Lexical ressources: Word2Vec, Exercise 1

Word2vec - Exercise 2

The aim of this exercise it to familiarize yourself with the word2vec architecture. Understanding the code and playing with it matters more than finding a definitive answer.

In this repository, you will find four python scripts.

- `w2v.py` is the main entry point of the program. When called from the command line, it processes arguments, builds a training environment and starts training a word2vec CBOW model.
- `utils.py` contains the definition of a training environment (it's just a made-up object meant to simplify the code), along with a couple other all-purposes functions and constant values.
- `data.py` is where goes all the logic for handling raw data and converting it into the format expected by the Word2vec Model.
- `model.py` defines a pytorch implementation of a CBOW word2vec model. Pytorch is a popular framework for building and training neural networks. The whole mathematical logic of the word2vec algorithm should be defined in this script.

The exercise consists in implementing the alternate word2vec architecture, the so-called **skip-gram** architecture, mostly by reusing existing pieces of code. Data handling, for the most part, should already be done for you, and you should have at most minor adjustment to make. Most of the work should instead be done in the `model.py` script, where you are to define a new class.

Below is a step-by-step breakdown of how to do that, though not everything is laid out clearly. You should also refer to the slides from today's class for more details on the mathematical aspects of skip-gram.

2.1 Create a new class

The file `model.py` contains the implementation of the CBOW architecture. You will to create a new class in this file in order to implement a skip-gram architecture. The easy way is to copy and adapt the existing `Word2VecCBOWModel` class. The class must verify three things.

- The new class must extend the `torch.nn.Module` class.
- The new class must have a well-defined constructor, that first calls the constructor of its parent `torch.nn.Module` class, and then defines all network parameters as required
- The new class must define a `forward()` method, that takes as a parameter the expected input for the skip-gram model, and produces as output the expected prediction for the skip-gram model.

2.2 Define the new loss function

The loss function being used by the CBOW implementation is a standard negative log-likelihood implemented in Pytorch. You technically can use the same loss function, and consider each context word as a different target. The other possibility it to redfine the loss to consider all context words at once, and sum the outputs for all context words. You may have to re-think how your model is defined: in particular, your `forward()` method will need to return a vector for the word, as well as vectors for the context words.

If you do redefine the loss, note that implementing a loss function in pytorch is very much like implementing a neural network in general: you have to create a specific class, that will extend the `torch.nn.Module` class, and define a `forward()` method.

2.3 Adapt the training environment

Unlike the CBOW architecture, the skip-gram model takes a given word as an input, and tries to predict context items. There are therefore two things you must modify in the `utils.py` file:

- Create a new member to the environment class, that defines which of the two architectures is currently being used.
- Depending on which architecture is used, the training iteration (method `train()`) must handle words and contexts differently. Modify that function so that it starts by checking which algorithm (CBOW or skip-gram) the environment is training for, and then accordingly uses words as input or target and contexts as target or input.

2.4 Create a new command-line argument

The current entry point creates an environment that trains for CBOW by default. To make all your hard work useful, you will have to modify the `w2v.py` file:

- You need to add a command-line argument to the argument parser, so that the user may choose either architecture they prefer
- You need to pass the value of that command-line argument to the `build_env()` function, and modify that function accordingly. The model of the environment should no longer be systematically a CBOW module; likewise, if you defined a specific loss you will need to change that too.

2.5 Test your code!

Implementing a network is great, but it rarely goes the way you expect at the first try. Test your code bit-by-bit, work your way through until you have something that works. Here are some general tips that may prove useful:

- **Use the python interpreter** to test small pieces of code. Rather than testing your whole code at once, break it down in smaller pieces, and test those in the python interpreter directly. You'll be able to check that every little piece of code works as you expect before putting them all together.
- **Read the stacktrace** to get useful hints about what went wrong. Every fatal error in python comes with a stack trace, that tells you *what* happened (in the form of an error type and an error message) and *where* it happened (in the form of a "stack trace", the path that the execution has followed through your code before encountering the error). Knowing *what* broke *where* is a very good start to understanding *why* it broke, and *how* to avoid this failure.
- **Use a debugger** to debug your code. Sometimes stack-traces aren't obvious enough, or you've passed through a few too many lambdas, or you have too many bugs happening at once. Consider using a debugging tool, such as the python module `pdb` that comes shipped in with any and every python distribution. Debuggers allow you to manually control the execution flow of your program: the most basic thing is to tell it when to stop by putting *breakpoints*, at which point you'll be able to print and manipulate values and variables. The good old-fashioned `print` is also a way of looking at how variables evolve throughout execution, but it's less efficient in the long run.

2.6 More things to implement

- A nice way to test your code, and compare how it fares compared to an actual implementation is to use it in the visualisation module you've completed for exercise 1.
- The code, as it is currently defined, is not efficient. There are many things to improve on. In particular the CBOW architecture uses one-hot encodings and a Linear projection: the standard way would be to use a `torch.nn.Embedding` parameter, which takes as input indices stored in a `LongTensor` and produces a `FloatTensor` corresponding to the linear projection for the one-hot.
- One of the strengths of word2vec is that it came with a boatload of nifty machine-learning tricks, none of which have been implemented in this simplistic canvas. Two are especially well-known: the **hierarchical softmax** and the **negative sampling**. Coding them will require major changes to your code: go step-by-step and remember testing!