Week 6 - Functionify

In this exercise session, keep working on the project that you have selected last week.

The objective of today is to organize your code with functions. Functions add a level of abstraction to your code, and thereby improve readability, reusability, and encapsulation, condensing multiple instructions into a single line of code.

Furthermore, they offer a simple solution to avoid code duplication and to minimize complexity: if you want to use some lines of code in multiple places, wrap them in a function and reuse it!

Make sure to release your code by 10am next Monday (release notes)!

1 Hopfield network

1.1 Refactoring

We start by reorganizing the code from last week into functions. In this first part we do not introduce new functionality. You can write the functions in a separate file from `main.py`.

Your task: refactor the code by creating the following functions:

- generate the patterns to memorize. `generate_patterns(num_patterns, pattern_size)` receives in input two integers and returns a 2-dimensional numpy array, in which each row is a random binary pattern (possible values: $\{1, -1\}$) of size `pattern_size`.
- perturb a given pattern. `perturb_pattern(pattern, num_perturb)` accepts as input a binary pattern and the number of perturbations. It samples `num_perturb` elements of the input pattern uniformly at random and changes their sign. It returns the perturbed pattern.
- match a pattern with the corresponding memorized one. Write the function `pattern_match(memorized_patterns, pattern)` which receives as an input the matrix of the initially memorized patterns and another pattern. It returns `None` if no memorized pattern matches, otherwise it returns the index of the row corresponding to the matching pattern.
- apply the hebbian learning rule on some given patterns to create the weight matrix. Create a function `hebbian_weights(patterns)` which returns the weights matrix.
- apply the update rule to a state pattern. Write the function `update(state, weights)`, receiving the network state and the weights matrix and returning the new state.
- apply the asynchronous update rule to a state pattern. Write the function `update_async(state, weights)` which behaves similarly to the previously defined `update` function. However, instead of computing the full update $\mathbf{p}^{(t+1)} = \sigma\left(\mathbf{W}\mathbf{p}^{(t)}\right)$, it just updates the i-th component of the state vector (with i sampled uniformly at random) by computing the new value $p_i^{(t+1)} = \sigma\left(\mathbf{w}_i \cdot \mathbf{p}^{(t)}\right)$. In the previous expression, $\mathbf{w}_i$ denotes the i-th row of the matrix $\mathbf{W}$, while $\cdot$ is the standard scalar product between two vectors.
- `dynamics(state, weights, max_iter)`, to run the dynamical system from an initial state until convergence or until a maximum number of steps is reached. Convergence is achieved when two consecutive updates return the same state. The function should return a list with the whole state history.

- `dynamics_async(state, weights, max_iter, convergence_num_iter)`, to run the dynamical system from an initial state until a maximum number of steps is reached. For the asynchronous updates we cannot apply the previous convergence criterion. In fact, if the update rule samples an element of the pattern which is already at convergence, there will likely be no change in that iteration. You can therefore set a softer convergence criterion: if the solution does not change for `convergence_num_iter` steps in a row, then we can say that the algorithm has reached convergence. The function should return a list with the whole state history.

In a file called `main.py`, test your functions by executing the following experiments:

- Generate 80 random patterns of size 1000
- Choose a base pattern and perturb 200 of its elements
- Run the synchronous update rule until convergence, or up to 20 iterations. Did the network retrieve the original pattern?
- Run the asynchronous update rule for a maximum of 20000 iterations, setting 3000 iterations without a change in the state as the convergence criterion. Did the network retrieve the original pattern?

1.2 Storkey rule

Once your previous code is better organized, we can now add some more functionality to it.

Last week we have introduced the Hebbian learning rule to set the network weights, but other rules exist. In [Sto97], Amos Storkey showed that a different learning rule, now named after him, allows the network to memorize a larger number of patterns, given the same number of neurons. The rule to train the weights of the Hopfield network is given by the following iterative definition:

$$w_{ij}^{\mu} = w_{ij}^{\mu-1} + \frac{1}{N}(p_i^{\mu}p_j^{\mu} - p_i^{\mu}h_{ji}^{\mu} - p_j^{\mu}h_{ij}^{\mu}) \tag{1}$$

where

$$h_{ij}^{\mu} = \sum_{k:i\neq k\neq j} w_{ik}^{\mu-1}p_k^{\mu} \tag{2}$$

and p_i^{μ} denotes the i-th component of the μ-th pattern to memorize, while w_{ij}^{μ} are the components of the matrix $\mathbf{W}$ after the μ-th pattern has been processed.

Your task:

- Implement the Storkey learning rule for some patterns to create the weight matrix. Implement a function `storkey_weights(patterns)` which returns the weights matrix.[1]
- Verify that also the Storkey rule defines a dynamical system, which converges to the most similar memorized pattern. You can proceed as in the previous exercise, keeping the same network parameters, but using the Storkey weights instead of the Hebbian weights. Does the network still converge to the original pattern?

[1] Hint: proceed iteratively. Initialize the weights matrix to zeros with the command `np.zeros((pattern_size, pattern_size))`. At each iteration, process one of the input patterns. First compute the matrix $\mathbf{H}$ with components h_{ij}, then use the pattern and the matrix $\mathbf{H}$ to update the matrix $\mathbf{W}$ with components w_{ij}.

2 Turing pattern formation

2.1 Refactoring

We start by reorganizing the code of last week into functions. In this first part we do not introduce new functionality to the code. You can write the functions in a separate file from `main.py`

Your task: refactor the code by creating the following functions:

- `diffusion(u)`, which applies the diffusion operator to a state $\mathbf{u}^{(t)}$ and returns $\mathbf{D}\mathbf{u}^{(t)}$, defined (by index) as

$$Du_{i,j}^{(t)} = u_{i-1,j}^{(t)} + u_{i+1,j}^{(t)} + u_{i,j-1}^{(t)} + u_{i,j+1}^{(t)} - 4u_{ij}^{(t)} \tag{3}$$

- `diffusion_dynamics(u_0, d_t, d_x, num_iter, store_every=1)`, which makes the dynamical system evolve from an initial state $\mathbf{u}^{(0)}$ `num_iter` steps, following the update rule

$$u_{ij}^{(t+1)} = u_{ij}^{(t)} + \Delta t \frac{u_{i-1,j}^{(t)} + u_{i+1,j}^{(t)} + u_{i,j-1}^{(t)} + u_{i,j+1}^{(t)} - 4u_{ij}^{(t)}}{\Delta x^2} \tag{4}$$

  The input parameters `d_t` and `d_x` denote the temporal step Δt and the spatial step Δx, respectively. The function returns a list of states, representing the evolution of the system. The parameter `store_every`, defaulted to 1, regulates how many steps a state is stored in the list to return (it can be used to reduce the length of the list, e.g, by returning only one every 10 states).

Test the functions that you have implemented by verifying that they produce the same results as the code that you wrote last week.

2.2 Reaction-diffusion problem

Now that you have implemented the diffusion component of the dynamical system, we can introduce the reaction term in the equation. The solution to the time-dependent reaction-diffusion problem can be found by applying the rules defined by 5 and 6. In vectorial form, they write:

$$\mathbf{u}^{(t+1)} = \mathbf{u}^{(t)} + \Delta t \left[\gamma f(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}) + \frac{1}{\Delta x^2} \mathbf{D}\mathbf{u}^{(t)} \right] \tag{5}$$

$$\mathbf{v}^{(t+1)} = \mathbf{v}^{(t)} + \Delta t \left[\gamma g(\mathbf{u}^{(t)}, \mathbf{v}^{(t)}) + \frac{d}{\Delta x^2} \mathbf{D}\mathbf{v}^{(t)} \right] \tag{6}$$

where

$$f(u,v) = a - u - h(u,v), \quad g(u,v) = \alpha(b-v) - h(u,v), \quad h(u,v) = \frac{\rho u v}{1 + u + K u^2}$$

while $\mathbf{D}\mathbf{u}^{(t)}$ and $\mathbf{D}\mathbf{v}^{(t)}$ can be computed from $\mathbf{u}^{(t)}$ and $\mathbf{v}^{(t)}$ by applying the diffusion operator, as defined in (3).

Your task:

- Implement the functions `h(u, v, rho, K)`, `f(u, v, a, rho, K)` and `g(u, v, alpha, b, rho, K)`

- Create a function `update(u, v, d_x, d_t, alpha, k, rho, a, b, d, gamma)` to execute the update rule for $\mathbf{u}^{(t)}$ and $\mathbf{v}^{(t)}$ for one iteration
- Write the function `dynamics(u_0, v_0, num_iter, d_x, d_t, alpha, k, rho, a, b, d, gamma, store_every=1)`, which applies the update rule for `num_iter` steps, and returns two lists (one for $\mathbf{u}$ and one for $\mathbf{v}$) with the history of the states. The parameter `store_every` has the same meaning as in the function `diffusion_dynamics`
- Write the function `perturb(state, std)` which takes a state as input (a 2-D numpy array) and returns a perturbed state, by adding gaussian noise, with 0 mean and `std` standard deviation.

To test the previously defined functions we define a problem with the following parameters: $M = 100$, $N = 100$, $\Delta x = 0.1$, $\Delta t = 0.0001$, $\alpha = 1.5$, $K = 0.125$, $\rho = 13$, $a = 103$, $b = 77$, $d = 7$, $\gamma = 0.5$.

1. Define the initial states $\mathbf{u}^{(0)}$ and $\mathbf{v}^{(0)}$, two $M \times N$ matrices, whose elements are the value of u and v in each node of the discretization grid (as explained last week). Initialize them as random perturbations of the stable state, by applying gaussian noise with unitary standard deviation to it. For the given set of parameters, the stable state is 23 for u and 24 for v.
2. Simulate the system dynamics for 1000 iterations and check that the values of the state's elements are in a reasonable range. Next week's exercise session will focus on the visualization of the results, and you will be able to verify that the system models a pattern formation process.

References

[Sto97] Amos Storkey. "Increasing the capacity of a Hopfield network without sacrificing functionality". In: *International Conference on Artificial Neural Networks*. Springer. 1997, pp. 451–456.