Chapter – 4 (implementation – Part 4 – Test the AI )

**Artificial Intelligence**

**AI-app (CNN-A3C) "Breakdown" Game play** (part 4)

Test agent (explanation)

Main run code (explanation)

Test the AI

part 4: Test the AI: Test agent, main run

**4.13: Test agent (test.py explanation)**

We made the brains of A3C and we trained them. But now we still have to make a *test agent* which will not update the model at all but it will just use the shared model to make its own explorations. We'll *record videos* of the *test agent* playing Breakout with a *certain score*. Instead of coding line by line, we'll focus on understanding it.

* ***Import the libraries:*** We start by importing the necessary libraries for the project. We import the following libraries

**import** torch

**import** torch.nn.functional **as** F

**from** envs **import** create\_atari\_env

**from** model **import** ActorCritic

**from** torch.autograd **import** Variable

**import** time

**from** collections **import** deque

* **test() *function:*** We define the **test()** function, which will make this test agent to do its own *exploration* and *play* the breakout game.

# *Making the test agent (won't update the model but will just use the shared model to explore)*

**def** **test**(rank, params, shared\_model):

* This test function takes three arguments the first one is **rank** which is still used to *de-synchronize* the *test agent* as we did for the *training* *agents*.
* Then we need **parameters** and of course we have the **shared\_model** because this *test agent* will use a *shared model* to do its *own* *exploration*.
* ***De-synchronize:*** Following line of code de-synchronizes to test agent.

**torch.manual\_seed**(params.seed + rank) # *asynchronizing the test agent*

* ***Import the environment:*** Following line of code means that we run one environment with video.

env = **create\_atari\_env**(params.env\_name, video=**True**) # *running an environment with a video*

* When we run the main.py (next section) code, env\_name will be replaces by the breakoutv0, so that we can go into that specific environment and play the game
* video=**True** means that will get the videos of our AI playing Breakout.
* ***De-synchronizing the environment:*** Then at the next line of code we de-synchronize this environment (same as train() function)

**env.seed**(params.seed + rank) # *asynchronizing the environment*

* ***Creating a model and initializations:*** To get our model, we create an object of the **ActorCritic** class and we put the input shape­— env.observation\_space.shape[0] and our outputs— env.action\_space.

model = **ActorCritic**(env.observation\_space.shape[0], env.action\_space) # *creating one model*

* ***eval mode:*** something new here since we're done with the training, we don't want to put the model in train mode we want to put it in eval mode.

# *putting the model in "eval" mode because it won't be trained*

**model.eval**()

* It just basically put the test agent in a mode that will basically *test it's performance* (i.e *evaluate* it).
* ***Input states:*** Then here we get our input states which are the input images (numpy array) from the game. Then we convert them into torch tensors.

state = **env.reset**() # *getting the input images as numpy arrays*

state = **torch.from\_numpy**(state) # *converting them into torch tensors*

* ***Rewards, done:*** Here we initialize *sum of the rewards* and *done*:

reward\_sum = 0 # *initializing the sum of rewards to 0*

done = **True** # *initializing done to True*

* ***Time:*** Something new again, we introduce this start\_time with **time.time**() function to measure the *time of computations*(that's because we want to get the starting point).

# *getting the starting time to measure the computation time*

start\_time = **time.time**()

* ***Deque:*** For the actions we use a very *practical type of queue* that allows to add an element to the *queue* from the *right or from the left*.

# *cf https://pymotw.com/2/collections/deque.html*

actions = **deque**(maxlen=100)

* The reference link: <https://pymotw.com/2/collections/deque.html>
* ***Deque:*** A *double-ended queue*, or deque, supports adding and removing elements from *either end*. The more commonly used *stacks* and *queues* are degenerate forms of *deques*, where the inputs and outputs are restricted to a *single end*.
* ***Episode length:*** Then we *initialize* the *length of an episode* with zero.

# *initializing the episode length to 0*

episode\_length = 0

* ***While LOOP:*** We'll increment that episode length in this WHILE-loop by 1.

**while** **True**: # *repeat*

        episode\_length += 1 # *incrementing the episode length by one*

**if** done: # *synchronizing with the shared model (same as train.py)*

**model.load\_state\_dict**(**shared\_model.state\_dict**())

            cx = **Variable**(**torch.zeros**(1, 256), volatile=**True**)

            hx = **Variable**(**torch.zeros**(1, 256), volatile=**True**)

**else**:

            cx = **Variable**(cx.data, volatile=**True**)

            hx = **Variable**(hx.data, volatile=**True**)

* When the game is done/over we *reload* the *last state* of the *shared model* (that was updated by the other models), but here we don't update the shared model.
* So if the game is over/done we re-initiate the cell states cx and then hidden state hx and else if the game is not over we keep those states.
* Also we make sure that those variables are torch Variables.
* ***Prediction:*** Based on those cell states cx and then hidden state hx (updated based on the done or not) we get the predictions of the model

        value, action\_value, (hx, cx) = **model**((**Variable**(**state.unsqueeze**(0), volatile=**True**), (hx, cx)))

* So we get the value which is the *output of the critic* the action\_value which is the *output of the actor* and the states tuple (hx, cx).
* ***Selecting Actions:*** We generate a distribution of probabilities of the actions (i.e. the Q values), we do this with the softmax() function. This function normalizes the action values into a probability distribution.

        prob = **F.softmax**(action\_value)# *Compute probabilities of actions*

# *the test agent does not explore, it directly plays the best action*

        action = **prob.max**(1)[1].**data.numpy**() # *Select the action with the highest probability*

* **F.softmax**(action\_value) converts *action* values into *probabilities*.
* **prob.max**(1)[1]: Finds the *index* of the *action* with the *highest probability*.
* **data.numpy**(): Converts the result into a *NumPy array* for compatibility.
* ***No Log-Probabilities in Testing:*** Unlike training, the test agent does not require Log-Probabilities because it is not learning or optimizing. It only *focuses* on *playing* the game.
* It will simply *execute actions* using a method similar to SoftmaxBody() from the previous Doom project, but *no training is involved* here.
* ***Selecting the Best Action:*** The test agent selects the action with the highest probability (using ***argmax***) directly from the softmax output. So we use this prob to play the action by taking directly the ***argmax*** of these probabilities i.e. taking the action that has the highest probability.
* This is because the *test agent does not perform exploration*. It is designed to evaluate performance by consistently choosing the best action.
* ***Why No Exploration in Testing?***
* *Exploration* is important during ***training*** to *learn about less obvious actions* and improve the policy. Remember, during training, we want to give our agents a chance to select *lower-probability actions* during *exploration*, rather than always choosing the action with the highest probability.
* But in ***testing***, the agent's goal is to *demonstrate its learned behavior*, so it always picks the most probable (best) action.
* Once we play the action using **env.step**(action[0, 0]) we reach the next state, we get the next reward and done (is game over or not). Since we just got a new reward, we're going to update *sum of the reward*.

        state, reward, done, \_ = **env.step**(action[0, 0])

# *done = done or episode\_length >= params.max\_episode\_length*

        reward\_sum += reward

All code at once (test.py)

# *Test Agent*

**import** torch

**import** torch.nn.functional **as** F

**from** envs **import** create\_atari\_env

**from** model **import** ActorCritic

**from** torch.autograd **import** Variable

**import** time

**from** collections **import** deque

# *Making the test agent (won't update the model but will just use the shared model to explore)*

**def** **test**(rank, params, shared\_model):

**torch.manual\_seed**(params.seed + rank) # *asynchronizing the test agent*

    env = **create\_atari\_env**(params.env\_name, video=**True**) # *running an environment with a video*

**env.seed**(params.seed + rank) # *asynchronizing the environment*

    model = **ActorCritic**(env.observation\_space.shape[0], env.action\_space) # *creating one model*

**model.eval**() # *putting the model in "eval" mode because it won't be trained*

    state = **env.reset**() # *getting the input images as numpy arrays*

    state = **torch.from\_numpy**(state) # *converting them into torch tensors*

    reward\_sum = 0 # *initializing the sum of rewards to 0*

    done = **True** # *initializing done to True*

    start\_time = **time.time**() # *getting the starting time to measure the computation time*

    actions = **deque**(maxlen=100) # *cf https://pymotw.com/2/collections/deque.html*

    episode\_length = 0 # *initializing the episode length to 0*

**while** **True**: # *repeat*

        episode\_length += 1 # *incrementing the episode length by one*

**if** done: # *synchronizing with the shared model (same as train.py)*

**model.load\_state\_dict**(**shared\_model.state\_dict**())

            cx = **Variable**(**torch.zeros**(1, 256), volatile=**True**)

            hx = **Variable**(**torch.zeros**(1, 256), volatile=**True**)

**else**:

            cx = **Variable**(cx.data, volatile=**True**)

            hx = **Variable**(hx.data, volatile=**True**)

        value, action\_value, (hx, cx) = **model**((**Variable**(**state.unsqueeze**(0), volatile=**True**), (hx, cx)))

        prob = **F.softmax**(action\_value)

        action = **prob.max**(1)[1].**data.numpy**() # *the test agent does not explore, it directly plays the best action*

        state, reward, done, \_ = **env.step**(action[0, 0]) # *done = done or episode\_length >= params.max\_episode\_length*

        reward\_sum += reward

**if** done: # *printing the results at the end of each part*

**print**("Time {}, episode reward {}, episode length {}".**format**(

**time.strftime**("%Hh %Mm %Ss", **time.gmtime**(**time.time**() - start\_time)), reward\_sum, episode\_length))

            reward\_sum = 0 # *reinitializing the sum of rewards*

            episode\_length = 0 # *reinitializing the episode length*

**actions.clear**() # *reinitializing the actions*

            state = **env.reset**() # *reinitializing the environment*

**time.sleep**(60) # *doing a one minute break to let the other agents practice (if the game is done)*

        state = **torch.from\_numpy**(state) # *new state and we continue*

* ***Printing the results:*** When the game is done (when AI finishes playing the game) we're going to print the results with the Time, the episode reward, the length of the episode episode length i.e how much time did it last playing Breakout.

**if** done: # *printing the results at the end of each part*

**print**("Time {}, episode reward {}, episode length {}".**format**(

**time.strftime**("%Hh %Mm %Ss", **time.gmtime**(**time.time**() - start\_time)),

reward\_sum, episode\_length))

* ***Reinitializing:*** Since the game is over and we want to start a new game we're going to re-initialize everything. **actions.clear**() resets the input images by putting all the bricks altogether.

            reward\_sum = 0 # *reinitializing the sum of rewards*

            episode\_length = 0 # *reinitializing the episode length*

**actions.clear**() # *reinitializing the actions*

            state = **env.reset**() # *reinitializing the environment*

* We use **time.sleep**(60) to do a break of one minute to *let the other agents practice*.

# *doing a one minute break to let the other agents practice (if the game is done)*

**time.sleep**(60)

* ***New state:*** Following line of code will get us the new state and then we can continue in the new game.

        state = **torch.from\_numpy**(state) # *new state and we continue*

We'll discuss the main *function* (main.py) in the next section we will not code it line by line but we'll explain the code.

**4.14: Main run code (explanation)**

In this section we'll explain the *main code* (main.y) that will execute the whole thing. We start by importing the *libraries* and the *modules* and also the different *classes* and *functions* we made like ActorCritic from our **model.py** file, the train function from the **train.py** file and the test function from the **test.py** file, also we import our optimizer my\_optim.

**from** \_\_future\_\_ **import** print\_function

**import** os

**import** torch

**import** torch.multiprocessing **as** mp

**from** envs **import** create\_atari\_env

**from** model **import** ActorCritic

**from** train **import** train

**from** test **import** test

**import** my\_optim

* ***Gathering all the parameters:*** We organize all adjustable parameters into a Params() class. This class is used to create a Params object, which holds values like *learning rate*, *Gamma* or *Tau*, and environment settings.

# *Gathering all the parameters (that we can modify to explore)*

**class** Params():

**def** **\_\_init\_\_**(self):

        self.lr = 0.0001 # *Learning rate (small for stable training)*

        self.gamma = 0.99 # *Discount factor for rewards*

        self.tau = 1. # *GAE decay parameter*

        self.seed = 1

        self.num\_processes = 16

        self.num\_steps = 20

        self.max\_episode\_length = 10000

        self.env\_name = 'Breakout-v0'

* lr: It is learning rate, we choose a small learning rate.
* gamma: Gamma parameter, we take 0.99.
* tau: Tau is set to 1. Often referred to as the *GAE decay parameter*, it determines the level of smoothing in the advantage estimation.A *smaller tau* gives more weight to *immediate rewards*, while a *larger tau* incorporates *more distant rewards* into the advantage calculation.
* seed: Seed is also 1.
* num\_processes: We set 16 processes. It's the number of parallel processes.
* num\_steps: 20 Steps per update
* max\_episode\_length: max length of 10000 (we spoke about that earlier) this is the parameter we set to make sure the agent doesn't get stuck indefinitely into a state of the environment so this will stop the game. It prevents endless episodes.
* env\_name: name of our environment, 'Breakout-v0'. We can switch to other environments by changing the env\_name parameter, allowing the agent to play other versions of Breakout or even different Atari games.
* ***Main run:*** Now let's discuss the main code for the main run. Following set one thread per core, we get all of our parameters by creating a new object of the **Params**() class which will initialize all the parameters attached to following params object.
* Then we set the seed, and then we get our environment using the **create\_atari\_env**() function with the name of our environment 'Breakout-v0' using params.env\_name.

#----  *Main run* ----

os.environ['OMP\_NUM\_THREADS'] = '1' # *1 thread per core*

# *creating the params object from the Params class, that sets all the model parameters*

params = **Params**()

**torch.manual\_seed**(params.seed) # *setting the seed (not essential)*

env = **create\_atari\_env**(params.env\_name) # *we create an optimized environment thanks to universe*

* However, this is *not the usual way to create an environment*. To improve the process and performance, we use the universe package from *OpenAI Gym*. universe provides optimized environments, enhancing efficiency.
* ***Shared model:*** Then we get our shared model by creating an object of the **ActorCritic** class. It's important to understand that this is the *model* that shared by the *different agents*. So we have different threads and different cores.

# *shared\_model is the model shared by the different agents (different threads in different cores)*

shared\_model = **ActorCritic**(env.observation\_space.shape[0], env.action\_space)

* And speaking of threads, the line **shared\_model.share\_memory**() stores the model in the *computer's shared memory*, allowing *all threads* to *access* it, even if they run on different cores.

# *storing the model in the shared memory of the computer,*

#  *which allows the threads to have access to this shared memory even if they are in different cores*

**shared\_model.share\_memory**()

* Then we get our optimizer **my\_optim.SharedAdam()**linking it to the shared model's parameters **shared\_model.parameters**() and with a learning rate of 0.001, lr=params.lr. And again it's important to understand that the *optimizer* is also *shared* because it's going to *act* on the *shared model*.
* So the optimizer is also shared, as it updates the shared model. The line **optimizer.share\_memory**() stores the *optimizer* in *shared* *memory*, allowing all agents to access it for model optimization.

# *the optimizer is also shared because it acts on the shared model*

optimizer = **my\_optim.SharedAdam**(**shared\_model.parameters**(), lr=params.lr)

# *same, we store the optimizer in the shared memory*

#  *so that all the agents can have access to this shared memory to optimize the model*

**optimizer.share\_memory**()

* ***Process:*** Now we initialize our processes, the *test process doesn't update* the *shared model* but it just uses it to try it in one part and print the score and record the videos. So that's exactly what is done with target=test, that's the test process and this process **mp.Process**() is from torch.multiprocessing. And what it does is that it basically runs a function on an independent thread.
* Then when we do **p.start**(), we start a new process which was just initialized by **mp.Process**().
* With this **processes.append**(p) we add the process in the list of the processes.

processes = [] # *initializing the processes with an empty list*

# *allowing to create the 'test' process with some arguments 'args' passed to the 'test' target function*

#  *the 'test' process doesn't update the shared model but uses it on a part of it*

#  *torch.multiprocessing.Process runs a function in an independent thread*

p = **mp.Process**(target=test, args=(params.num\_processes, params, shared\_model))

**p.start**() # *starting the created process p*

**processes.append**(p) # *adding the created process p to the list of processes*

# *making a loop to run all the other processes that will be trained by updating the shared model*

**for** rank **in** **range**(0, params.num\_processes):

    p = **mp.Process**(target=train, args=(rank, params, shared\_model, optimizer))

**p.start**()

**processes.append**(p)

* And finally in the FOR-loop we just do a loop to run all the other processes that will be trained by updating the shared model.
* Following loop will stop the program safely.

# *creating a pointer that will allow to kill all the threads when at least one of the threads,*

#  *or main.py will be killed, allowing to stop the program safely*

**for** p **in** processes:

**p.join**()

All code at once (Main code)

# *Main code*

**from** \_\_future\_\_ **import** print\_function

**import** os

**import** torch

**import** torch.multiprocessing **as** mp

**from** envs **import** create\_atari\_env

**from** model **import** ActorCritic

**from** train **import** train

**from** test **import** test

**import** my\_optim

# *Gathering all the parameters (that we can modify to explore)*

**class** Params():

**def** **\_\_init\_\_**(self):

        self.lr = 0.0001

        self.gamma = 0.99

        self.tau = 1.

        self.seed = 1

        self.num\_processes = 16

        self.num\_steps = 20

        self.max\_episode\_length = 10000

        self.env\_name = 'Breakout-v0'

#----  *Main run* ----

os.environ['OMP\_NUM\_THREADS'] = '1' # *1 thread per core*

params = **Params**() # *creating the params object from the Params class, that sets all the model parameters*

**torch.manual\_seed**(params.seed) # *setting the seed (not essential)*

env = **create\_atari\_env**(params.env\_name) # *we create an optimized environment thanks to universe*

# *shared\_model is the model shared by the different agents (different threads in different cores)*

shared\_model = **ActorCritic**(env.observation\_space.shape[0], env.action\_space)

# *storing the model in the shared memory of the computer,*

#  *which allows the threads to have access to this shared memory even if they are in different cores*

**shared\_model.share\_memory**()

# *the optimizer is also shared because it acts on the shared model*

optimizer = **my\_optim.SharedAdam**(**shared\_model.parameters**(), lr=params.lr)

# *same, we store the optimizer in the shared memory*

#  *so that all the agents can have access to this shared memory to optimize the model*

**optimizer.share\_memory**()

processes = [] # *initializing the processes with an empty list*

# *allowing to create the 'test' process with some arguments 'args' passed to the 'test' target function*

#  *the 'test' process doesn't update the shared model but uses it on a part of it*

#  *torch.multiprocessing.Process runs a function in an independent thread*

p = **mp.Process**(target=test, args=(params.num\_processes, params, shared\_model))

**p.start**() # *starting the created process p*

**processes.append**(p) # *adding the created process p to the list of processes*

# *making a loop to run all the other processes that will be trained by updating the shared model*

**for** rank **in** **range**(0, params.num\_processes):

    p = **mp.Process**(target=train, args=(rank, params, shared\_model, optimizer))

**p.start**()

**processes.append**(p)

# *creating a pointer that will allow to kill all the threads when at least one of the threads,*

#  *or main.py will be killed, allowing to stop the program safely*

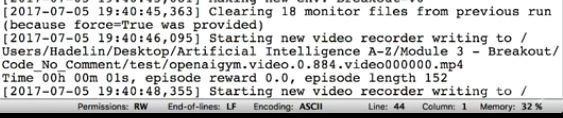
**for** p **in** processes:

**p.join**()

**4.15: Test the AI**

We'll run the **main.py** script to *train* the *AI*. We have *multiple scripts*, and the main script (main.py) will execute all the others. It *builds* the *neural network* and *trains* both the *agents* and the *critic*.

* Essentially, we'll *execute* the *script* to check its *progress*, but we won't wait for the results as the training takes several hours. The results have already been prepared in the 'test' folder, including a pre-recorded video that we'll show. The AI was trained for several hours to achieve this.
* Notice the videos are populated inside that folder. Inside the 'template' folder, we'll find a 'test' folder, which will be populated with videos. These videos showcase the results of our training if we let it run for a couple of hours.



* We'll watch the videos one by one, starting with the *first*, to see if the AI is playing Breakout and whether it managed to solve it.
* Note that the *first video* will be quite *poor*, as the paddle doesn't move at all. This is because, in the *early stages*, the AI has *no knowledge* of what to do and is still *pre-processing* the *images*. The same applies to the *second* and *third videos*.

|  |  |
| --- | --- |
| * In the 4th video, there is some activity. The AI is *moving* the *paddle* and *playing the game*, although not very effectively or quickly, indicating it is still a relatively basic AI. * In the 5th video, the AI is playing Breakout. While not perfect, it *performs similarly to a human player*, using some *weird tactics* like *quickly moving the paddle back and forth*. It achieves a decent score of around 27 or 30. In the 6th video, the AI improves further as the ball moves faster, managing to achieve a score of 30 and surpassing its previous performance. * In the 7th video, we see the A3C model in action. It performs better than a human player, playing for a longer duration (1 minute and 10 seconds) and achieving a high score of 78. The model uses *CNN*, *LSTM with memory*, and *shared memory* for its operation. * Notice in the 8th video, the playtime decreased to 51 seconds. We'll not get each time the best result and this is due to *exploration*—*AI doesn’t always produce the best result as it explores new strategies to learn*. As a result, improvements may not be evident in every video. The score in this video was 60. |  |

From other implementations of A3C, the version with LSTM performs better than others (other models last 15 to 20 seconds, while our implementation sustains play for over 50 seconds or even 1 minute). This demonstrates that we have a robust A3C model.

All code at once (test.py)

# *Test Agent*

**import** torch

**import** torch.nn.functional **as** F

**from** envs **import** create\_atari\_env

**from** model **import** ActorCritic

**from** torch.autograd **import** Variable

**import** time

**from** collections **import** deque

# *Making the test agent (won't update the model but will just use the shared model to explore)*

**def** **test**(rank, params, shared\_model):

**torch.manual\_seed**(params.seed + rank) # *asynchronizing the test agent*

    env = **create\_atari\_env**(params.env\_name, video=**True**) # *running an environment with a video*

**env.seed**(params.seed + rank) # *asynchronizing the environment*

    model = **ActorCritic**(env.observation\_space.shape[0], env.action\_space) # *creating one model*

**model.eval**() # *putting the model in "eval" mode because it won't be trained*

    state = **env.reset**() # *getting the input images as numpy arrays*

    state = **torch.from\_numpy**(state) # *converting them into torch tensors*

    reward\_sum = 0 # *initializing the sum of rewards to 0*

    done = **True** # *initializing done to True*

    start\_time = **time.time**() # *getting the starting time to measure the computation time*

    actions = **deque**(maxlen=100) # *cf https://pymotw.com/2/collections/deque.html*

    episode\_length = 0 # *initializing the episode length to 0*

**while** **True**: # *repeat*

        episode\_length += 1 # *incrementing the episode length by one*

**if** done: # *synchronizing with the shared model (same as train.py)*

**model.load\_state\_dict**(**shared\_model.state\_dict**())

            cx = **Variable**(**torch.zeros**(1, 256), volatile=**True**)

            hx = **Variable**(**torch.zeros**(1, 256), volatile=**True**)

**else**:

            cx = **Variable**(cx.data, volatile=**True**)

            hx = **Variable**(hx.data, volatile=**True**)

        value, action\_value, (hx, cx) = **model**((**Variable**(**state.unsqueeze**(0), volatile=**True**), (hx, cx)))

        prob = **F.softmax**(action\_value)

        action = **prob.max**(1)[1].**data.numpy**() # *the test agent does not explore, it directly plays the best action*

        state, reward, done, \_ = **env.step**(action[0, 0]) # *done = done or episode\_length >= params.max\_episode\_length*

        reward\_sum += reward

**if** done: # *printing the results at the end of each part*

**print**("Time {}, episode reward {}, episode length {}".**format**(**time.strftime**("%Hh %Mm %Ss", **time.gmtime**(**time.time**() - start\_time)), reward\_sum, episode\_length))

            reward\_sum = 0 # *reinitializing the sum of rewards*

            episode\_length = 0 # *reinitializing the episode length*

**actions.clear**() # *reinitializing the actions*

            state = **env.reset**() # *reinitializing the environment*

**time.sleep**(60) # *doing a one minute break to let the other agents practice (if the game is done)*

        state = **torch.from\_numpy**(state) # *new state and we continue*

All code at once (main.py)

# *Main code*

**from** \_\_future\_\_ **import** print\_function

**import** os

**import** torch

**import** torch.multiprocessing **as** mp

**from** envs **import** create\_atari\_env

**from** model **import** ActorCritic

**from** train **import** train

**from** test **import** test

**import** my\_optim

# *Gathering all the parameters (that we can modify to explore)*

**class** Params():

**def** **\_\_init\_\_**(self):

        self.lr = 0.0001

        self.gamma = 0.99

        self.tau = 1.

        self.seed = 1

        self.num\_processes = 16

        self.num\_steps = 20

        self.max\_episode\_length = 10000

        self.env\_name = 'Breakout-v0'

#----  *Main run* ----

os.environ['OMP\_NUM\_THREADS'] = '1' # *1 thread per core*

params = **Params**() # *creating the params object from the Params class, that sets all the model parameters*

**torch.manual\_seed**(params.seed) # *setting the seed (not essential)*

env = **create\_atari\_env**(params.env\_name) # *we create an optimized environment thanks to universe*

# *shared\_model is the model shared by the different agents (different threads in different cores)*

shared\_model = **ActorCritic**(env.observation\_space.shape[0], env.action\_space)

# *storing the model in the shared memory of the computer,*

#  *which allows the threads to have access to this shared memory even if they are in different cores*

**shared\_model.share\_memory**()

# *the optimizer is also shared because it acts on the shared model*

optimizer = **my\_optim.SharedAdam**(**shared\_model.parameters**(), lr=params.lr)

# *same, we store the optimizer in the shared memory*

#  *so that all the agents can have access to this shared memory to optimize the model*

**optimizer.share\_memory**()

processes = [] # *initializing the processes with an empty list*

# *allowing to create the 'test' process with some arguments 'args' passed to the 'test' target function*

#  *the 'test' process doesn't update the shared model but uses it on a part of it*

#  *torch.multiprocessing.Process runs a function in an independent thread*

p = **mp.Process**(target=test, args=(params.num\_processes, params, shared\_model))

**p.start**() # *starting the created process p*

**processes.append**(p) # *adding the created process p to the list of processes*

# *making a loop to run all the other processes that will be trained by updating the shared model*

**for** rank **in** **range**(0, params.num\_processes):

    p = **mp.Process**(target=train, args=(rank, params, shared\_model, optimizer))

**p.start**()

**processes.append**(p)

# *creating a pointer that will allow to kill all the threads when at least one of the threads,*

#  *or main.py will be killed, allowing to stop the program safely*

**for** p **in** processes:

**p.join**()