Introduction

One of the major modern areas of artificial intelligence is Natural Language Processing (NLP). A common task in this area is the prediction of a word from a context of surrounding words. On its own, this shows up in applications like the autocomplete feature on your smartphone, but it can also be very useful as an intermediate step in more complicated systems (such as word2vec, in which the way the prediction network is trained gives rise to nice semantic clustering and arithmetic properties).

There are many approaches to training a system to make this prediction accurately. In this project, we’ll be using neural networks (via Chainer, which you should be familiar with by this point).

The Data - A Whale of a Tale

We’ll be using the first two chapters of the novel Moby Dick as our dataset, with the first chapter serving as the training set and the second chapter serving as the testing set.

You can find the raw text for these chapters at:
http://www.gutenberg.org/files/2701/2701-0.txt
Task Overview

High level of what you need to do (details can be found in the appropriate section later in this document):

1. Prepare the data for processing
2. Train a neural network on the training set
3. Test your network on the testing set
4. Analyze your results

Preparing the Data

First, download the data as plain text. Extract the parts of the data you will need for your testing and training sets (chapters 1 and 2). Then, write code to:

1. Load the data into Python.
2. Clean up the data - remove all characters except letters, spaces, apostrophes, and hyphens immediately surrounded by letters (for example, “sea-captain’s vessel” should remain as-is, while “and then - gasp!” should have the ‘-’ and ‘!’ excised). This can be accomplished with either regexes or a simple loop through each character. You should also convert all letters to lower case (python has a built-in function to do this). [NEW] Don’t worry about handling the hyphens. We discovered that the source we pointed you to includes some non-ASCII hyphen characters. So, it’s ok to just treat ”sea-captain” as ”sea” and ”captain” separated by a space. [NEW]
3. Tokenize the data. Split it into two lists of tokens (individual units) we’ll call \textit{trainTokens} and \textit{testTokens}, one for the training set and one for the testing set. For this project, we’ll go with 1 word = 1 token, although in other applications compound words or words with affixes might be split up into multiple tokens. This should be easy if you followed the previous step - just use the built-in Python utility to split the text around whitespace delimiters.
4. Build a list of unique tokens. From your tokenized data, construct a single list of all the unique words that appear across both chapters, ordered by their first appearance in the novel. Each distinct word should appear exactly once in this list (even if the word appears in both chapters). We’ll refer to this list as \textit{uniqueTokens}.
5. Build input vectors. For each token in your \textit{uniqueTokens} list, create a vector of length 100 filled with 1s and 0s. We don’t care how you generate these vectors - you could create them randomly, or have some sort of iteration scheme. Just make sure each token gets a unique vector (so if
you generate these randomly, make sure to check for and resolve collisions appropriately), and also make sure that you can access these vectors later (if you generate these in some deterministic, independent way, you can just make a method call, but if you generate them randomly you’ll likely want to store them in a file).

These vectors will serve as the input for our neural network.

**Input and Output Format for Our Neural Network**

The input layer of our neural network will consist of 300 units. This input will represent three consecutive words, so if our text looks like “and then the Great”, the first 100 units will be for “and”, the second 100 for “then”, and the third 100 for “the”.

The output layer of our neural network will consist of a number of units equal to the number of distinct words present in the data - this is why we had you construct the uniqueTokens list. When training, the “correct” output will be presented as a vector that has a 1 in the slot corresponding to the word’s position in the uniqueTokens list and a 0 in all other slots.

The number of hidden layers is up to you to decide. You will need at least 1 hidden layer; we recommend you start out by trying 1 hidden layer with 1000 units.

Ideally, the network would be able to produce this “correct” output each time, but this is infeasible (why? Well, that’s your job to answer, in the Analysis section). Instead, the network will produce positive floating point outputs such that the sum across all output layers is 1. This way, the output value of each unit can be roughly interpreted as the probability that the corresponding word follows the input context.

**Training and Testing**

Use the first chapter of Moby Dick as training data. Our training pairs will be in the form \((c, w)\), where \(c\) is the context (the three 100-length vectors for the three words immediately before the word we are trying to predict) and \(w\) is the correct answer (the vector that has the same size as uniqueTokens, has a 1 in only the slot corresponding to the correct word, and has 0s everywhere else).

So, if we have the example sentence:

> “What of it, if some old hunks of a sea-captain orders me to get a broom and sweep down the decks?”

and we’re trying to predict the word “sea-captain” (notice that we count this as one word, not two), then the context \(c\) would be the vectors for the three words “hunks”, “of”, and “a”, in that order.
The testing (validation) set will work the same way, except for with the second chapter of Moby Dick.

You should use 20 epochs and [NEW] the Adam optimizer. [NEW] The activation function will differ - for the hidden layers, it should be ReLU, but for the output layer it should be Softmax (look it up - it’s a variant on the sigmoid which ensures that the sum over all output units is always 1, giving it a natural interpretation as a likelihood or probability). [NEW] Chainer seems to implicitly be calling softmax on the final layer, so you can just return the final layer as-is (meaning that you never have to explicitly invoke softmax). [NEW]

Analysis

Questions to answer (1-3 sentences for each; use your judgment!):

1. What is the accuracy of your network on the training set? On the testing set? Is there a significant difference between the two?

2. How do you think this would change if you trained on the first 10 chapters and tested on the 11th?

3. Why is it important to retain the ordering of the words within a given context?

4. What is the advantage of using a neural network for this task, rather than simply collecting all the 4-tuples in the text and computing a probability table manually (e.g. if “onward to the chase” appears twice and “onward to the horizon” appears once, given “onward to the” the probabilities would be 2/3 for “chase”, 1/3 for “horizon”, and 0 for all other words)?

5. Even with a much more sophisticated network, why would it be unrealistic to expect accuracy anywhere close to 100% (or even 50%)?

What to turn in

You will need to turn in your Chainer network snapshot (a la Project 2), along with a writeup (in .pdf format) of your answers to the questions in the Analysis section.