CS 6320: Natural Language Processing

Homework 2: Part-of-Speech Tagging (100 points)

Due: March 24 2023, 2:00 pm

For this homework, you will design and implement a part of speech tagger (POST) using two statistical models: (1) a Hidden Markov Model (HMM); and (2) a Recurrent Neural Network (RNN).

Data

To train and evaluate your POST, you will use a modified Brown corpus [1]. To simplify the problem, we will use a tagset that contains 11 tags, i.e. Noun, Pronoun, Verb, Adjective, Adverb, Conjunction, Preposition, Determiner, Number, Punctuation and Other (indicated as X). The corpus has been provided via **train.zip**.

Part - 1: POST using HMMs (45 points)

To train the Hidden Markov Model, you will design and implement a class 'HMMTagger' that supports the following methods:

1. `load_corpus(path)`: This function loads the corpus at the given path and returns it as a list of POS-tagged sentences. Each line in the files should be treated as a separate sentence, where sentences consist of sequences of whitespace-separated strings taking the form "token/POS". Your method should return a list of lists, with individual entries being 2-tuples of the form (token, POS).

   As an example, consider the input sentence given below:

   `B./NOUN J./NOUN Connell/NOUN is/VERB the/DETERMINER present/ADJECTIVE treasurer/NOUN and/CONJUNCTION manager/NOUN ./PUNCT`

   Your method should return the output as: `[('B.', 'NOUN'), ('J.', 'NOUN'), ('Connell', 'NOUN'), ('is', 'VERB'), ('the', 'DET') ...]`

2. `initialize_probabilities(sentences)`: This method takes as input the list of sentences generated by `load_corpus()` and initializes all variables required by the tagger. In particular, if $\{t_1, t_2, ...t_n\}$ denotes the set of tags and $\{w_1, w_2, ...w_m\}$ refers to the set of tokens found in the input sentences, you should compute the following:

   (a) The initial tag probabilities $\pi(t_i)$ for $1 \leq i \leq n$, where $\pi(t_i)$ is the probability that the POS tag of the first word of a sentence is t_i.

   (b) The transition probabilities $a(t_i \longrightarrow t_j)$ for $1 \leq i, j \leq n$, where $a(t_i \longrightarrow t_j)$ is the probability that tag t_j occurs after tag t_i.

   (c) The emission probabilities $b(t_i \longrightarrow w_j)$ for $1 \leq i, j \leq m$, where $b(t_i \longrightarrow w_j)$ is the probability that token w_j is generated, given tag t_i.

   Note that you will have to use add-one smoothing to ensure that the tagger works with new inputs.

3. `viterbi_decode(sentence)`: This method returns the most likely tag sequence for an input sentence using the Viterbi algorithm. For example, consider the input sentence shown below:

   sentence = 'People race tomorrow .'

   Your method should return the output as ['NOUN', 'VERB', 'NOUN', 'PUNCTUATION']. Note that this is just an example and your answer may not necessarily match the output shown here.

A sample skeleton file **tagger.py** has been provided as a reference to this homework. If you are using a programming language other than Python, please use the same class definition(s) for that language. Essentially, we should be able to import your class as a package and use these methods directly.

Part- 2: POST using RNNs (45 points)

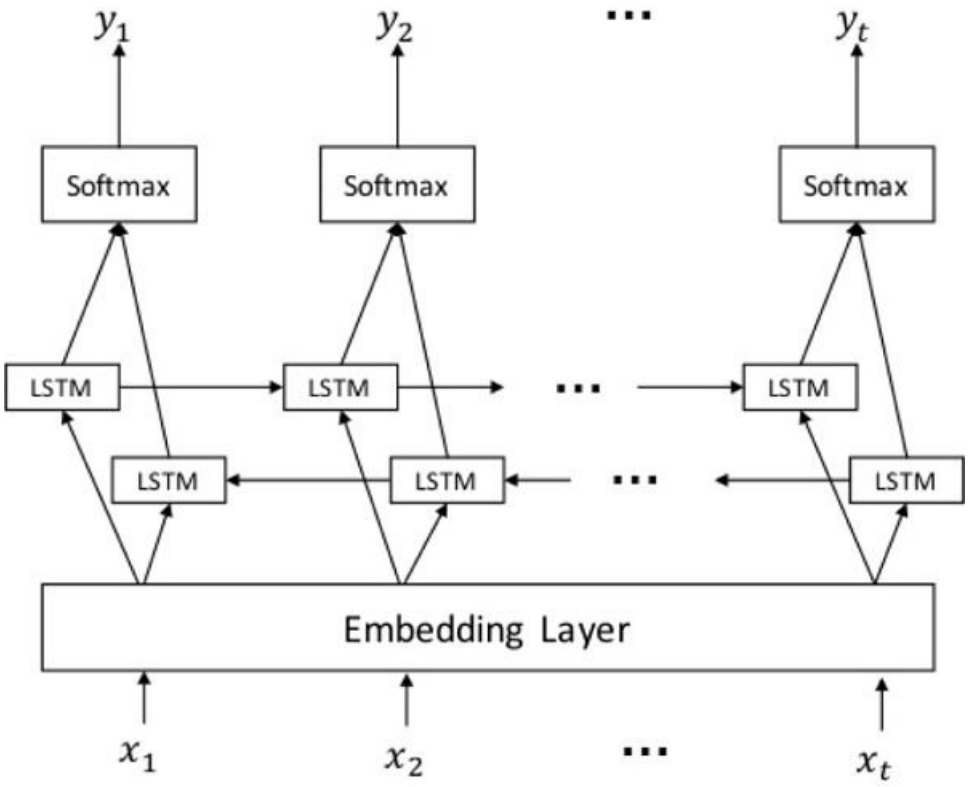

Figure 1: POST using RNNs

Now, we will train a simple RNN architecture to train our POST.

For implementation, you will use keras, a very easy-to-use deep learning library with Python. There are other libraries available too, like TensorFlow, PyTorch, Caffee, Dynet, etc. : feel free to use those libraries if you are more comfortable working with them. However, the TA will be able to provide support and help only for keras, TensorFlow and PyTorch.

Additionally, you will write all your code in Google colab: a free notebook server provided by Google to run your Python code. Colab comes bundled with a GPU so you can use it to make your code run fast (faster than a CPU or even the csgrads1 server).

Specifically, you will perform the following steps to implement your POST.

1. `load_corpus(path)`: This function loads the corpus at the given path and returns it as a list of POS-tagged sentences. Note that this function is exactly the same as the function we defined for 'HMMTagger' and is only repeated here for the sake of clarity.

2. `create_dataset(sentence)`: This function takes the list of sentences generated by `load_corpus()` and constructs two numpy arrays that can be fed to the RNN.

   Any deep learning library works with numbers, not words or tags. Thus, we assign each word and tag a unique integer. This can be done by identifying all unique words and tags and indexing them into a dictionary. Note that while defining your word dictionary, you will define or add one special token to the word list: [PAD] (more on that later).

   Using these dictionaries, define two lists of lists: `train_X` and `train_y`. `train_X` will contain lists of integers, where each word in a sentence will be converted into its unique integer. `train_y` will contain the lists of associated tags, where each tag is likewise, represented by its unique integer. Before returning these lists, convert them to numpy arrays.

3. `pad_sequences(train_X, train_y)`: Deep Learning libraries can deal only with sequences of fixed lengths i.e. all sentences must have same number of words. To achieve this, you will add [PAD] tokens to each list in `train_X` so that all lists in `train_X` have the same length. Likewise, we will pad zeros to each list in `train_y` to ensure we do not have a token-tag length mismatch. To pad sequences, you can use Keras's convenient `pad_sequences()` utility function or you may write your own function.

4. `define_model()`: Lets define our model. The model architecture is shown in Figure 1. Here's what comprises our model:

(a) We'll need an embedding layer that computes a word vector model for our words. Recall the word2vec and GloVe models taught in class and how they embed words as vectors. This can be done using Keras's `Embedding` layer. Set the size of the vectors to 128.

(b) We'll need an LSTM layer with a Bidirectional modifier. Use Keras's `LSTM` layer to add an LSTM to the model. Note that you must set the `return_sequences` parameter so that the LSTM outputs a sequence, not only the final value. Set the number of hidden layers to 256.

(c) After the LSTM Layer we need a fully-connected layer (known as `Dense` in Keras) that identifies the correct POS tag. Since this layer needs to run on each element of the sequence, we need to add the `TimeDistributed` modifier before the `Dense` layer.

You may use `model.summary()` to get a quick peek at the model's architecture and make sure its correct before you start training it.

5. `to_categorical(train_y, num_tags = 11)`: There's one more thing to do before training. We need to transform the sequences of tags to sequences of One-Hot Encoded tags as this is what the fully connected layer outputs. Define `to_categorical()` method to carry out this task.

6. `train(model, train_X, train_y)`: To train the model, use Keras's `fit()` method. Set the batch size to 128, no. of iterations to 40 and validation_split to 0.2.

7. `test(model, sentence)`: To evaluate the model, give a sentence i.e. list of words as input the model and predict its POS tags. Please note that you will have to represent this list of words as a list of unique integers (the same way you did for training) before running your model on it. Also note that your model outputs a list of logits, which must be converted to tags i.e. define a reverse of `to_categorical()` to get the actual tags.

A template google colab notebook is available here: Link to colab notebook. Copy this notebook into your Google drive and fill in the template to execute the program.

Please note that in both cases, the definitions of the functions are not rigid. Feel free to edit the parameters of the functions; and define your own functions and variables as you see fit.

Part - 3: Evaluation and Submission (10 points)

For evaluation, report the POS tags predicted by both models on the sentences given below:

- the planet jupiter and its moons are in effect a mini solar system .
- computers process programs accurately .

Submit the following bundled into a single zip file via eLearning:

1. Your code file(s)
2. A Readme giving clear and precise instructions on how to run the code.
3. A plaintext file showing the output of your code for the two sentences under both settings i.e. HMM and RNN.
4. A short report describing your results and what lessons you learned.

References

[1] Robert Burchfield. Frequency analysis of english usage: Lexicon and grammar. by w. nelson francis and henry kučera with the assistance of andrew w. mackie. boston: Houghton mifflin. 1982. x+ 561. *Journal of English Linguistics*, 18(1):64–70, 1985.

Useful Links

Check out these additional blogs to help you write your code:

1. https://nlpforhackers.io/lstm-pos-tagger-keras/
2. https://becominghuman.ai/part-of-speech-tagging-tutorial-with-the-keras-deep-learning-library-d7f93fa05537
3. https://www.tensorflow.org/guide/keras/rnn
4. https://blog.floydhub.com/a-beginners-guide-on-recurrent-neural-networks-with-pytorch/