Assignment 3: Deep Learning for NLP

CSC544 Spring 2020

Ungraded

1 Introduction

The goal of this coding assignment to get expertise in PyTorch, especially in developing code from the ground up. You will be implementing a sequence-to-sequence Recursive Neural Network (RNN) model for Part-of-Speech tagging. In particular, you are expected to:

1. Populate the starter code in designated places marked by TODO(student).
   (a) The starter code is on Blackboard in a file called `starter.py`.
2. Write code for reading the data files and producing numpy arrays that will be used for training. This should be implemented in class DatasetReader.
3. Write code for constructing and training the model [here, you should be creative!]. This should be implemented in class SequenceModel. The model needs to output expected outcomes (i.e. predicted tags).

   You must implement the functions that are annotated in the starter code. You should explore good hyperparameters [e.g. batch size, learning rate], which are always a function of your model architecture and the training algorithm (there is no one answer that fits all!).
4. Train models on the Italian dataset provided.

2 Installing PyTorch

To install PyTorch locally via `Pip`, do the following steps (we assume you do not have access to a GPU. You will not need access to a GPU for this assignment.)

Windows or Linux

```
$ pip install torch==1.4.0+cpu torchvision==0.5.0+cpu
-f https://download.pytorch.org/whl/torch_stable.html
```

or

```
$ conda install pytorch torchvision cpuonly -c pytorch
```

Mac

```
$ pip install torch torchvision
```

or

`$ conda install pytorch torchvision -c pytorch`

Start by trying to install via `Pip`, then try `Conda`. For other installation options, please see `https://pytorch.org/`.

3 Task 1: Data Processing

The data you will be using is Italian and the format is word/part-of-speech-tag but the part-of-speech tags are different than the English-specific Penn Tree-Bank. The codes are defined here: https://universaldependencies.org/tagset-conversion/it-isdt-uposf.html and the referenced Universal POS tags are here: https://universaldependencies.org/docsv1/u/pos/all.html

In class, we talked about vector-based representations of words. Here we will use what is called a one-hot vector encoding for words and part of speech tags. The functions below which you will implement will convert the word and part of speech tag strings into these encodings.

First you will implement ReadFile. This function is used by ReadData. The function header is copied here for your convenience:

```
def ReadFile(filename, term_index, tag_index):
  """Reads file into dataset, while populating term_index and
     tag_index.

  Args:
    filename: Path of text file containing sentences and tags.
      Each line is a sentence and each term is followed by "/tag".
      Note: some terms might have a "/" e.g. my/word/tag -- the
      term is "my/word" and the last "/" separates the tag.
    term_index: dictionary to be populated with every unique
      term (i.e. before the last "/") to point to an integer.
      All integers must be utilized from 2 to number of unique
      terms + 1, without any gaps nor repetitions. 1 is reserved
      for unknown words.
    tag_index: same idea as term_index, but for tags. Start at
      index 1. Assume no unknown tags in testing.

  Return:
    The parsed file as a list of lists: [parsedLine1,
    parsedLine2,...]
    each parsedLine is a list: [(term1, tag1), (term2, tag2),...]
"""
```

Implement BuildMatrices. The function header is copied here for your convenience:

```
def BuildMatrices(dataset):
  """Converts dataset [returned by ReadFile] into numpy arrays
  for tags, terms, and lengths.

  Args:
    dataset: Returned by method ReadFile. It is a list (length N)
      of lists:
```

```
      [sentence1, sentence2, ...], where every sentence is a list:
      [(word1, tag1), (word2, tag2), ...], where every word and
      tag are integers.

  T is the maximum length. You can define this as a global variable
                                    . You will use it when you
                                    create a SequenceModel class
                                    later.
  Returns:
    Tuple of 3 numpy arrays: (terms_matrix, tags_matrix,
      lengths_arr)
      * terms_matrix: shape (N, T) int64 numpy array. Row i
                                        contains
        the word indices in dataset[i].
      * tags_matrix: shape (N, T) int64 numpy array. Row i contains
        the tag indices in dataset[i].
      * lengths: shape (N) int64 numpy array. Entry i contains the
        length of sentence in dataset[i].

  For example, calling as:
    In[]: BuildMatrices([
      [(1,2), (4,10)], [(13, 20), (3, 6), (7, 8), (3, 20)]])
  should return the tuple:
    Out[]: (
      [[1, 4, 0, 0], [13, 3, 7, 3]],    # Note: 0 padding.
      [[2, 10, 0, 0], [20, 6, 8, 20]],
     [2, 4]
    )
  """
```

4 Task 2: Code a Basic RNN Model

In this task, you are expected to populate the class SequenceModel in `starter.py`. The basic model we are attempting is an RNN, shown in Figure 1.

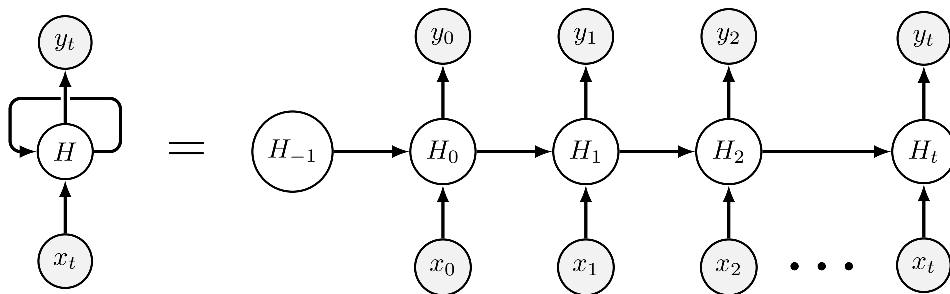

Figure 1: RNN abstract model.

RNN's take one word of input at a time (x_i) to predict an output (y_i). They do this by combining word x_i with the hidden state for the last step, h_{i-1}, and then using the new hidden state h_i to predict y_i. We'll show you the vectors you need to program to implement this:

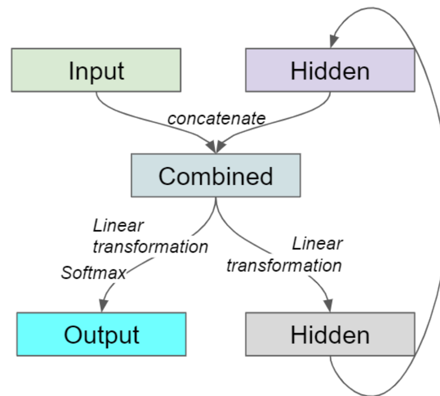

Figure 2: A simple RNN diagram.

1. **Input:** This is the one-hot encoding of each word in your input sentence, produced in the previous part of this homework. (This corresponds to x_i in Figure 1).

2. **Hidden (purple)**: This is the hidden state of the RNN. Before you start with any input, you can randomly initialize the hidden state. But for the second input onwards, you should take the bottom hidden state (gray) and set it equal to the top (purple).

3. **Combined**: This is a concantenation of the input vector and the hidden state from the last step.

4. **Output**: This is the output tag you'll predict. You'll get here after applying a linear tranformation to the hidden layer and a softmax.

5. **Hidden (gray)**: The second hidden layer is produced by applying a linear transformation to the Combined layer.

You might find the following functions helpful:

- `nn.Linear`: create a linear mapping layer. (Hint: useful for producing **Hidden (gray)** and **Output)**.

- `torch.cat`: concatenates two vectors in PyTorch. (Hint: useful for producing **Combined**).

- `nn.LogSoftmax`: Performs the softmax operation (Hint: useful for producing **Output**).

5 Training and testing your model

All of the required methods in `starter.py`:

1. `save_model`: Saves the trained model to a file.
2. `load_model`: Loads the trained model from a file.
3. `train`: Given sentences (i.e. matrices of term IDs and their lengths) and part-of-speech tag IDs for each word, train the model.
4. `evaluate`: Given sentences (i.e. matrices of term IDs and their lengths), return the part-of-speech tag ID for every word in every sentence.

```
# TRAINING PROGRAM
model = SequenceModel(num_terms, num_tags, max_length)
model.build()
while (time_spent < K):
    train(model, terms, tags, lengths)
    model.save_model("/some/file/path")

# TESTING PROGRAM
model = SequenceModel(num_terms, num_tags, max_length)
model.load_model("/some/file/path")
model.build()

predicted_tags = evaluate(model, test_terms, test_lengths)
# and compare with ground-truth
```

Below are some tips for improving your accuracy.

6 Tip 1. Code an RNN Model with a pretrained embedding layer.

Embedding layers are a fundamental part of neural network models. In this section we will give you hints on how to utilize embeddings and pretrained embeddings to improve your accuracy.

NOTE: You *may* chose to implement these techniques if you are not getting high accuracy. However, it is not required. If you are comfortable with the accuracy you are getting, **you may chose not to use embeddings.**

As you can see in Figure 3, we make a slight modification from the previous diagram, we add an embeddings layer after the input. The **Embedding** layer is a layer of your choice of size (usually 50, 100 or 300). It translates the one-hot vector for each word into a dense representation. What's great about embedding layers is that they can be used with *pretrained embeddings* to allow you to leverage work that others have done to train embeddings on other, usually more general tasks. Word2Vec[1] and Glove[2] are good embeddings but many other embeddings, trained in different ways, are available in different languages as well [3].

[1] For English: https://github.com/mmihaltz/word2vec-GoogleNews-vectors
[2] For English `https://nlp.stanford.edu/projects/glove/`
[3] http://vectors.nlpl.eu/repository/

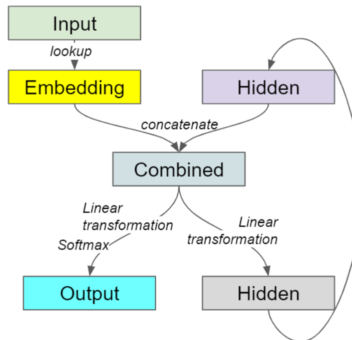

Figure 3: An RNN with embeddings diagram.

Note: While using embeddings layers and pretrained layers are NOT required for this assignment, they are tricks you can try to improve your accuracy.

You might find these functions useful:

1. `nn.Embedding`: Takes two parameters: `num_embeddings` and `embedding_dim` to specify the size of the embeddings layer.

2. Load in fixed, pretrained embeddings:

```
emb_layer = nn.Embedding(num_embeddings, embedding_dim)
emb_layer.load_state_dict({'weight': weights_matrix})
emb_layer.weight.requires_grad = False
```

7 Some more tips:

1. Try removing less frequent words and replacing them with out-of-vocabulary for increased performance.

2. Make sure you open the files with `open('<filename.txt>', encoding='utf-8')` to read in the non-ASCII characters.

3. Try adding an embedding layer. Additionally, try using pretrained embeddings, which for different languages can be found here: `https://wikipedia2vec.github.io/wikipedia2vec/pretrained/`.

4. Try experimenting with different architectures: hidden unit size, number of layers, connections between layers.

5. Compare different hyperparameter/learning configuration variants: learning rates, gradient descent variants and schedules, etc.

6. Try experimenting with different inputs: ablate the feature sets (i.e. the input vocabulary), introduce new (meaningful) features.

8 Command Line Interface

`starter.py` has the following CLI built into it:

```
$ python .starter.py -h

Train RNN.

optional arguments:
  -h, --help  show this help message and exit
  -i I        training file path.
  -m M        Model output path.
  -o O        Predictions output path.
  -t T        Length of time training (seconds).
  -e E        Num epochs.
```