PROGRAMMING IN PYTHON I

Unit 11: Speeding up Python code

Michael Widrich
Institute for Machine Learning

Motivation

- Native Python code, without additional modules, is relatively slow
  - Interpreted language, no compilation/optimization of whole program code
  - Dynamic typing, run-time used on checking/handling dynamic object datatypes
- We have already seen some modules that allow us to write faster Python code
  - So far by providing dedicated optimized functions and tools for parallel execution of tasks
  - NumPy, pandas, subprocess, multiprocessing, ...
- Now we will learn how to
  - approach the task of speeding up our programs in general
  - compile and optimize custom Python code (see code files for this Unit)

Speeding up Python code: Considerations

- Speeding up Python code requires work and possibly code adaptation
  - There is no one-fits-all solution
  - Might introduce bugs or result in less general code
- Before trying to speed up your code, you should consider:
  - Is it worth investing the effort?
    (Also w.r.t. other tasks that might need to be done.)
  - Is it worth the possible limitations and less general code?
  - Is it worth the risk of introducing bugs if you need to alter an already tested system?

Speeding up Python code: Where to start?

- To speed up our code efficiently, we need to know which parts of the code are slowing us down
  - A common mistake is to invest effort into speeding up code parts which are only marginally relevant for over-all run-time
- To identify the bottlenecks, we need to time our code
  - Measure run-time of code and sub-parts of code to determine contribution to over-all run-time
    - E.g. via `time` (see code files) or `timeit` (next semester) modules
  - Be aware that run-times might change depending on hardware, OS and drivers, package versions, other running processes, processed data, ...
    - Take average over repeated timings where possible, use same setup to compare timings
  - Time your code before and after optimization

Further reading: `https://docs.python.org/3/library/timeit.html`

Using the tools we have (1)

- There are different levels at which code can be speed up and optimized
  - Which level will yield best improvements depends on use-case
  - Design choices on one level may impact other levels
- With what we learned in this semester, we can already speed up and optimize our code
- The following slides will focus on what I find most relevant for basic ML Python applications

Further reading: `https://en.wikipedia.org/wiki/Program_optimization`

Using the tools we have (2)

- Design
  - How do we want to solve our task
  - What do we want to pay attention to (e.g. focus on run-time or memory consumption)
  - Depends on the goal we want to achieve
- Algorithms and data structures
  - (Abstract) choice of algorithms and data structures
  - E.g.:
    - Reformulating a formula such that it contains less computations
    - Choosing an array with elements of same datatype vs. elements of variable datatypes
  - The best performing algorithm might depend on the use-case (e.g. large or small matrices or mix of both?)

Using the tools we have (3)

- Source code
  - Choices of how algorithms are realized
  - E.g.:
    - List-comprehension instead of loop
    - Replacing native Python function with NumPy equivalent
- Platform-dependent optimization
  - Optimizing your code for the machine you are running it on
  - Very important for ML but dependent on hardware and system (see next semester)
  - E.g.:
    - Running operation on CPU or GPU?
    - Storing data in HDD, SDD, or RAM?
    - Using multiprocessing to run operations in parallel? If yes, how many parallel operations?*

*) Further reading: `https://en.wikipedia.org/wiki/Parallel_slowdown`