Nikita Haduong
UW ID: qu
Time: 3 hours, couldn't think of a Test difficulty, had lots of fun with the first Maintaining/ Documentation problem (maybe a research project idea?)

Homework 1

Maintaining, or writing code that can later easily be further developed, even just after a week or two of not looking at it, is a highly frustrating/difficult task that has resulted oftentimes in simply rewriting everything from scratch (with some small amounts of referencing and copy/ pasting). Fortunately, I have not yet had to develop/maintain a large codebase, as such rewriting would be impossible. One example is writing a script that preprocesses text from a specific dataset. I have needed to preprocess the text for this dataset multiple times because each preprocessing task results in slightly more fine-grained output, with more features that we didn't think to preprocess in previous runs. Rather than modifying the original preprocessing script and adding to it, I have often copy-pasted chunks of the original script into a new file and adding additional lines, resulting in many copies of preprocessing scripts with poor documentation.

Code maintenance includes many aspects of software engineering, including documentation and design, and every component has their own difficulties associated with "good practice." Being able to develop a codebase that is easy to maintain by many contributors is a difficult but necessary task because these codebases often need to persist for years and receive updates from many people, all of which have their own coding styles. The human factor here, with coding styles, is a large part of why code maintenance is still so difficult, despite the multitudes of style guides and "best coding practices" manuals that are written and received attempts at being enforced (such as Python's PEP style guides). Even if these style guides suggest writing a documentation string for every function, human interpretation will result in different levels of descriptiveness of these strings. One contributor might write "function F takes a string, converts it into part of speech tags, and returns a dictionary", while another might write"function F is a string-to-part-of-speech converter using NLTK". The first person omits the details of how the string is converted but is more descriptive about the input and outputs of the function. They are probably assuming that if someone looks into the function, they can see that NLTK is clearly being used. Another issue is the frequency of writing the documentation strings. How do you determine which functions get documentation strings, since these strings can take a long time to write when you could be spending the time writing more functions?

A possible solution to this specific documentation-maintenance problem would be to auto-generate summaries of every function in such a way that it's clear exactly what each line/ component in the function is doing such that if someone wanted to modify the function, they could easily locate the spots in the function that they need to edit. But this is a difficult problem. It requires a very intelligent system and some additional strictness in coders using the right/ similar styles. Currently, this "intelligent system" that is documenting code is a human, perhaps an intern. Human labor can't keep up with the amount of code being generated, however, and developers would rather write new features than write up documentation.

One of the biggest design challenges I've been facing is with the execution order of modules (specifically in Unity, but generally with respect to games). Where do I put the modules, and how do I write modules that need to be executed roughly in some sequential order, but I have no control over the actual order of execution? To explain a specific situation: say I need to initialize two variables in two separate modules (that don't need to talk to each other), but the second variable is dependent on the first one being initialized. Within the larger scope of the program, I can say "execute module-1 and module-2 at the beginning of the run", but I can't necessarily say "execute module-1AND THEN module-2" because module-1 is automatically executed by Unity (a game engine), just not necessarily before module-2.

Module execution order and design is important to people because sometimes the task at hand requires clearly sequential execution, sometimes it doesn't matter, and sometimes it needs as-parallel-as-possible execution, which could technically be impossible due to hardware limitations. If using some sort of framework, such as Unity, where there's a lot of infrastructure code that can be difficult to dig through, one can only resort to hacks to get modules to execute in the correct order, which can mean a lot of extraneous code (e.g. an additional line that says DON'T EXECUTE MODULE-2 UNTIL MODULE-1 EXECUTES). If developing software at a low level without the framework overhead, then you can manually add some priority flag to the modules, which is probably the best option one can ask for, as they are directly able to decide exactly the execution order of modules. However, most people nowadays are not writing software at such a low level, so the next best option would be for framework developers to expose API options that allow users to access the priority flags.

I honestly haven't done much (or rather, any) testing of software in my software development career (which is rather short), so I can only think of some challenges that might have to do more with implementation than general software engineering. One general challenge is for testing the soundness (correctness?) of large neural models. These models require many resources that are particularly scarce in academia (GPUs, time), so it is difficult to run thorough tests on these models (at least in academia). The models are large and a black box, and it could be possible that some iterations during optimization have side effects or encounter situations that make the optimization behavior do something unexpected--which we would never be able to see or interpret: we would only see some final results. An additional challenge in testing these models is that we aren't too sure what kinds of tests we need to run, as the models can have billions of parameters, and we don't actually understand what the parameters are referring to or why they have to be the value that they are, after a hyperparameter search.

Some tests for these models might include perturbing the data or introducing adversarial data points to see how robust the model is. However, failure by the model to accurately classify these altered data doesn't necessarily mean the model itself is failing or unsound--it's more that the model has learned some patterns that don't really make sense to a human and work most of the time. It looks like there has been some recent work on creating more classical test suites for neural models (Sun et. al. 2018).