass at the observed tag (i.e. has probability 1 for the correct tag and 0 otherwise) and the loss at position t is given by:

$$CE_t = -\log p(\boldsymbol{y}_t^* \mid \boldsymbol{r}_t)$$

where $\boldsymbol{y}_t^*$ represents the gold tag at position t. The total training loss is the sum of the CE losses at each position of each training sentence.

Prediction. Since the model we have described so far uses independent classifiers at each time step, predicting a full sequence of tags for a new input sentence can be accomplished by independtly argmaxing over the possible tags at each position – no dynamic program is required for exact decoding.

5 Adding structure: Neural CRF

In the previous model, when we predict a tag $\boldsymbol{y}_t$ at time t, we only need the state $\boldsymbol{s}_t$, which depends only on the inputs $\boldsymbol{x}_{1:n}$. In other words, tags are conditionally independent given $\boldsymbol{x}_{1:n}$.

In this section, we add sequential structure to the model by allowing dependencies between neighboring tags $\boldsymbol{y}_t$ and $\boldsymbol{y}_{t-1}$. Here, instead of using independent locally normalized classifiers to define the conditional distribution on the tag sequence, we use a conditional random field (CRF). The neural archicture still defines the score of the tag sequence, but there are two changes: (1) the scoring function will include potentials defined over pairs of neighboring tags and (2) scores are mapped to a probability distribution by exponentiating and globally normalizing over *all* possible tag sequences.

One way to define and parameterize a sequential neural CRF is to directly add parameters for the pairwise potentials. Specifically, we let the 'emission' score of a tag y at timestep t be defined as before: in terms of the bidirectional RNN, denoted $\boldsymbol{r}_{t,y}$. However, we now define a 'transition' score between current tag y and next tag y' at timestep t, denoted $\boldsymbol{A}_{t,y,y'}$. The score and probability of a tag sequence can be written as:

$$\text{score}(\boldsymbol{y}_{1:n}, \boldsymbol{x}_{1:n}) = \sum_{t=1}^{n} \boldsymbol{r}_{t,\boldsymbol{y}_t} + \sum_{t=1}^{n-1} \boldsymbol{A}_{t,\boldsymbol{y}_t,\boldsymbol{y}_{t+1}}$$

$$p(\boldsymbol{y}_{1:n}|\boldsymbol{x}_{1:n}) = \frac{\exp(\text{score}(\boldsymbol{y}_{1:n}, \boldsymbol{x}_{1:n}))}{\sum_{\boldsymbol{y}'_{1:n}} \exp(\text{score}(\boldsymbol{y}'_{1:n}, \boldsymbol{x}_{1:n}))}$$

Here, we parameterize the transition scores $\boldsymbol{A}_{t,y,y'}$ directly: we let $\boldsymbol{A}_{t,y,y'} = \boldsymbol{U}_{y,y'}$, where $\boldsymbol{U}_{y,y'}$ is a parameter specifying the transition score of going from tag y to tag y', regardless of timestep.

Another approach is to introduce sequential dependencies by using **tag embeddings**. Here, we use our original scoring function with only 'emission' scores:

$$\text{score}(\boldsymbol{y}_{1:n}, \boldsymbol{x}_{1:n}) = \sum_{t=1}^{n} \boldsymbol{r}_{t,\boldsymbol{y}_t}$$

but will let state vector $\boldsymbol{s}_t$ (which determines $\boldsymbol{r}_t$) depend on the previous tag in a simple way. Specifically, let $\boldsymbol{e}_y$ represent the embedding vector of tag y. We redefine the combination function as:

$$\boldsymbol{s}_t = \phi\left(\boldsymbol{W}_c[\overrightarrow{\boldsymbol{s}_t}; \overleftarrow{\boldsymbol{s}_t}; \boldsymbol{e}_{y_{t-1}}] + \boldsymbol{b}_c\right)$$

We use an auxiliary START tag for y_0 as the first tag of the sequence. Using this approach, the transition matrix is implicitly encoded by the parameters of the RNN. The model is illustrated in Figure 2 on the right side.

Learning and inference: The training objective for the sequential neural CRF is the log conditional likelihood of the training data. Since the CRF is globally normalized, computing the training objective requires summing over all possible tag sequences. Because the model has sequential dependences, this in turn requires running the forward algorithm. Luckily, using modern neural toolkits (see implementation suggestions below) the gradients of the training objective can be computed automatically. Similarly, to predict the most likely sequence of tags, we must use the Viterbi algorithm.

Deliverable 1: Implement code to train a sequential neural CRF using a bi-directional RNN paramterization on `train.data` to perform NER. You may choose from either of the described approaches, or your own approach. `dev.data` should only be used to tune the hyper-parameters. Submit (1) the code and (2) the tagging output on `dev.data` with the naming convention `<andrewid>-dev.data.out`

Deliverable 2: Submit the curves with training and validation likelihood on the y axis vs the number of epochs during training. Optimize for as many epochs[2] possible before your model starts over-fitting on `train.data`.

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

Some potentially useful information: Training can be slow especially without a GPU and lack of batching in the input data. We ran our benchmarks on a CPU machine with a batch size of one. Our implementation takes around 20 minutes per epoch and achieves the best dev score after 3-4 epochs. For reference, on the full training data our system achieves an F1 of around 80 on `dev.data`. As it takes around 1.5 hours to judge the quality of your model, we suggest using half or quarter of the data during development. For reference, our system achieves a dev F1 around 75 when trained on half of the data, and 55 when trained on a quarter. Your final submission should be trained on the complete training data.

6 Bring your own Task (BYOT)

This part of the assignment involves defining your own structured-prediction task and designing a neural CRF to attack it. We have provided an NLP dataset that has multiple tasks associated with it. You may choose to do any one of these tasks, or pick something entirely different so long as your model incorporates some kind of structured dependency (e.g. sequential structure, semi-Markov structure, tree-structure, etc.).

Suggested Tasks and Dataset: The dataset is sampled from OntoNotes (Release 5.0) which is a collection of newswire (News), broadcast news (BN), broadcast conversation (BC), telephone conversation (Tele) and web data (Web). The dataset contains annotation supporting several possible structured NLP tasks:

- Coreference resolution - Natural language contains pronouns and nouns which corefer to entities in the world. OntoNotes contains annotation that connects coreferent mentions. Predicting these connections from text can be formulated as a sequence prediction task. Can you extend your sequential neural CRF to attack coref?
- Syntatic Parsing - OntoNotes contains constituency parse annotation, another great structured prediction task. The bidirectional RNN archiecture may already be good at characterizing constituents. But can you modify your neural CRF to capture tree-structure?
- NP Chunking - Instead of full parsing, another standard NLP task is to try to identify the segments of a sentence that are noun phrases. Can you accomplish this with your sequential neural CRF? Would it be better to add semi-Markov structure?
- Word Sense Disambiguation (WSD)- Words often have multiple senses. Without understanding their context, it can be difficult to diasmbiguate which sense was intended. However, neighboring words may themselves be ambiguous. One way to attack this problem is to predict word senses jointly and let neighboring senses depend on one another. Can you extend your sequential neural CRF to autmoatically accomplish WSD?
- Named Entity Recognition (NER) - You have already attacked NER using a sequential neural CRF. However, in many ways it makes more sense to treat NER as a joint chunking and tagging problem: segment the text into chunks that correspond to entities, then tag each chunk with a type. Can you modify your implementation to treat NER as a segmentation problem? In other words, can you change your model to incorporate semi-Markov structure instead of just Markov structure?

Deliverable 3: Implement code to train for the task you choose and submit it as a part of the zip file.

Deliverable 4: Write a short report (no more than 2 pages in NIPS format) describing (1) the task (2) the model (3) Training Curves and (4) a brief discussion of your findings: did your approach work?

7 Implementation recommendation

You can use whatever you like to implement this assignment. However, the TAs will be best able to help if you use either PyTorch or Tensorflow. We recommend implementing the neural CRF by coding up the computation of training likelihood (which involves the forward algorithm, but no backward pass) as a computation graph. Then, training can be accomplished via backprop. If you prefer, you can ofcourse compute the gradients yourself – but the neural parameterization may make this difficult. Further, one of the PyTorch tutorials is a sequential neural CRF (see the link below). You are free to refer to this tutorial for reference, but your implmentation must contain your own code.

Suggested Readings: Neural-CRF paper-`https://arxiv.org/pdf/1508.01991.pdf`, Practical Tutorial-`http://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html`

8 Submission instructions

Upload and submit your work on canvas in a .zip file with the following folder structure. *Please follow the naming conventions and folder structures, as we use scripts to evaluate the submissions.*

```
<andrewid>
    |---- NER
          |---- curves.pdf
          |---- <andrewid>-dev.data.out
          |---- code
    |---- BYOT
          |---- <andrewid>-report.pdf
          |---- code
```