CS4248
AY 2019/20 Semester 1
Programming Assignment 2

Due Date

Friday 8 November 2019 at 9am. Late assignments will not be accepted and will receive ZERO mark.

Objective

In this assignment, you are to write a program in Python 3.5 to perform part-of-speech (POS) tagging, similar to programming assignment 1 (PA1). However, in programming assignment 2 (PA2), you will implement your POS tagger using neural network methods and the PyTorch open source library. It will be a good exercise for you to learn how to use PyTorch to implement a deep learning approach, and to compare it with the HMM approach in PA1 on the same POS tagging task.

Approach

In this assignment, you will use a convolutional neural network (CNN) and a bi-directional long short-term memory (LSTM) to implement your POS tagger. First, a character-level CNN takes character embeddings of the characters of each word to construct a vector representation of the word. The CNN word representation is then concatenated with a separate word embedding to form the input word representation of each word. A bi-directional LSTM then constructs the output word vector representation of each word, which is then fed through a linear projection and the softmax function to produce a probability distribution over the different POS tags of a word. Cross-entropy loss function is used as the objective function for training. Dropout regularization is used during training to prevent overfitting.

A more detailed description of the approach follows.

Word representation

Each word w_i is represented by an embedding vector $\mathbf{e}_i = \mathbf{E}_w[\text{idx}(w_i)] \in \mathbb{R}^{d_{emb}}$, obtained from a lookup table (embedding matrix) $\mathbf{E}_w \in \mathbb{R}^{|V| \times d_{emb}}$, in which d_{emb} is the word vector dimension, V is the vocabulary, and the operator $\text{idx}(w)$ maps a word $w \in V$ to a unique integer index, which corresponds to the row in $\mathbf{E}_w$ that contains the embedding vector for w.

In addition, the word also contains the representation of its character sequence. Each character $c_{i,j}$ of word w_i is represented by a character embedding $\boldsymbol{\chi}_{i,j} = \mathbf{E}_c[\text{charidx}(c_{i,j})] \in \mathbb{R}^{d_{char}}$, also obtained from an embedding matrix $\mathbf{E}_c \in \mathbb{R}^{|C| \times d_{char}}$, where d_{char} denotes the character embedding dimension size, C is the set of characters, and $\text{charidx}(c)$ maps a character $c \in C$ to a unique integer index corresponding to a row in $\mathbf{E}_c$.

These character embeddings are fed into a convolutional neural network (CNN). The CNN has a 1D convolution that moves a sliding window of size κ over the character sequence, applying ℓ different convolutional filters to each window in the character sequence. Max pooling is used in the CNN.

Character-Level Convolutional Neural Network

For any given $c_{i,j}$, the input to the CNN, that is $\overline{\boldsymbol{\chi}}_{i,j}$, is the concatenation of the character embeddings of the character and its $(\kappa - 1)$ surrounding characters (left and right):

$$\begin{aligned}\overline{\boldsymbol{\chi}}_{i,j} &= \left[\boldsymbol{\chi}_{i,j-(\kappa-1)/2}; \ldots; \boldsymbol{\chi}_{i,j+(\kappa-1)/2}\right] \\ &= \bigoplus \left(\boldsymbol{\chi}_{i,j-(\kappa-1)/2:j+(\kappa-1)/2}\right) \in \mathbb{R}^{\kappa \cdot d_{char}}\end{aligned}$$

ℓ different convolutional filters, $\mathbf{u}_1, \ldots, \mathbf{u}_\ell$, are arranged into a matrix $\mathbf{U}$ and together with a bias vector $\mathbf{b}_{conv}$, the convolution operation is defined as follows:

$$\boldsymbol{\phi}_{i,j} = g\left(\overline{\boldsymbol{\chi}}_{i,j} \cdot \mathbf{U} + \mathbf{b}_{conv}\right)$$
$$\boldsymbol{\phi}_{i,j} \in \mathbb{R}^{\ell} \; ; \; \overline{\boldsymbol{\chi}}_{i,j} \in \mathbb{R}^{\kappa \cdot d_{char}} \; ; \; \mathbf{U} \in \mathbb{R}^{\kappa \cdot d_{char} \times \ell} \; ; \; \mathbf{b}_{conv} \in \mathbb{R}^{\ell}$$

Through the character convolution for word w_i, we have vectors $\boldsymbol{\phi}_{i,1}, \ldots, \boldsymbol{\phi}_{i,|w_i|}$. These vectors are combined or pooled into a single vector $\mathbf{c}_i \in \mathbb{R}^l$, representing the character sequence of word w_i. We use the max-pooling operation, where we assign the k-th dimension of the vector $\mathbf{c}_i$ by the maximum value of the k-th dimension among the vectors $\boldsymbol{\phi}_{i,1}, \ldots, \boldsymbol{\phi}_{i,|w_i|}$ as follows:

$$\mathbf{c}_i[k] = \max_{1 \le j \le |w_i|} \boldsymbol{\phi}_{i,j}[k] \quad \forall k \in \{1, \ldots, \ell\}$$

Final Representation of Words

A word w_i is represented by its embedding vector $\mathbf{e}_i$ and the character level representation $\mathbf{c}_i$, therefore giving the input representation of w_i as

$$\mathbf{x}_i = [\mathbf{e}_i; \mathbf{c}_i] \in \mathbb{R}^{d_x}$$

where $d_x = d_{emb} + l$.

Modeling Word Sequence by Long Short-Term Memory (LSTM)

We use a bidirectional LSTM where the word representation vectors $\mathbf{x}_i, \forall i \in \{1, \ldots, N\}$ are fed sequentially to the forward LSTM from left to right and to the backward LSTM from right to left. For each time step i corresponding to the position of a word in the sentence, the forward LSTM produces a hidden representation vector $\vec{\mathbf{h}}_i \in \mathbb{R}^{d_h}$, which is fed to the next time step:

$$\vec{\mathbf{h}}_i = \overrightarrow{\text{LSTM}}\left(\mathbf{x}_i, \vec{\mathbf{h}}_{i-1}\right)$$

while the backward LSTM produces $\overleftarrow{\mathbf{h}}_i \in \mathbb{R}^{d_h}$, fed to the previous time step:

$$\overleftarrow{\mathbf{h}}_i = \overleftarrow{\text{LSTM}}(\mathbf{x}_i, \overleftarrow{\mathbf{h}}_{i+1})$$

with d_h denoting the dimension of the LSTM hidden representation vector.

The hidden representation of each word w_i is the concatenation of both forward and backward LSTM hidden representation vectors, formulated as

$$\mathbf{h}_i = [\vec{\mathbf{h}}_i; \overleftarrow{\mathbf{h}}_i] \in \mathbb{R}^{2d_h}$$

Transforming LSTM Hidden Representation into POS Tag Probability

There are 45 Penn Treebank POS tags, so we need to project the hidden vector of the LSTM from $2d_h$ dimensions to d_t $(= 45)$ dimensions corresponding to the size of the tag set, via linear projection:

$$\mathbf{s}_i = \mathbf{h}_i \cdot \mathbf{W}_t + \mathbf{b}_t$$

where $\mathbf{W}_t \in \mathbb{R}^{2d_h \times d_t}$ and $\mathbf{b}_t \in \mathbb{R}^{d_t}$.

The probability of a POS tag t being the tag at position i, namely $t_i = t$ is then given as:

$$p(t_i = t | w_1, \dots, w_N) = \frac{\exp(s_i[\text{tagidx}(t)])}{\sum_{m=1}^{d_t} \exp(s_i[m])}$$

where the function $\text{tagidx}(t)$ maps a POS tag t to an integer index within $\{1, \dots, d_t\}$.

Training and Testing

You are provided with the same training set and test set as in PA1. The same commands will be used to train and test your POS tagger, as follows:

```
python3.5 buildtagger.py sents.train model-file

python3.5 runtagger.py sents.test model-file sents.out
```

Deliverables

As in PA1, the commands `buildtagger.py` and `runtagger.py` as shown above will be executed to evaluate your POS tagger. Grading will be done after the submission deadline, by testing your POS tagger on a set of new, blind test sentences.

You will need to submit your files `buildtagger.py` and `runtagger.py` via CodeCrunch. Please do not change the file names and do not submit any files other than `buildtagger.py` and `runtagger.py`. Use the skeleton code for `buildtagger.py` and `runtagger.py` released to you in this assignment to add your code.

The URL of CodeCrunch is as follows:

https://codecrunch.comp.nus.edu.sg/index.php

Login to CodeCrunch with your NUSNET ID and choose the 'CS4248 PA2' task under the CS4248 module. Submit your two Python files there.

After submission, you can find the accuracy of your POS tagger on the 'Test Output' tab after execution is completed. The test file used in CodeCrunch is `sents.test`. We will run your code on the SoC cluster nodes xgpd0 to xgpd9. You can test your code on these nodes before submitting in CodeCrunch. You can login to these cluster nodes using your SoC Unix ID. Details of these computing nodes can be found at the following URL:

https://dochub.comp.nus.edu.sg/cf/guides/compute-cluster/hardware

Some Points to Note:

1. Your `buildtagger.py` should not take more than 10 minutes to complete execution on CodeCrunch, and `runtagger.py` 2 minutes. Your code will be terminated by CodeCrunch if it takes more than the allocated time to execute.

2. Arguments to the Python files are absolute paths, not relative paths. They will be passed as arguments to the Python files, so please do not hard code the paths in your code.

3. We will use python3.5 to run your code. As such, please only use python3.5 when testing your code. The version of PyTorch to use in this assignment is 0.4.1.

Grading

The marks awarded in this assignment will be based on the accuracy of your POS tagger on a blind test set of sentences.