Homework 3

- **Due** Jun 26, 2019 by 11:59pm

- **Points** 100

- **Submitting** a file upload
- **File Types** zip, tgz, and tar.gz

- **Available** until Jun 29, 2019 at 11:59am

This assignment was locked Jun 29, 2019 at 11:59am.

Natural Language Processing - Homework 3

Due: 11:59pm on Monday June 24th Wednesday June 26th
Total Points: 100

You are welcome to discuss the problems with other students but you must turn in your own work. Please review the academic honesty policy for this course (at the end of the syllabus page).

Create a .zip or .tgz archive containing any of the files you submit. Upload that single file to Courseworks.
The file you submit should use the following naming convention:

YOURUNI_homework3.[zip | tgz]. For example, my uni is db2711, so my file should be named db2711_homework3.zip or db2711_homework3.tgz.

Programming Component – Neural Network Dependency Parsing (100 pts)

The instructions below are fairly specific and it is okay to deviate from implementation details. **However: You will be graded based on the functionality of each function. Make sure the function signatures (function names, parameter and return types/data structures) match exactly the description in this assignment.**

Please make sure you are developing and running your code using Python 3.

Introduction

In this assignment you will train a feed-forward neural network to predict the transitions of an arc-standard dependency parser. The input to this network will be a representation of the current state (including words on the stack and buffer). The output will be a transition (shift, left_arc, right_arc), together with a dependency relation label.
Most of the parser is provided, but you will need to implement the input representation for the neural net, decode the output of the network, and also specify the network architecture and train the model.

As background reading for this project, you should consider taking a look at the following paper:

Chen, D., & Manning, C. (2014). [A fast and accurate dependency parser using neural networks (Links to an external site.)](#). In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 740-750).

Prequisites: Installing necessary packages

You will use the Keras package to construct the neural net. Keras is a high-level Python API that allows you to easily construct, train, and apply neural networks. However, Keras is not a neural network library itself and depends on one of several neural network backends. We will use the Tensorflow backend. TensorFlow is an open-source library for neural networks (and other mathematical models based on sequences of matrix and tensor computations), originally developed by Google.

Keras uses numpy data structures. You should already have numpy installed from homework 3.

Installing TensorFlow and Keras:

We suggest that you use the Python package management system pip.
On most systems, the following commands will work:

```
$ pip install tensorflow

$ pip install keras
```

Note that this will install the CPU version of TensorFlow that does not use the GPU to speed up neural network training. For this assignment, training on the CPU will be sufficient, but if your computer has a GPU (or you want to try running the assignment in the cloud), follow the installation instructions on the tensorflow page.
You could also try

```
$ pip install tensorflow-gpu
```

but **only if your system has an nvidia GPU**.

If you get stuck during the installation, you can find installation instructions for each package here:

Tensorflow: https://www.tensorflow.org/install/ (Links to an external site.)
Keras: https://keras.io/#installation (Links to an external site.)

Testing your Setup:
To test your setup, run a Python interpreter and type the following:

```
$ python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
RuntimeWarning: compiletime version 3.5 of module 'tensorflow.python.framework.fast_tensor_util' does not match runtime version 3.6
 return f(*args, **kwds)

>>> import keras
Using TensorFlow backend.
>>>
```

You can ignore any warnings printed by TensorFlow.

Getting Started

Please download the files and scaffolding needed for the dependency parser here hw3_files.zip (~7MB). The archive contains the following python scripts:

- conll_reader.py - data structures to represent a dependency tree, as well as functionality to read and write trees in the CoNLL-X format (explained below).
- get_vocab.py - extract a set of words and POS tags that appear in the training data. This is necessary to format the input to the neural net (the dimensionality of the input vectors depends on the number of words).
- extract_training_data.py - extracts two numpy matrices representing input output pairs (as described below). You will have to modify this file to change the input representation to the neural network.
- train_model.py - specify and train the neural network model. This script writes a file containing the model architecture and trained weights.
- decoder.py - uses the trained model file to parse some input. For simplicity, the input is a CoNLL-X formatted file, but the dependency structure in the file is ignored. Prints the parser output for each sentence in CoNLL-X format.
- evaluate.py - this works like decoder.py, but instead of ignoring the input dependencies it uses them to compare the parser output. Prints evaluation results.

You will only have to modify extract_training_data.py, train_model.py and decoder.py for this project.

There are also the following data files, corresponding to a standard split of the WSJ part of the Penn Treebank. The original Penn Treebank contains constituency parses, but these were converted automatically to dependencies.

- data/train.conll - Training data. ~40k sentences
- data/dev.conll - Development data. ~5k sentences. Use this to experiment and tune parameters.
- data/sec0.conll - section 0 of the Penn Treebank. ~2k sentence -- good for quick initial testing.
- data/test.conll - Test data. ~2.5k sentences. Don't touch this data until you are ready for a final test of your parser.

Dependency Format

The files are annotated using a modified CoNLL-X format (CoNLL is the conference on Computational Natural Language learning -- this format was first used for shared tasks at this conference). Each sentences corresponds to a number of lines, one per word. Sentences are separated with a blank line. You wil need to be able to read these annotations and draw

dependency trees (by hand) in order to debug your parser.

Each line contains fields, seperated by a single tab symbol. The fields are, in order, as follows:

- word ID (starting at 1)
- word form
- lemma
- universal POS tag
- corpus-specific POS tag (for our purposes the two POS annotations are always the same)
- features (unused)
- word ID of the parent word ("head"). 0 if the word is the root of the dependency tree.
- dependency relation between the parent word and this word.
- deps (unused)
- misc annotations (unused)

Any field that contains no entry is replaced with a _.

For example, consider the following sentence annotation:

```
1 The _ DT DT _ 2 dt _ _
2 cat _ NN NN _ 3 nsubj _ _
3 eats _ VB VB _ 0 root _ _
4 tasty _ JJ JJ _ 5 amod _ _
5 fish _ NN NN _ 3 dobj _ _
6 . _ . . _ 3 punct _ _
```

The annotation corresponds to te following dependency tree
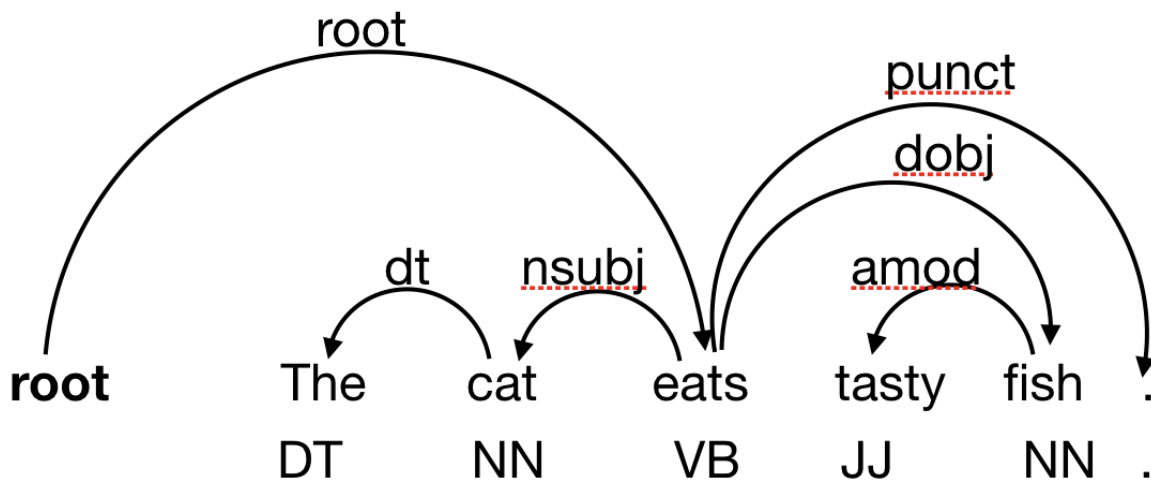
Take a look at data/sec0.conll for more examples.

The file conll_reader.py contains classes for representing dependency trees and reading in a CoNLL-X formatted data files. It is important that you understand how these data structures work, because you will use them to extract features below and also create dependency trees as parser output.

The class DependencyEdge represents a singe word and its incoming dependency edge. It includes the attribute variables id, word, pos, head, deprel. Id is just the position of the word in the sentence. Word is the word form and pos is the part of speech. Head is the id of the parent word in the tree. Deprel is the dependency label on the edge pointing to this label. Note that the information in this class is a subset of what is represented in the CoNLL format.

The class DependencyStructure represents a complete dependency parse. The attribute deprels is a dictionary that maps integer word ids to DependencyEdge instances. The attribute root contains the integer id of the root note.
The method print_conll returns a string representation for the dependency structure formatted in CoNLL format (including line breaks).

Part 1 - Obtaining the Vocabulary (0 pts)

Because we will use one-hot representations for words and POS tags, we will need to know which words appear in the data, and we will need a mapping from words to indices.

Run the following

```
$python get_vocab.py data/train.conll data/words.vocab data/pos.vocab
```

to generate an index of words and POS indices. This contains all words that appear more than once in the training data. The words file will look like this:

```
<CD> 0
<NNP> 1
<UNK> 2
<ROOT> 3
<NULL> 4
blocking 5
hurricane 6
ships 7
```

The first 5 entries are special symbols. <CD> stands for any number (anything tagged with the POS tag CD), <NNP> stands for any proper name (anything tagged with the POS tag NNP). <UNK> stands for unknown words (in the training data, any word that appears only once). <ROOT> is a special root symbol (the word associated with the word 0, which is initially placed on the stack of the dependency parser). <NULL> is used to pad context windows.

Part 2 - Extracting Input/Output matrices for training (35 pts)

To train the neural network we first need to obtain a set of input/output training pairs. More specifically, each training example should be a pair (x,y), where x is a parser state and y is the transition the parser should make in that state.

Take a look at the file extract_training_data.py
States: The input will be an instance of the class State, which represents a parser state. The attributes of this class consist of a stack, buffer, and partially built dependency structure deps. stack and buffer are lists of word ids (integers).
The top of the stack is the last word in the list stack[-1]. The next word on the buffer is also the last word in the list, buffer[-1].
Deps is a list of (parent, child, relation) triples, where parent and child are integer ids and relation is a string (the dependency label).

Transitions: The output is a pair (transition, label), where the transition can be one of "shift", "left_arc", or "right_arc" and the label is a dependency label. If the transition is "shift", the dependency label is None. Since there are 45 dependency relations (see list deps_relations), there are 45*2+1 possible outputs.

Obtaining oracle transitions and a sequence of input/output examples.
As discussed in class, we cannot observe the transitions directly from the treebank. We only see the resulting dependency structures. We therefore need to convert the trees into a sequence of (state, transition) pairs that we use for training. That part has already been implemented in the function get_training_instances(dep_structure). Given a

DependencyStructure instance, this method returns a list of (State, Transition) pairs in the format described above.

TODO: Extracting Input Representations

Your task will be to convert the input/output pairs into a representation suitable for the neural network. You will complete the method get_input_representation(self, words, pos, state) in the class FeatureExtractor. The constructor of the class FeatureExtractor takes the two vocabulary files as inputs (file objects). It then stores a word-to-index dictionary in the attribute word_vocab and POS-to-index dictionary in the attribute pos_vocab.

get_input_representation(self, words, pos, state) takes as parameters a list of words in the input sentence, a list of POS tags in the input sentence and an instance of class State. It should return an encoding of the input to the neural network, i.e. a single vector.

To represent a state, we will use the top-three words on the buffer and the next-three word on the stack, i.e. stack[-1], stack[-2], stack[-3] and buffer[-1], buffer[-2], buffer[-3]. We could use embedded representations for each word, but we would like the network to learn these representations itself. Therefore, the neural network will contain an embedding layer and the words will be represented as a one-hot representation. The actual input will be the concatenation of the one-hot vectors for each word.

This would typically require a 6x|V| vector, but fortunately the keras embedding layer will accept integer indices as input and internally convert them. We therefore just need to return a vector (a 1-dimensional numpy array) of length 6.

So for example, if the next words on the buffer is "dog eats a" and the top word on the stack is "the", the return value should be a numpy array numpy.array([4047, 4, 4, 8346, 8995, 14774]) where 4 is the index for the <NULL> symbol and 8346, 8995, 14774 are the indices for "dog", "eats" and "a".

Note that you need to account for the special symbols (<CD>,<NNP>,<UNK>,<ROOT>,<NULL>) in creating the input representation. Make sure you take into account states in which there are less than 3 words on the stack or buffer.

This representation is a subset of the features in the Chen & Manning (2014) paper. Feel free to experiment with the complete feature set once you got the basic version running.

TODO: Generating Input and Output matrices

Write the method get_output_representation(self, output_pair), which should take a (transition, label) pair as its parameter and return a one-hot representation of these actions. Because there are 45*2+1 = 91 possible outputs, the output should be represented as a one-hot vector of length 91.

You might want to take a look at the [keras.utils.to_categorical (Links to an external site.)](#) method to make this easier.

Saving training matrices

The neural network will take two matrices as its input, a matrix of training data (in the basic case a N x 6 matrix, where N is the number of training instances) and an output matrix (an Nx91 matrix).

The function get_training_matrices(extractor, in_file) will take a FeatureExtractor instance and a file object (a CoNLL formatted file) as its input. It will then extract state-transition sequences and call your input and output representation methods on each to obtain input and output vectors. Finally it will assemble the matrices and return them.

The main program in extrac_training_data.py calls get_training_matrices to obtain the matrices and then writes them to two binary files (encoded in the numpy array binary format). You can call it like this:

python extract_training_data.py data/train.conll data/input_train.npy data/target_train.npy

You can also obtain matrices for the development set, which is useful to tune network parameters.

python extract_training_data.py data/dev.conll data/input_dev.npy data/target_dev.npy

Part 3 - Designing and Training the network (30 pts)

TODO: Network topology Now that we have training data, we can build the actual neural net. In the file train_model.py, write the function build_model(word_types, pos_types, outputs). word_types is the number of possible words, pos_types is the number of possible POS, and outputs is the size of the output vector.

We are using the Keras package to build the neural network ([https://keras.io/ (Links to an external site.)](https://keras.io/)). You should read through the quick-start guide for the Keras sequental model here: [https://keras.io/getting-started/sequential-model-guide/ (Links to an external site.)](https://keras.io/getting-started/sequential-model-guide/) You will also need the documentation for the Embedding layer ([https://keras.io/layers/embeddings/ (Links to an external site.)](https://keras.io/layers/embeddings/))

Start by building a network as follows:

- One [Embedding (Links to an external site.)](#) layer, the input_dimension should be the number possible words, the input_length is the number of words using this same embedding layer. This should be 6, because we use the 3 top-word on the stack and the 3 next words on the buffer.
  The output_dim of the embedding layer should be 32.

- A [Dense (Links to an external site.)](#) hidden layer of 100 units using relu activation. (note that you want to [Flatten (Links to an external site.)](#)the output of the embedding layer first).
- A Dense hidden layer of 10 units using relu activation.
- An output layer using [softmax activation. (Links to an external site.)](#)

Finally, the method should prepare the model for training, in this case using [categorical crossentropy (Links to an external site.)](#) as the loss and the [Adam optimizer (Links to an external site.)](#) with a learning rate of 0.01.

```
model.compile(keras.optimizers.Adam(lr=0.01), loss="categorical_crossentropy")
```

TODO: Training a model

The main function of train_model.py will load in the input and output matrices and then train the network. We will train the network for 5 epochs with a batch_size of 100. Training will take a while on a CPU-only setup.

Finally it saves the trained model in an output file.

Training the model takes about 3 minutes per epoch on my 2016 MacBook pro.

You can call the program like this:

```
python train_model.py data/input_train.npy data/target_train.npy data/model.h5
```

Part 4 - Greedy Parsing Algorithm - Building and Evaluating the Parser (35 pts)

We will now use the trained model to construct a parser. In the file decoder.py, take a look at the class Parser. The class constructor takes the name of a keras model file, loads the model and stores it in the attribute model. It also uses the feature extractor from part 2.

TODO: Your task will be to write the method parse_sentence(self, words, pos), which takes as parameters a list of words and POS tags in the input sentence. The method will return an instance of DependencyStructure.

The function first creates a State instance in the initial state, i.e. only word 0 is on the stack, the buffer contains all input words (or rather, their indices) and the deps structure is empty.

The algorithm is the standard transition-based algorithm. As long as the buffer is not empty, we use the feature extractor to obtain a representation of the current state. We then call model.predict(features) and retrieve a softmax actived vector of possible actions.

In principle, we would only have to select the highest scoring transition and update the state accordingly.

Unfortunately, it is possible that the highest scoring transition is not possible. arc-left or arc-right are not permitted the stack is empty. Shifting the only word out of the buffer is also illegal, unless the stack is empty. Finally, the root node must never be the target of a left-arc.

Instead of selecting the highest-scoring action, select the highest scoring permitted transition. The easiest way to do this is to create a list of possible actions and sort it according to their output probability (make sure the largest probability comes first in the list). Then go through the list until you find a legal transition.

The final step (which is already written for you) is to take the edge in state.deps and create a DependencyStructure object from it.

Running the program like this should print CoNLL formatted parse trees for the sentences in the input (note that the input contains dependency structures, but these are ignored -- the output is generated by your parser).

```
python decoder.py data/model.h5 data/dev.conll
```

To evaluate the parser, run the program evaluate.py, which will compare your parser output to the target dependency structures and compute labeled and unlabeled attachment accuracy.

```
python evaluate.py data/model.h5 data/dev.conll
```

Labeled attachment score is the percentage of correct (parent, relation, child) predictions. Unlabeled attachment score is the percentage of correct (parent, child) predictions.

The score for the parser is relatively low (~70 LAS). The current state of the art is ~90. Feel free to experiment with additional features to improve the parser.