Week 10 - Profiling

Profiling is an analysis technique to measure certain properties of a program to optimize its performance. For instance, one might be interested in the number of times a certain function is called, the fraction of time spent in a part of the code, if there are bottlenecks, etc.

In this exercise session, you will use different tools to profile your code to identify its weaknesses. In a second step, you will then optimize your code.

Make sure to release your code by 10am next Monday (release notes)!
For your release be sure that your API is correct, and clearly indicate any dependencies of your code!

1 Measuring runtime of code and functions

1.1 timeit

A quick way to measure the runtime of single functions is to employ the library `timeit` (https://docs.python.org/3/library/timeit.html). It is very easy to use:

```
from timeit import default_timer as timer

start = timer()
main()
end = timer()
duration = end - start # duration in seconds
```

1.2 cProfile

For a comprehensive approach to profiling we recommend `cProfile`. It can be used in the following way:

1. Take the code of your project which runs the evolution of the system (for example the code from the script *main.py* or any other script you use for running your weekly experiments, e.g. *experiments_week8.py*) and make sure that this code sits in a function called `main()` (in the script `main.py`).
2. Create a script called *profiling.py* to run `cProfile`. The script should contain the following code where you need to replace `<your_module>` with the actual name of your module:

   ```
   import cProfile
   import pstats
   from <your_module> import main

   cProfile.run("main()", "restats")
   p = pstats.Stats('restats')
   p.sort_stats('cumulative').print_stats(20)
   # Change the number if you want to visualize more stats
   ```

   From the terminal, you can run the script and try to interpret the output.

Interpretation of the output: At the top you should see the most computationally intensive functions. If the number of iterations is sufficiently large, you should expect a large portion of the execution time to be spent inside the `update` function.

1.3 pytest-benchmark

Two weeks ago we wrote our first tests, by using `doctests` and `pytests`. Luckily, there is a very smooth integration of speed measurements into `pytests`! Have a look at the documentation (`https://pytest-benchmark.readthedocs.io/en/latest/`) and integrate pytest-benchmarks into your testing scripts.

For example, if you want to profile the *hebbian weights* function, you can include the benchmark option in your test function. It has to be passed as argument of the function and then you can use *benchmark* alone or *benchmark.pedantic* to increase the number of rounds. In the documentation, you will find both rounds and iterations. During benchmarking, the tested function will run for *n* rounds, where it will iterate *m* times in every round. In total, the tested function will be tested $n \times m$ times.

Here, we recommend to keep fix the number of iterations at 1 and to run different number of rounds (e.g., 5 or 10). For example, look at the following code that will run the function `hebbian_weights` 5 times:

```
def test_hebbian_weights(benchmark):
    ...
    weights = benchmark.pedantic(hebbian_weights,args=(patterns,), rounds=5, \
    iterations=1)
    ...
```

If you have more than one argument, you can modify `args=(arg1,)` to `args=(arg1,arg2)`.
Use `pytest-benchmark` for benchmarking the following functions:

- **Hopfield project**

  Using the following parameters (similar to the example of the checkerboard): $num\_patterns$=50, $network\_size$=2500, $num\_perturbations$=1000, max_iter_sync=20, max_iter_async=30000

  - measure duration of generating **W** with the hebbian rule
  - measure duration of generating **W** with the storkey rule
  - run the update function in a synchronous way
  - run the update function in an asynchronous way
  - measure duration of the energy function

- **Turing pattern formation project**

  Using the same parameters of the last exercise session about testing:

  - measure duration of the diffusion with **u** having shape (100,100)
  - measure duration of binarization function
  - run the dynamics function for 100 iterations

2 Optimization strategies

Did you expect the profiling results? Which functions were the slowest ones? Do you see common patterns of your slowest functions?

Here, we provide some optimization techniques you can leverage to optimize your code:

for-loops

for loops are typically slow operations in Python. Whenever it is possible, vectorized operations of high-performance numerical libraries (e.g., Numpy, Scipy) are much faster. Furthermore, python container datatypes (e.g., list, dict, set) support the highly optimized *comprehension* syntax. For example, if you want to compute the square of a list of numbers, a traditional for loop like

```
values = list(range(100))
squared = []
for v in values:
    squared.append(v * v)
```

is slower than the compact version

```
values = list(range(100))
squared = [v * v for v in values]
```

although they carry out the same operations. Other container datatypes also support the same syntax. If you have implemented operations like filling the matrix elements or vector-vector multiplications using nested loops, you might want to replace them by Numpy operations. You might take advantage of the functions `np.fill_diagonal`, `np.dot`, `np.outer`. You can also assign the same value to multiple elements of a matrix efficiently by passing a vector of indices.

numba/jit

`Numba` is a library that translates python functions to optimized machine code and can hence speed up your code. The Numba-compiled algorithms get closer to the speed of C and FORTRAN. You need to install `Numba` using `conda install numba` in your virtual environment. To use `Numba` with your functions, you simply need to add decorators (`@jit`), as described in the documentation (https://numba.readthedocs.io/en/stable/user/jit.html).

cython

Another powerful library you can use is `Cython`. It allows you to speed-up your Python code automatically translating it into C code. First, install `Cython` in your virtual environment with `pip install Cython`. In order to use this library, you need to develop a module with the function that you want to be converted. For example, try to use `Cython` with the update/dynamics function.

1. Create a module called `update_cython.pyx` in which you will add update function.
2. The update module has to be built using a `setup.py` script which should contain:

   ```
   from distutils.core import setup
   from Cython.Build import cythonize

   setup(ext_modules = cythonize('update_cython.pyx'))
   ```

3. Now, you can build the module by typing on the terminal:

   ```
   python setup.py build_ext --inplace
   ```

   A new file called `update_cython.c` should appear in your directory.
4. Although the cython module will have a .c extension, you can import it as a python module. Therefore, in the script where you use the update function, you can try to comment the previous update function and import the cython one using `from update_cython import update`

Run the code again. Is it faster now?

3 Optimization opportunities

After having figured out where optimization opportunities are and how to meet them, we now want to speed up your code as much as possible. First try to reduce as many for-loops as you can! Then, feel free to try out the different optimization strategies and see which one works best for you.

3.1 Hopfield network

The following are suggestions to optimize specific functions of your code for the hopfield network:

- `def generate_patterns(num_patterns, patterns_size)`

  You can use the function `np.random.choice` to compute the value of pattern matrix between -1 and 1.

- `def perturb_pattern(pattern, num_perturb)`

  You can use the function `np.random.choice` to retrieve the indexes of the pattern you want to modify.

- `def hebbian_weights(patterns)`

  You can take advantage of matrix multiplication (`np.dot`) to compute the sum between patterns. Specific conditions can then be matched by using the `np.fill_diagonal` function.

- `def storkey_weights(patterns)`

  You can use different tricks to speed up the computation of the **H** matrix. For example, you can think about the combination of matrix summation, difference or multiplication taking advantage of the following functions: `np.outer, np.dot, np.diag`. You can make use of the test you already developed to check the correctness of the matrix.

- `def update(state, weights)`

  Also here, have a look on `np.dot` and `np.where`.

- `def energy(state, weights)`

  The energy function could be improved by making use of, e.g., `np.dot`.

Run the profiler again. Is your code faster now? You can go on optimizing and profiling until you are satisfied with the performance of your code.

3.2 Turing pattern formation

The following are suggestions to optimize specific functions of your code for the turing pattern formation:

- `def diffusion(u)`

  One helpful function here is `np.pad`. For computing the actual diffusion, a 2D-convolution operation comes in handy, e.g. via the library of `scipy.signal`.

- `def binarize(state)`

  This function can benefit from `np.where`.

Run the profiler again. Is your code faster now? You can go on optimizing and profiling until you are satisfied with the performance of your code.