Chapter – 4 (implementation – Part 3 – Training Mechanism)

**Artificial Intelligence**

**AI-app (CNN-A3C) "Breakdown" Game play** (part 3)

ensureSharedGrad (explanation),

train()

synchronize with shared model

actions

update shared networks

policy loss and value loss

Training Mechanism:

Synchronization with Shared Model

**4.8: Train mechanism: ensureSharedGrad (explanation), train()**

Now we have the brain of the model and we also have the optimizer. Now we are ready to *train* our *different agents* (i.e. different brains of actor and critic).

In this part we will make a *big train function (not a class)* which will contain *all the A3C algorithm* in our train.py file. Later we'll apply this function in main.py file (the main code) at the final step.

* ***Libraries:*** First we import some libraries. First we import torch, classic libraries with the torch module and also functional module torch.nn.functional.
* We'll import *Atari environment* (create\_atari\_env) from the env, which will Break Out.
* Then we will of course import our ActorCritic class from our model.py file.
* And finally we will use the Variable from torch.autograd to run highly performing computations at the gradient (using dynamic graphs).

**import** torch

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

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

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

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

* ensure\_shared\_grads(): This is just a function that will make sure everything works correctly if the *model* used by the agent *doesn't have any shared gradients*.

# *Implementing a function to make sure the models share the same gradient*

**def** **ensure\_shared\_grads**(model, shared\_model):

**for** param, shared\_param **in** **zip**(**model.parameters**(), **shared\_model.parameters**()):

**if** shared\_param.grad **is** **not** **None**:

**return**

        shared\_param.\_grad = param.grad

* train(): This train function will take several arguments.
* rank: This parameter is just to shift the seed so that each *training agent is* ***desynchronized***.
* params: All the parameters of the environment.
* shared\_model: The shared model is what the agent will get to run its little exploration on a certain number of steps.
* optimizer: Is the optimizer that we've discussed earlier (shared ADAM optimizer).
* We know that ***A3C*** stands for it stands for ***Asynchronous Actor-Critic Agents*** (also called *Asynchronous Advantage Actor Critic*, it's a Multi-Agent Reinforcement Learning).
* So we have to *desynchronize* (disturb the synchronization) each *training agent* and to desynchronize them we're going to use the **rank** to *shift* each **seed** with the "rank" parameter
* So the **rank** parameter here is just to *shift the* ***seed*** so that each Training Agent is Desynchronized.
* For example if there is N *training agents* then the *ranks* will go from 1 to N and there will be *one integer per agent* from 1 to N.
* So when we *shift the* ***seed*** by *one* ***thread*** all the *pseudo random numbers* created by the thread will be totally independent from the other threads.
* ***Shifting the seed:*** However, the *seed* are fixed numbers. So when we *reproduce* the experience we will find *exactly the same events*. And that's because it's *deterministic* w.r. to the *seed*.
* That's why we need to do is *desynchronize* each training agents by shifting the seed with the rank.

# *shifting the seed with rank to asynchronize each training agent*

**torch.manual\_seed**(params.seed + rank)

* To do that we're going to use **torch.manual\_seed** from torch library.
* Now we're going to take the seeds of all the agents by using: params.seed
* Then we shift params.seed by rank to *desynchronize* each of those agents (i.e params.seed + rank).
* So **torch.manual\_seed**(params.seed + rank) will shift the seed with the rank to *desynchronize* each training agents because there is one seed for each training agent.
* ***Get the environment:*** The next step is to get the environment. The variable env gets the environment for Breakout by using create\_atari\_env() from the envs module.

# *creating an optimized environment thanks to the create\_atari\_env function*

env = **create\_atari\_env**(params.env\_name)

* We have to input just one argument which is the "parameters" of the environment. We get them from the "params" input of our train() function, therefore we've used params.env\_name. In the main.py it'll load the environment Breakout V0.
* ***Align the agent-seed:*** Now it's time to align the seed of the environment on the one of the agents. Why do we do that?
* It's because *each* ***agent*** *of the A3C-model* has its "own vision" of the environment like its own copy of the environment and therefore we need to *align* ***each*** *of the* ***agents*** on one *specific version* of the environment.
* To do that we're going to use the seed because *each seed* determines a *specific environment*.
* So by associating a *different seed* to *each agent* we'll get each agent will have its own environment.

# *aligning the seed of the environment on the seed of the agent*

**env.seed**(params.seed + rank)

* We use **env.seed()** to get the seed of a agent, and the we align it by setting params.seed + rank. That will align the seed of the environment to the seed of the agent.
* Because params.seed + rank corresponds to the *seed of the agent* that was *shifted by* ***rank*** to get desynchronized training agents (because they're all on a different seed).
* ***Getting the model:*** Now we get our model (i.e. the A3C brain) the ActorCritic class from model.py. We'll create a new object of this ActorCritic class and we're going to call this model.

# *creating the model from the ActorCritic class*

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

* This object will contain all the convolutions, LSTM, Linear-full-connection and the forward-function to propagate the signal. It will basically contain the *brains* of the *actor* and the *critic* with the ability to *propagate* the *signal* throughout the *brain* to get the *final* *output*.
* We have to input two arguments to this ActorCritic class. Those are actually the arguments of the \_\_init\_\_() function of the ActorCritic class: num\_inputs *(which is input shape i.e. the dimension of our input image)*, and action\_space *(contains the set of actions)*. We need to get these from our *imported environment* **env**.
* To get the *input-shape* from the environment we'll use env.observation\_space.shape[0].
* Similarly we'll use env.action\_space to get the *set of actions* from our environment.
* ***Prepare our input states:*** The next step is to prepare our *input states*. Remember, we're still doing *Deep Reinforcement Learning* so the *input states* are the *input images* and therefore
* This is actually a numpy-array which will contain *one channel* because we will work with *black and white* images and it will have dimensions of 42x42. But it's important to understand that the *input states* are *input images*.

# *state is a numpy array of size 1\*42\*42, in black & white*

state = **env.reset**()

* state is the variable for the input state. To get an numpy array we'll use **env.reset**(). This will initialize state as a numpy array of dimensions 1 x 42 x 42 (1 channel - black and white image and 42 x 42 is dimension of the image: ).
* ***Converting to torch-tensor:*** Now we have the *numpy-array*, in this step we will *convert* it into a *torch-tensor*. So we'll update the state again because we don't need the *numpy-array* anymore.
* We'll use **torch.from\_numpy**() and use the *numpy-array* state as an input argument (which we want to convert into a torch tensor).

# *converting the numpy array into a torch tensor*

state = **torch.from\_numpy**(state)

* ***done:*** We just need to initialize the done variable, that indicates if an *episode is over* or if the *game is over*. We'll initialize it to **True** to specify that this done variable will be equal to **True** when the game is done.

done = **True** # *when the game is done*

* That will be useful for later so that the AI doesn't play indefinitely to BreakOut.

**def** **train**(rank, params, shared\_model, optimizer):

**torch.manual\_seed**(params.seed + rank) # *shifting the seed with rank to asynchronize each training agent*

    env = **create\_atari\_env**(params.env\_name) # *creating an optimized environment thanks to the create\_atari\_env function*

**env.seed**(params.seed + rank) # *aligning the seed of the environment on the seed of the agent*

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

    state = **env.reset**() # *state is a numpy array of size 1\*42\*42, in black & white*

    state = **torch.from\_numpy**(state) # *converting the numpy array into a torch tensor*

    done = **True** # *when the game is done*

Now we've done basically the beginning of this **train()** function. The most important part here was that we have to desynchronize each training agents, is the one first principle of the A3C algorithm we have to apply.

In the next section, we will proceed to the synchronization with the shared model. Note that, there will be *different models* and the *shared model* is a model that all the agents share. And so we have to *synchronize* with the *shared model* so that each agent can get this shared model to proceed to a *small exploration* of a certain number of *steps*.

**4.9: Train mechanism: synchronize with shared model**

We are going to synchronize with the shared model in our **train()** function. First we're going to initialize the length of one episode.

We call this variable episode\_length, we first initialize it to 0, and later it'll be incremented. We'll use a WHILE-loop to do that.

# *initializing the length of an episode to 0*

episode\_length = 0

* ***While-loop:*** First we'll increment the episode\_length.

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

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

Next we are going to synchronize with the shared model. i.e. the agent will use the shared model to do its little *exploration* on a certain number of *steps*.

* ***Shared model:*** How we get this shared model? We'll use **load\_state\_dict**() on our **model** to get the *state dictionary* of our shared model.

# *synchronizing with the shared model*

# *the agent gets the shared model to do an exploration on num\_steps*

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

We apply **state\_dict**() over **shared\_model** to get the parameters of the shared model. That's how our **model** will get the **shared\_model** to do its little exploration.

* ***Two cases:*** Once the model gets this shared model now we have to *distinguish two cases*. The first one is "if the game is done", we have to re-initialize the *hidden-states* **hx** and the *cell-states* **cx** of the LSTM and the model.
* ***If the game is done (case 1):*** We're going to re-initialize them (**cx** and **hx**) with only zeroes, there will be a vector of 256 zeroes because remember, the *outputs* of the **LSTM** has the *dimension* is **1x256**.

# *if it is the first iteration of the while loop or if the game was just done, then:*

**if** done:

# *the cell states of the LSTM are reinitialized to zero*

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

# *the hidden states of the LSTM are reinitialized to zero*

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

* We'll use **torch.zeros()**, and 1, 256 as arguments. 1 for the vector and 256 is the number of elements (zeros).
* We will convert that into a torch Variable so that some gradients will be computed.
* ***Game playing (case 2):*** In that case, we're going to keep the old *cell-states* **cx** and *hidden-states* **hx**. We directly convert those into torch Variable s.

**else**:

# *we keep the old cell states, making sure they are in a torch variable*

cx = **Variable**(cx.data)

# *we keep the old hidden states, making sure they are in a torch variable*

hx = **Variable**(hx.data)

Now after dealing with those cases, we stay in the WHILE-loop because basically all the training process is going to happen in this loop.

* ***Initialize core variables:*** Now we're going to initialize several variables which are going to be at the *heart of the computations* in the *training*.

# *initializing the list of values (V(S))*

values = []

# *initializing the list of log probabilities*

log\_probs = []

# *initializing the list of rewards*

rewards = []

# *initializing the list of entropies*

entropies = []

* **values**: is the output of the critic i.e. the V(S) function.
* **log\_probs**: is the list of log probabilities.
* **rewards**: is the list of rewards.
* **rewards**: is the list of entropies (we never used it in our previous projects).

We've initialized these 4-variables with empty lists. Next we'll use a FOR-loop to update these variables.

* ***Update the core variables:*** Let's start a new FOR loop and this loop will update the values of above *four variables*. So this new FOR loop is going to be a *FOR loop over the exploration* steps and therefore we use step as our *looping variable*.

# *going through the num\_steps exploration steps*

**for** step **in** **range**(params.num\_steps):

* Because, params.num\_steps is the number of steps of the *exploration*.

|  |  |
| --- | --- |
| * For all the **steps** in the **exploration**, we're going to get the *predictions* of the model (i.e. what is returned by the model). * To get these *predictions* we can simply take the **model** and apply the *inputs* to it (that *input signal* goes through the brains & the model and that will get us the outputs). * Note that, we will get several outputs: * It will get us the values of the **V(S)** function which is the *output* of the *critic*. * Then the Q-values **Q(S, A)** which is the *output* of the *actor*. | Recall **forward()** from **model.py**  **def** **forward**(self, inputs):          inputs, (hx, cx) = inputs          x = **F.elu**(self**.conv1**(inputs))          x = **F.elu**(self**.conv2**(x))          x = **F.elu**(self**.conv3**(x))          x = **F.elu**(self**.conv4**(x))          x = **x.view**(-1, 32 \* 3 \* 3)          # *the LSTM takes as input x and the old hidden & cell states*  # *ouputs the new hidden & cell states*          hx, cx = self**.lstm**(x, (hx, cx))          # *getting the useful output: the hidden states*  # *(principle of the LSTM)*          x = hx          # *returning the output of the critic (V(S)),*  # *the output of the actor (Q(S,A)), and*  # *the new hidden & cell states ((hx, cx))*  **return** self**.critic\_linear**(x), self**.actor\_linear**(x), (hx, cx) |

* But remember, it will also output the *tuple* of the *hidden-states* and *cell-states*. If we go back to our **model.py**, in the **forward()** function we can see that indeed it returns the *output of the critic*- i.e. the *value of the V-function V(S)* then the *output of the actor* which are the *Q-values Q(S, A)* and also the *output of the LSTM* which is this tuple (hx, cx) the *hidden-states* hx and *cell-states* cx. So be careful because it's quite different than what we've done before.
* We're now going to apply the model to the inputs which is **state**. However, we cannot use it directly, we need to do several things related to torch.
* ***unsqueeze:*** First we need to *unsqueeze* the states to add the *fake dimension* that must have the index 0, that's because the model can only accept a *batch of inputs* and not an input by itself in a *Vector* or a *Tensor*.
* We need to convert our input states into a *torch Variable*.
* Now the *input state* is almost ok, but remember that the inputs of the *forward function* are actually the *input image* (that's what we just took care of) and also this *tuple* of (hx, cx) the *hidden-states* hx and *cell-states* cx and therefore we need to add this *tuple* as the *second part of the input*.
* ***So we have our two inputs:*** the *first one is* the input states (that is input images) all converted into torch-Variable and *unsqueezed* to add a *fake dimension* of the batch and the *second input* is the tuple of (hx, cx) the *hidden-states* hx and *cell-states* cx.

**model**((**Variable**(**state.unsqueeze**(0)), (hx, cx)))

* And now we are ready to get our predictions:

# *getting from the model the output V(S) of the critic,*

# *the output Q(S,A) of the actor, and the new hidden & cell states*

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

* **Unpacking:** Since **model**((**Variable**(**state.unsqueeze**(0)), (hx, cx))) returns three predictions: the output of the *critic*, the output of the *actor* and the *tuple* of the *hidden-states* hx and *cell-states* cx of the LSTM, we're going to store those three outputs into three new variables.
* **value:** gets the value of V(S) function,
* **action\_values:** gets the output of the actor, i.e. Q-values Q(S,A), since these are associated with the action, we call it action\_values.
* **(hx, cx):** the *tuple* of the *hidden-states* hx and *cell-states* cx of the LSTM

That’s how we get our three outputs returned by the model.

* ***Probabilities:*** After we got the predictions, we need to use a soft-Max to play the right action. So the next step is to get our probabilities.

# *generating a distribution of probabilities of the Q-values according to the softmax:*

# *prob(a) = exp(prob(a))/sum\_b(exp(prob(b)))*

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

We've used **softmax**() from the functional module **F**. It will generate a *distribution of probabilities* of the outputs of the actor (i.e. action\_values which are the Q-values).

* ***Entropy:*** Now we have to work with the entropy. To get the entropy, we not only need the probabilities but also the LOG-probabilities because the entropy is the *sum* of the *product* of *probabilities* and *LOG-probabilities* multiplied by -1.
* *Log-probability:* Log-probability is generated from **log\_oftmax**() from the functional module **F**. We apply it to our Q-values i.e. action\_values. So instead of taking a *distribution of the probabilities* we take a *distribution of the log-probabilities*.

# *generating a distribution of log probabilities of the Q-values according to the log softmax:*

# *log\_prob(a) = log(prob(a))*

log\_prob = **F.log\_softmax**(action\_values)

* *Entropy****:*** Now we get the ***entropy*** using the *probability* prob and *log-probability* log\_prob. The formula for entropy is:

entropy = -(log\_prob \* prob).**sum**(1) # *H(p) = - sum\_x p(x).log(p(x))*

Here we use **sum(1)**: This indicates summing over the second dimension.

* Finally we append this entropy to our list of entropies.

**entropies.append**(entropy) # *storing the computed entropy*

That's how we store the last computed entropy in the list of entropies.

In the next section we'll play the action by taking a random draw of this generated distribution of probabilities. And *after* we play the *action* we'll get the *value* of the **state** and we will eventually *store* our new transitions: **state**, **reward** and **done**.

Entropy in Classical Information Theory

In classical information theory, *entropy* measures the *uncertainty* or *unpredictability* of a *random variable*. It tells us *how much information* we need to *describe the outcome* of the variable.

* ***Random Variable:*** Think of a random variable as something that can take different values (outcomes) with certain probabilities. For example, rolling a die where each side (1 to 6) has a 1/6 chance of showing up.
* ***Entropy:*** Entropy tells us *how uncertain we are* about the *outcome*. If all outcomes are equally likely (like rolling a fair die), the uncertainty (or entropy) is high. If one outcome is much more likely (like a loaded die always showing 6), the uncertainty is low, so the entropy is lower.
* In information theory, the *entropy of a random variable* quantifies the *average* ***level*** *of* ***uncertainty*** or ***information*** associated with the variable's *potential states* or *possible outcomes*.
* This measures the *expected amount of information needed to describe the state of the variable*, considering the *distribution of probabilities* across all *potential states*.
* Given a discrete random variable , which takes values in the set and is distributed according to , the entropy (H) is calculated using this formula:

Note: In a mathematical statement, the symbol "**:=**" means "is defined as" or "is assigned as"

* Where denotes the sum over the variable's possible values.
* is the probability of a specific outcome .
* The logarithm **log** measures the *amount of information* each outcome provides. The choice of *base* for the logarithm changes the *unit of* *measurement* (**bits** if base 2, **nats** if base e, etc.).
* The choice of base for **log**, the logarithm, varies for different applications. *Base 2* gives the **unit of bits** (or "**shannons**"), while *base e* gives "**natural units**" **nat**, and *base 10* gives units of "**dits**", "**bans**", or "**hartleys**".
* Entropy is the *average amount of information* you would need to *describe* the *outcome of a random process*, considering the *probability* of *each possible* result. For example:
* A *fair coin* (heads or tails, both with 50% probability) has *higher entropy* than a *biased coin* that lands heads 99% of the time because the outcome of the *fair coin* is *more uncertain*.
* An equivalent definition of ***entropy*** is the ***expected value*** *of the* ***self-information*** *of a* ***variable***.

The concept of *information entropy* was introduced by *Claude Shannon* in his 1948 paper "A Mathematical Theory of Communication", and is also referred to as *Shannon entropy*.

|  |  |
| --- | --- |
| * Does it mean "more entropy equals more information"?   *More entropy means more uncertainty*, *not necessarily more useful information*.   * Entropy measures the uncertainty or unpredictability in a *random process*. When the *entropy is high*, it means the outcome is *harder to predict* because all possible outcomes are almost equally likely. This means more bits or information are needed to describe the outcome. * *Information* in this context refers to the *amount of detail* or *specificity* needed to *describe the outcome*. More entropy implies that when an event happens, you need to provide more information to specify what actually occurred. * ***High Entropy:*** When entropy is high (e.g., a fair die), there's a lot of uncertainty, so every outcome provides more information because you didn't know what would happen in advance. * ***Low Entropy:*** When entropy is low (e.g., a biased die that always lands on 6), there's less uncertainty, and you don't need much information to describe the outcome (you mostly expect 6). | FIG 1: 2-coin entropy (wikipedia) |

**train()** function: as far as we’ve built it up until now

**def** **train**(rank, params, shared\_model, optimizer):

**torch.manual\_seed**(params.seed + rank) # *shifting the seed with rank to asynchronize each training agent*

    env = **create\_atari\_env**(params.env\_name) # *creating an optimized environment thanks to the create\_atari\_env function*

**env.seed**(params.seed + rank) # *aligning the seed of the environment on the seed of the agent*

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

    state = **env.reset**() # *state is a numpy array of size 1\*42\*42, in black & white*

    state = **torch.from\_numpy**(state) # *converting the numpy array into a torch tensor*

    done = **True** # *when the game is done*

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

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

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

# *synchronizing with the shared model - agent gets the shared model to do an exploration on num\_steps*

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

**if** done: # *if it is the first iteration of the while loop or if the game was just done, then:*

            cx = **Variable**(**torch.zeros**(1, 256)) # *the cell states of the LSTM are reinitialized to zero*

            hx = **Variable**(**torch.zeros**(1, 256)) # *the hidden states of the LSTM are reinitialized to zero*

**else**: # *else:*

            cx = **Variable**(cx.data) # *we keep the old cell states, making sure they are in a torch variable*

            hx = **Variable**(hx.data) # *we keep the old hidden states, making sure they are in a torch variable*

        values = [] # *initializing the list of values (V(S))*

        log\_probs = [] # *initializing the list of log probabilities*

        rewards = [] # *initializing the list of rewards*

        entropies = [] # *initializing the list of entropies*

**for** step **in** **range**(params.num\_steps): # *going through the num\_steps exploration steps*

# *getting from the model the output V(S) of the critic,*

            # *the output Q(S,A) of the actor, and the new hidden & cell states*

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

            # *generating a distribution of probabilities of the Q-values according to the softmax:*

            # *prob(a) = exp(prob(a))/sum\_b(exp(prob(b)))*

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

            # *generating a distribution of log probabilities of the Q-values according to the log softmax:*

            # *log\_prob(a) = log(prob(a))*

            log\_prob = **F.log\_softmax**(action\_values)

            entropy = -(log\_prob \* prob).**sum**(1) # *H(p) = - sum\_x p(x).log(p(x))*

**entropies.append**(entropy) # *storing the computed entropy*

**4.10: Train mechanism: actions**

We just computed the entropy and added it to the entropies list. Now we take a *random draw* of an *action* according to the *distribution of probabilities* of the *softmax*.

* ***Random action:*** We are still in the FOR-loop and still running on the steps (i.e. loop variable step). We take our distribution of probabilities prob and use multinomial() function to take a random draw from this distribution of probabilities and then we get that data. We store it to the action variable, to play the action.
* Note that the action will actually be a *tensor with only one value* so instead of seeing this as simple values we should see this as a *tensor of damnations* 1x1 that contains this value for the action (because it isn't squeezed).

# *selecting an action by taking a random draw from the prob distribution*

action = **prob.multinomial**().data

* ***Update Log probability:*** Now we update the *log probability* associated to the *action* that was just played.

# *getting the log prob associated to this selected action*

log\_prob = **log\_prob.gather**(1, **Variable**(action))

* We take the previous **log\_prob** that we computed before and then use the **gather()** method.
* In the second argument we used **action** (converted to a *Torch Variable* because the **gather()** function just indexes with a tensor integer), in the first argument 1 is used to get the log probability that is associated to this action.
* ***Update the lists:*** Since we got the **log\_prob** associated to the action that was just played, now it's time to append those **value** (output of the **model()**) and to our **log\_prob** lists. We already append the **entropy** to the **entropies** list, and we'll get the rewards afterwards (so we'll append it later).

**values.append**(value) # *storing the value V(S) of the state*

**log\_probs.append**(log\_prob) # *storing the log prob of the action*

* ***Playing the action:*** We selected the action by taking a *random draw* from the *distribution of probabilities*, but we actually haven't played it yet. We're going to play it now so that we can *reach* the *new state* and get the *new transition*.

# *playing the selected action, reaching the new state, and getting the new reward*

state, reward, done, \_ = **env.step**(**action.numpy**())

* To play it, we'll use our environment **env** (because the action is played in the environment) then we're going to use the **step**() method and the action **action.numpy**() as parameter.
* This will output new state, new reward, (by reaching a state we got new reward) and done value (if the game is done or not). So with this we play the action and reach a new state, we get a reward and also if we're done with the game or not.
* ***Checking game over:*** We're going to add a *condition* that will make sure that an agent is not *stucked* in some state.

# *if the episode lasts too long (the agent is stucked), then it is done*

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

* The condition is saying that the *episode* of the game should not last too much time. Also in the main function there will be a max\_episode\_length parameter which will be equal to **10000** (we'll get it from our parameters **prams**). And we don't want an episode to last more than 10000 units.
* So if the game is done or the length of the episode is larger than the maximum length of episode (which we'll set to 10000), the game will be done and we will start a new game.
* ***Clamping the reward:*** Clamp the reward between -1 and +1. We already *got the reward* but we need to make sure that the reward is between -1 and +1. To do this we simply need to update the reward by doing this:

# *clamping the reward between -1 and +1*

reward = **max**(**min**(reward, 1), -1)

* Above code limits the value of reward between -1 and +1. Here's how it works step-by-step:
* **min**(reward, 1) takes the smaller value between **reward** and **1**. If reward is greater than 1, it will return 1. Otherwise, it returns reward.
* **max**(**min**(reward, 1), -1): The result from **min**(reward, 1) is passed to **max**, which compares this result with -1. If the result from **min**(reward, 1) is less than -1, it will return -1. Otherwise, it returns the previous result (which is already between -1 and 1).
* The purpose of this operation is to ensure that the reward value is restricted to the range [-1, 1]. So:
* If reward > 1, it will be clamped to 1.
* If reward < -1, it will be clamped to -1.
* If reward is already between -1 and 1, it remains unchanged.

This is commonly used in *Reinforcement Learning* to *normalize* or limit rewards within a specific range.

* ***Check the episode:*** We'll use **done** again to check if the episode is over, in that case we will restart the environment.

**if** done: # *if the episode is done:*

episode\_length = 0 # *we restart the environment*

state = **env.reset**() # *we restart the environment*

* We need to check that, because we just reached a *new state*. We just passed a *new transition*. So we need to check that after passing this *new transition*, the game is not done. If it's done then in that case we will *restart* the *environments* by setting the *episode length* to zero. And also the **state** will be *re-initialized*.
* ***Tensorizing new state:*** Since we reached a *new state* and it is right now an *numpy array* because remember the states are the *input* *images* which originally are *numpy arrays*. So we need to convert the *new state* into a ***Torch Tensor***. We update our **state** as below:

state = **torch.from\_numpy**(state) # *tensorizing the new state*

* We'll use the **from\_numpy**() function to convert this numpy array **state** (the input images) into a torch tensor.
* ***Update rewards list:*** Before getting out of this FOR-loop (that is the loop on our steps: **step**), we'll append the **reward** to the **rewards** list.
* We've updated all the lists (i.e. **values**, **log\_probs**, **entropies**) here except for the **rewards**.

**rewards.append**(reward) # *storing the new observed reward*

Append the last reword that was just received.

* ***Breaking the loop:*** Just before we get out of the FOR-loop we need to check that if it's done once again, if it is then we stop the exploration. We're simply going to apply a break.

**if** done: # *if we are done*

# *we stop the exploration and we directly move on*

# *to the next step: the update of the shared model*

**break**

* If it's done we *stop the exploration* and we directly move on to the *next step* which will be the *update of the shared model* and now we are done with this FOR-loop.
* Now that the *agent* has done its *exploration*, it will *update* the *shared model* and we will take care of that in the next section.

The entire FOR-loop (section 4.9 & 4.10)

# *initializing the list of values (V(S))*

values = []

# *initializing the list of log probabilities*

log\_probs = []

# *initializing the list of rewards*

rewards = []

# *initializing the list of entropies*

entropies = []

# *going through the num\_steps exploration steps*

**for** step **in** **range**(params.num\_steps):

# *getting from the model the output V(S) of the critic,*

# *the output Q(S,A) of the actor, and the new hidden & cell states*

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

# *generating a distribution of probabilities of the Q-values according to the softmax:*

# *prob(a) = exp(prob(a))/sum\_b(exp(prob(b)))*

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

# *generating a distribution of log probabilities of the Q-values according to the log softmax:*

# *log\_prob(a) = log(prob(a))*

log\_prob = **F.log\_softmax**(action\_values)

entropy = -(log\_prob \* prob).**sum**(1) # *H(p) = - sum\_x p(x).log(p(x))*

**entropies.append**(entropy) # *storing the computed entropy*

# *selecting an action by taking a random draw from the prob distribution*

action = **prob.multinomial**().data

# *getting the log prob associated to this selected action*

log\_prob = **log\_prob.gather**(1, **Variable**(action))

**values.append**(value) # *storing the value V(S) of the state*

**log\_probs.append**(log\_prob) # *storing the log prob of the action*

# *playing the selected action, reaching the new state, and getting the new reward*

state, reward, done, \_ = **env.step**(**action.numpy**())

# *if the episode lasts too long (the agent is stucked), then it is done*

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

# *clamping the reward between -1 and +1*

reward = **max**(**min**(reward, 1), -1)

**if** done: # *if the episode is done:*

episode\_length = 0 # *we restart the environment*

state = **env.reset**() # *we restart the environment*

state = **torch.from\_numpy**(state) # *tensorizing the new state*

**rewards.append**(reward) # *storing the new observed reward*

**if** done: # *if we are done*

# *we stop the exploration and we directly move on*

# *to the next step: the update of the shared model*

**break**

**4.11: Train mechanism: update shared networks**

We're now *outside* the *FOR-loop* but still *inside* the *WHILE-loop*. Now the *agent* has done its *exploration* and then what it's about to do is to *update* the *shared network*.

* ***Initialize cumulative reward:*** The first thing we're going to do is *initialize* the *cumulative reward* we're going to call it R. We'll initialize R as a *torch tensor* but that will have *dimensions* 1x1 because it's just a value but we wanted to be a *tensor*.

R = **torch.zeros**(1, 1) # *intializing the cumulative reward*

* So we're using **zeros**(1, 1). So basically the *cumulative reward* is *initialized* to 0.
* ***Update R:*** If we're not done (i.e. if the game is not over), we set the *cumulative reward* to be equal to the *value of the last state* reached by the shared network.

# *if we are not done: we initialize the cumulative reward with the value of the last shared state*

**if** **not** done:

value, \_, \_ = **model**((**Variable**(**state.unsqueeze**(0)), (hx, cx))) # *value of the last shared state*

R = value.data # *initialize the cumulative reward with the value of the last shared state*

* So we're going to get the **value** output. (i.e. the value of the **V(S)** function outputs of our model) and we will give this value to the *cumulative reward* R.
* To get this value we'll use value, \_, \_, because we'll use only the first output from the **model**(). The \_, \_ are used for action\_values, (hx, cx) (recall section 4.9 ,*Update the core variables*).
* We used the same inputs (**Variable**(**state.unsqueeze**(0)), (hx, cx)) (as we did in section 4.9), to get the value. The *first one is* the input states (that is input images) all converted into torch-Variable and *unsqueezed* to add a *fake dimension* of the batch and the *second input* is the tuple of (hx, cx) the *hidden-states* hx and *cell-states* cx.
* We then give this value to R, to access the value of the value we've used value.data. Now the if-condition is done.
* ***Updating the*** values ***list:*** Since we just got a new value by getting the output (the first output) of the **model**(), so we now append this new value R to the **values** list. Notice we need to convert R to the torch-Variable first.

# *storing the value V(S) of the last reached state S*

**values.append**(**Variable**(R))

* **policy\_loss**, **value\_loss**: Now we are going to initialize the *losses* and remember we have two losses:
* The *loss of the policy* that is the loss related to the *predictions of the* ***agent***.
* The *loss of the value* which is loss related to the *predictions of the* ***critic***.
* So we are going to introduce these two variables **policy\_loss**, **value\_loss** and initialize them into zero.

policy\_loss = 0 # *initializing the policy loss*

value\_loss = 0 # *initializing the value loss*

* We also make sure that the *cumulative reward* R is a torch-Variable, so we make the following conversion:

R = **Variable**(R) # *making sure the cumulative reward R is a torch Variable*

* Because we will be computing a *gradient* with respect to it, and the *cumulative reward* is going to be a term of the *value loss*. After being a torch-Variable, now it's attached to the dynamic graphs with a gradient.
* ***GAE:*** Finally, the last thing we need to do before starting the *big training loop* (where we apply *Stochastic Gradient Descent* to reduce the *loss* between *predictions* and *targets*) is to initialize the *GAE: Generalized Advantage Estimation*.
* Don't confuse it with an Auto-Encoder. Be careful—this **GAE** variable that we're about to initialize refers to *Generalized Advantage* *Estimation*.
* Note that, *Generalized Advantage Estimation* is by definition the *advantage* of *playing* the *action "a"* by *observing* the *state "s"*. So it's a function of the *action "a"* and the *state "s"* and it is equal to the *difference* between the *Q-values Q(a, s)* and the value of the *V-function V(s)*.
* *Generalized Advantage Estimation:*
* We want to initialize it to 0, and it also needs to be a *torch tensor*, so we're going to use the same method we used to initialize the *cumulative reward R*. We'll apply the **zeros()** function to set it as a tensor with a *single value* of 0.

# *Generalized Advantage Estimation: A(a,s) = Q(a,s) - V(s)*

gae = **torch.zeros**(1, 1) # *initializing the Generalized Advantage Estimation to 0*

* So it will be initialized to zero, and therefore the Q-values of the *action a* and the *state s* will be equal to the value of the *V-function of the state s* i.e.

Next we are ready to start the FOR-loop.

R = **torch.zeros**(1, 1) # *intializing the cumulative reward*

**if** **not** done: # *if we are not done:*

            value, \_, \_ = **model**((**Variable**(**state.unsqueeze**(0)), (hx, cx))) # *value of the last shared state*

            R = value.data # *we initialize the cumulative reward with the value of the last shared state*

**values.append**(**Variable**(R)) # *storing the value V(S) of the last reached state S*

        policy\_loss = 0 # *initializing the policy loss*

        value\_loss = 0 # *initializing the value loss*

        R = **Variable**(R) # *making sure the cumulative reward R is a torch Variable*

# *Generalized Advantage Estimation: A(a,s) = Q(a,s) - V(s)*

        gae = **torch.zeros**(1, 1) # *initializing the Generalized Advantage Estimation to 0*

**4.12: Train mechanism: policy loss and value loss**

Now we're going to make a *FOR-loop* that'll compute the **policy\_loss** and the **value\_loss** and once we have these two losses, we'll be able to use our optimizer to apply the *stochastic gradient descent* to reduce the losses.

* ***For-loop:*** We're going to start from the *last step* that was *done during the exploration* and we're going to move *backward* in time.

# *starting from the last exploration step and going back in time*

**for** i **in** **reversed**(**range**(**len**(rewards))):

* That's why we're going to use **reversed**(**range**(**len**(rewards))).
* Because rewards is the list and since *each step of the exploration* is associated to a *reward* because at each step we get reward when **len**(rewards) is this *number of steps*. We used **reversed**() so that we can *move back* in time.
* ***Update the cumulative reward:*** Now we're going to *update* the *cumulative reward* (i.e. R ) it's actually the same as what we did for Doom.

# *R = gamma\*R + r\_t*

# *= r\_0 + gamma \* r\_1 + gamma^2 \* r\_2 ... + gamma^(n-1)\*r\_(n-1) + gamma^nb\_step \* V(last\_state)*

R = params.gamma \* R + rewards[i]

* gamma: which we get from our parameters param.
* R *: cumulative reward*
* rewards[i]: reward of the steps. Since we go reverse, i is the *reward of the last step*, then in the next iteration it'll be *reward of the previous step* and so on.

So each time we update R By multiplying it by *gamma* and then adding this *reward at the step*.

* By doing this, the community reward will be equal at the end of the current FOR-loop:
* We'll get that because after the end of *previous FOR-loop* we updated the value of the *last step* and we set to be equal to this value.

So that's the main thing to understand this computation of the cumulative reward.

* ***Advantage:*** Now we have the right value for the *cumulative reward* so it's time to compute the advantage. It's just the advantage of getting this reward compared to value. It will be equal to this cumulative reward R minus the value of the V(s) function obtained at the step i.

# *R is an estimator of Q at time t = i so advantage\_i = Q\_i - V(state\_i) = R - value[i]*

advantage = R - values[i]

* ***Value loss:*** Now we compute the *value loss* (the first loss) using the *cumulative reward* and the *advantage*. **value\_loss** was initialized to zero, we update it as below:

# *computing the value loss*

value\_loss = value\_loss + 0.5 \* **advantage.pow**(2)

* **pow**(2) is used to square the advantage.

It's the loss generated by the *predictions* of the value of the *V-function* output by the critic.

* It makes sense that this is the value-loss because remember the advantage of the action a and the state s is the *difference* between the *Q-value* and the *value of the V-function*.
* So when we play the *optimal action*, well we get the stationary state with Q\*(a\*, s) of optimal action a\* played in the state s equals the optimal value V\*(s) of the state s.

Q\*(a\*, s) = V\*(s)

* optimal Q = Q\*
* optmal value V = V\*
* optimal action = a\*
* So it's quite intuitive to understand that when the *advantage is not equal to zero* then there will be a difference between these Q\*(a\*, s) and V\*(s). That's how the loss is measured.
* ***Policy loss:*** To compute it, we need to consider again the *Generalized Advantage Estimation* and to get it we first need the *temporal difference* of the *state values*. Once we get the *temporal difference*, we'll get the *generalized advantage estimation* and once we get the generalized advantage estimation we will get the policy loss.
* ***Temporal difference:*** We name it TD, which is equal to the *reward* of the step i plus *gamma times the value* of the step i+1 and minus the *value* of the step i. That's the formula of the temporal difference and the state values.

# *computing the temporal difference*

TD = rewards[i] + params.gamma \* values[i + 1].data - values[i].data

* ***gae:*** Now we can update the generalized advantage estimation (gae). We multiply it by parameters **gamma** and **tau** and then add the **TD**.

# *gae = sum\_i (gamma\*tau)^i \* TD(i) with gae\_i*

# *= gae\_(i+1)\*gamma\*tau + (r\_i + gamma\*V(state\_i+1) - V(state\_i))*

gae = gae \* params.gamma \* params.tau + TD

* We are in the a FOR-loop and each time we multiply the gae by **gamma** and **tau** and we add TD (temporal difference). So it's important to understand that at the end of this FOR-loop this generalized advantage estimation gae will be equal to (for all the step i):
* ***Policy loss:*** Now that we have the generalized advantage estimation gae and the temporal difference TD we can finally compute the policy loss.

# *computing the policy loss*

policy\_loss = policy\_loss - log\_probs[i] \* **Variable**(gae) - 0.01 \* entropies[i]

* We multiply the *log-probabilities* obtained at the step i with generalized advantage estimation gae (transformed to Variable) and subtract that from the policy\_loss then we add *minus 0.1 times the entropy* (the entropy obtained at the step i in the FOR-loop).
* Now be careful this computation is inside the FOR-loop which means that at the end of the FOR-loop what you'll get is
* Policy of step i is the *softMax probabilities of the actions* and
* Entropy of the step i we've computed earlier and we appended to the list.
* ***Policy Loss Calculation:*** The *policy loss* is calculated by weighting the *log probabilities of actions* by the *advantage function* (in this case, *Generalized Advantage Estimation (GAE)*). The GAE term represents the *advantage of taking an action* at each *time step*, which guides the *policy* to favor *actions* that lead to better-than-expected outcomes.
* The *entropy* term is typically added as a regularization term to encourage *exploration* by maximizing entropy, meaning the *policy doesn't get too deterministic* and can still *explore alternative actions*.

Where:

policy\_loss is the accumulated loss over time steps.

log\_probs[i] represents the *log probability* of taking action at time step under the current policy.

GAE is the *advantage* calculated using *Generalized Advantage Estimation* for each step.

entropies[i] is the entropy of the policy at step i, providing regularization to encourage exploration.

above code implies that for each *time step* , we compute

policy\_loss = policy\_loss - **Variable**(gae) - 0.01 \* entropies[i]

* The final policy loss formula, after summing over all steps, is:
* In practice, policy gradient methods (such as Proximal Policy Optimization (PPO) and A2C/A3C) use this type of loss formulation to update the policy. The negative sign is because most frameworks perform minimization rather than maximization. So, to maximize the reward, we minimize the negative of the policy gradient.
* is the probability of action ​ under the policy .
* is computed for each time step i and represents the *advantage* at that step.
* is an entropy bonus term that encourages exploration, controlled by a small coefficient (here, 0.01).
* Policy of step i i.e. refers to the *softmax probabilities* of the actions.
* In Reinforcement Learning , represents the *policy* that the agent follows, which is a mapping from . Specifically:
* is the policy function, which provides the *probability distribution* over *possible actions* given a *particular state*.
* are the parameters of this policy, typically the *weights* of a *neural network* that parameterize the function.

So, denotes the probability of taking action given state under the policy parameterized by .

* ***Reason of using negative sign:*** Since the *log of the probability* and the *entropy* are *negative* values and since we want to *minimize their absolute value* we must see this loss as the *log likelihood* as opposed to a *distance*.
* We want to maximize the probability of playing the action that will maximize the advantage. That's the whole idea behind it.
* ***The purpose of this entropy coefficient i.e. this factor 0.01:*** The purpose of it is just to *prevent* it from *falling too quickly* into a *trap* where we have a distribution of *probabilities with zeros* for *all* the *actions* except one which has a probability of one.
* And if that happens that would minimize the entropy.
* So that's why we're adding this small coefficient *0.01* here that will make the *entropy increase* in the gradient descent.
* ***SGD:*** Now we have the two losses so it's time to perform stochastic gradient descent to reduce these two losses. So we exit the FOR-loop. We initialize the optimizer and then we do the backward propagation.
* ***Initialize the optimizer:*** We take our optimizer and initialize all the gradient parameters to zero. To do this we use **zero\_grad()**.

# *initializing the optimizer*

**optimizer.zero\_grad**()

* ***Backward propagation:*** Now we're going to perform backward propagation but we're going to give twice importance to the *policy loss* than the *value loss* because the **policy\_loss** is smaller.
* We apply the backward() method to perform backward propagation and thanks to this trick here (policy\_loss + 0.5 \* value\_loss) that we have twice as much importance to the policy\_loss than the value\_loss.

# *we give 2x more importance to the policy\_loss than the value\_loss*

# *because the policy loss is smaller*

(policy\_loss + 0.5 \* value\_loss).**backward**()

* ***Clip the gradient:*** We're going to use another trick to prevent the gradient from taking extremely large values and degenerates the algorithm.

# *clamping the values of gradient between 0 and 40*

# *to prevent the gradient from taking huge values and degenerating the algorithm*

**torch.nn.utils.clip\_grad\_norm**(**model.parameters**(), 40)

* It'll make sure that the *norm* of the *gradient* stays between 0 and 40. That's how we prevent the gradient from taking to large values.
* ***Shared gradient:*** Remember at the beginning of the train.py file we made a function **ensure\_shared\_grads()** to ensure that the *agent* and the *shared model* share the *same gradients*. Now it's time to apply this function.

# *making sure the model of the agent and the shared model share the same gradient*

**ensure\_shared\_grads**(model, shared\_model)

* To make sure that the model and the shared\_model share the same gradients, we apply those as parameters. (It's just a precaution maybe that's not totally necessary but it makes sure that we won't get any issue.)
* ***Optimization step:*** Finally we perform the *optimization step* to reduce the losses.

# *running the optimization step*

**optimizer.step**()

|  |
| --- |
| * And now the training step of our AI-brains is over. Don't worry if you haven't fully understood everything yet, as this is an advanced concept (and it’s highly powerful, developed by the creators of PyTorch). * It's completely normal not to grasp everything on the first attempt, but with practice and repeated effort, you'll become more comfortable and confident over time. * We've now completed the training process. So far, we've accomplished the most important tasks: building the 'brain' by designing the *neural network architecture*, including *convolutions*, *LSTM*, and *fully connected layers*. We *trained* *this 'brain'* using the training code in this section. Essentially, the core of the algorithm is complete. So, congratulations!! You’ve successfully implemented the A3C model! * Now, we have a few more steps to complete, but this is where the fun part begins. We need to create a test.py file, which will test the AI agent and *generate videos* of the AI playing the game Breakout. Watching these videos will be very exciting. * We won’t need to write every line of code in the test.py file because, as mentioned earlier, we’ve already completed the most important parts of A3C. * We also have a main.py file, which will tie everything together and execute the process. Once we run main.py, it will generate the *AI* *brain*, handle the *training*, and allow the AI to *play new games* of Breakout. The videos of the gameplay will be generated automatically.   In the next section, we’ll focus on the test.py file to test the AI on new games. |

update shared networks and losses (section 4.11 & 4.12)

R = **torch.zeros**(1, 1) # *intializing the cumulative reward*

# *if we are not done: we initialize the cumulative reward with the value of the last shared state*

**if** **not** done:

value, \_, \_ = **model**((**Variable**(**state.unsqueeze**(0)), (hx, cx))) # *value of the last shared state*

R = value.data # *initialize the cumulative reward with the value of the last shared state*

# *storing the value V(S) of the last reached state S*

**values.append**(**Variable**(R))

policy\_loss = 0 # *initializing the policy loss*

value\_loss = 0 # *initializing the value loss*

R = **Variable**(R) # *making sure the cumulative reward R is a torch Variable*

# *Generalized Advantage Estimation: A(a,s) = Q(a,s) - V(s)*

gae = **torch.zeros**(1, 1) # *initializing the Generalized Advantage Estimation to 0*

# *starting from the last exploration step and going back in time*

**for** i **in** **reversed**(**range**(**len**(rewards))):

# *R = gamma\*R + r\_t*

# *= r\_0 + gamma \* r\_1 + gamma^2 \* r\_2 ... + gamma^(n-1)\*r\_(n-1) + gamma^nb\_step \* V(last\_state)*

R = params.gamma \* R + rewards[i]

# *R is an estimator of Q at time t = i so advantage\_i = Q\_i - V(state\_i) = R - value[i]*

advantage = R - values[i]

# *computing the value loss*

value\_loss = value\_loss + 0.5 \* **advantage.pow**(2)

# *computing the temporal difference*

TD = rewards[i] + params.gamma \* values[i + 1].data - values[i].data

# *gae = sum\_i (gamma\*tau)^i \* TD(i) with gae\_i*

# *= gae\_(i+1)\*gamma\*tau + (r\_i + gamma\*V(state\_i+1) - V(state\_i))*

gae = gae \* params.gamma \* params.tau + TD

# *computing the policy loss*

policy\_loss = policy\_loss - log\_probs[i] \* **Variable**(gae) - 0.01 \* entropies[i]

# *initializing the optimizer*

**optimizer.zero\_grad**()

# *we give 2x more importance to the policy\_loss than the value\_loss*

# *because the policy loss is smaller*

(policy\_loss + 0.5 \* value\_loss).**backward**()

# *clamping the values of gradient between 0 and 40*

# *to prevent the gradient from taking huge values and degenerating the algorithm*

**torch.nn.utils.clip\_grad\_norm**(**model.parameters**(), 40)

# *making sure the model of the agent and the shared model share the same gradient*

**ensure\_shared\_grads**(model, shared\_model)

# *running the optimization step*

**optimizer.step**()

The entire Train mechanism (section 4.9, 4.10, 4.11 & 4.12)

**import** torch

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

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

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

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

# *Implementing a function to make sure the models share the same gradient*

**def** **ensure\_shared\_grads**(model, shared\_model):

**for** param, shared\_param **in** **zip**(**model.parameters**(), **shared\_model.parameters**()):

**if** shared\_param.grad **is** **not** **None**:

**return**

        shared\_param.\_grad = param.grad

**def** **train**(rank, params, shared\_model, optimizer):

# *shifting the seed with rank to asynchronize each training agent*

**torch.manual\_seed**(params.seed + rank)

# *creating an optimized environment thanks to the create\_atari\_env function*

    env = **create\_atari\_env**(params.env\_name)

# *aligning the seed of the environment on the seed of the agent*

**env.seed**(params.seed + rank)

# *creating the model from the ActorCritic class*

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

# *state is a numpy array of size 1\*42\*42, in black & white*

    state = **env.reset**()

# *converting the numpy array into a torch tensor*

    state = **torch.from\_numpy**(state)

    done = **True** # *when the game is done*

# *initializing the length of an episode to 0*

episode\_length = 0

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

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

# *synchronizing with the shared model*

# *the agent gets the shared model to do an exploration on num\_steps*

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

# *if it is the first iteration of the while loop or if the game was just done, then:*

**if** done:

# *the cell states of the LSTM are reinitialized to zero*

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

# *the hidden states of the LSTM are reinitialized to zero*

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

**else**:

# *we keep the old cell states, making sure they are in a torch variable*

cx = **Variable**(cx.data)

# *we keep the old hidden states, making sure they are in a torch variable*

hx = **Variable**(hx.data)

# *initializing the list of values (V(S))*

values = []

# *initializing the list of log probabilities*

log\_probs = []

# *initializing the list of rewards*

rewards = []

# *initializing the list of entropies*

entropies = []

# *going through the num\_steps exploration steps*

**for** step **in** **range**(params.num\_steps):

# *getting from the model the output V(S) of the critic,*

# *the output Q(S,A) of the actor, and the new hidden & cell states*

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

# *generating a distribution of probabilities of the Q-values according to the softmax:*

# *prob(a) = exp(prob(a))/sum\_b(exp(prob(b)))*

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

# *generating a distribution of log probabilities of the Q-values according to the log softmax:*

# *log\_prob(a) = log(prob(a))*

log\_prob = **F.log\_softmax**(action\_values)

entropy = -(log\_prob \* prob).**sum**(1) # *H(p) = - sum\_x p(x).log(p(x))*

**entropies.append**(entropy) # *storing the computed entropy*

# *selecting an action by taking a random draw from the prob distribution*

action = **prob.multinomial**().data

# *getting the log prob associated to this selected action*

log\_prob = **log\_prob.gather**(1, **Variable**(action))

**values.append**(value) # *storing the value V(S) of the state*

**log\_probs.append**(log\_prob) # *storing the log prob of the action*

# *playing the selected action, reaching the new state, and getting the new reward*

state, reward, done, \_ = **env.step**(**action.numpy**())

# *if the episode lasts too long (the agent is stucked), then it is done*

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

# *clamping the reward between -1 and +1*

reward = **max**(**min**(reward, 1), -1)

**if** done: # *if the episode is done:*

episode\_length = 0 # *we restart the environment*

state = **env.reset**() # *we restart the environment*

state = **torch.from\_numpy**(state) # *tensorizing the new state*

**rewards.append**(reward) # *storing the new observed reward*

**if** done: # *if we are done*

# *we stop the exploration and we directly move on*

# *to the next step: the update of the shared model*

**break**

R = **torch.zeros**(1, 1) # *intializing the cumulative reward*

# *if we are not done: we initialize the cumulative reward with the value of the last shared state*

**if** **not** done:

value, \_, \_ = **model**((**Variable**(**state.unsqueeze**(0)), (hx, cx))) # *value of the last shared state*

R = value.data # *initialize the cumulative reward with the value of the last shared state*

# *storing the value V(S) of the last reached state S*

**values.append**(**Variable**(R))

policy\_loss = 0 # *initializing the policy loss*

value\_loss = 0 # *initializing the value loss*

R = **Variable**(R) # *making sure the cumulative reward R is a torch Variable*

# *Generalized Advantage Estimation: A(a,s) = Q(a,s) - V(s)*

gae = **torch.zeros**(1, 1) # *initializing the Generalized Advantage Estimation to 0*

# *starting from the last exploration step and going back in time*

**for** i **in** **reversed**(**range**(**len**(rewards))):

# *R = gamma\*R + r\_t*

# *= r\_0 + gamma \* r\_1 + gamma^2 \* r\_2 ... + gamma^(n-1)\*r\_(n-1) + gamma^nb\_step \* V(last\_state)*

R = params.gamma \* R + rewards[i]

# *R is an estimator of Q at time t = i so advantage\_i = Q\_i - V(state\_i) = R - value[i]*

advantage = R - values[i]

# *computing the value loss*

value\_loss = value\_loss + 0.5 \* **advantage.pow**(2)

# *computing the temporal difference*

TD = rewards[i] + params.gamma \* values[i + 1].data - values[i].data

# *gae = sum\_i (gamma\*tau)^i \* TD(i) with gae\_i*

# *= gae\_(i+1)\*gamma\*tau + (r\_i + gamma\*V(state\_i+1) - V(state\_i))*

gae = gae \* params.gamma \* params.tau + TD

# *computing the policy loss*

policy\_loss = policy\_loss - log\_probs[i] \* **Variable**(gae) - 0.01 \* entropies[i]

# *initializing the optimizer*

**optimizer.zero\_grad**()

# *we give 2x more importance to the policy\_loss than the value\_loss*

# *because the policy loss is smaller*

(policy\_loss + 0.5 \* value\_loss).**backward**()

# *clamping the values of gradient between 0 and 40*

# *to prevent the gradient from taking huge values and degenerating the algorithm*

**torch.nn.utils.clip\_grad\_norm**(**model.parameters**(), 40)

# *making sure the model of the agent and the shared model share the same gradient*

**ensure\_shared\_grads**(model, shared\_model)

# *running the optimization step*

**optimizer.step**()

No comment

# *Training the AI*

**import** torch

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

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

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

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

**def** **ensure\_shared\_grads**(model, shared\_model):

**for** param, shared\_param **in** **zip**(**model.parameters**(), **shared\_model.parameters**()):

**if** shared\_param.grad **is** **not** **None**:

**return**

        shared\_param.\_grad = param.grad

**def** **train**(rank, params, shared\_model, optimizer):

**torch.manual\_seed**(params.seed + rank)

    env = **create\_atari\_env**(params.env\_name)

**env.seed**(params.seed + rank)

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

    state = **env.reset**()

    state = **torch.from\_numpy**(state)

    done = **True**

    episode\_length = 0

**while** **True**:

        episode\_length += 1

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

**if** done:

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

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

**else**:

            cx = **Variable**(cx.data)

            hx = **Variable**(hx.data)

        values = []

        log\_probs = []

        rewards = []

        entropies = []

**for** step **in** **range**(params.num\_steps):

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

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

            log\_prob = **F.log\_softmax**(action\_values)

            entropy = -(log\_prob \* prob).**sum**(1)

**entropies.append**(entropy)

            action = **prob.multinomial**().data

            log\_prob = **log\_prob.gather**(1, **Variable**(action))

**values.append**(value)

**log\_probs.append**(log\_prob)

            state, reward, done, \_ = **env.step**(**action.numpy**())

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

            reward = **max**(**min**(reward, 1), -1)

**if** done:

                episode\_length = 0

                state = **env.reset**()

            state = **torch.from\_numpy**(state)

**rewards.append**(reward)

**if** done:

**break**

        R = **torch.zeros**(1, 1)

**if** **not** done:

            value, \_, \_ = **model**((**Variable**(**state.unsqueeze**(0)), (hx, cx)))

            R = value.data

**values.append**(**Variable**(R))

        policy\_loss = 0

        value\_loss = 0

        R = **Variable**(R)

        gae = **torch.zeros**(1, 1)

**for** i **in** **reversed**(**range**(**len**(rewards))):

            R = params.gamma \* R + rewards[i]

            advantage = R - values[i]

            value\_loss = value\_loss + 0.5 \* **advantage.pow**(2)

            TD = rewards[i] + params.gamma \* values[i + 1].data - values[i].data

            gae = gae \* params.gamma \* params.tau + TD

            policy\_loss = policy\_loss - log\_probs[i] \* **Variable**(gae) - 0.01 \* entropies[i]

**optimizer.zero\_grad**()

        (policy\_loss + 0.5 \* value\_loss).**backward**()

**torch.nn.utils.clip\_grad\_norm**(**model.parameters**(), 40)

**ensure\_shared\_grads**(model, shared\_model)

**optimizer.step**()