Chapter – 3 (implementation – Part 4 - training mechanisms)

**Artificial Intelligence**

**AI-app (CNN-DQ): DOOM Game-play** (part 4)

Doom env, actions

Building an AI: cnn, body, AI (obj)

Experience Replay

Eligibity Trace: fn

Moving Average: MA-class

**3.11 step 11: Doom env, actions**

We have the AI with the brain (*architecture* of the *neural network*, outputs ***Q-values***) and the body (the way the actions are played).

* So it's time to train the AI with ***Deep Convolutional Q-learning***.
* That is we will now implement:
* ***Experience Replay***.
* We'll work with the ***Q-values*** and ***Rewards***
* ***Eligibility Trace:*** It will improve the training process a lot.
* Eligibility Trace is a powerful technique which consists of *accumulating the* ***reward*** over *several steps* and the ***Q-values*** are *learned* on this *accumulation of rewards*.
* In our previous *Self-driving-car project*, the ***Q-values*** were *learned after* ***each transition***, therefore after getting ***each reward***.
* However this time, we will be learning the ***Q-values*** after getting ***several rewards*** instead of just *one reward*.
* Instead of having *one transition after the other* and *updating the Q value each time*, now the ***Q-values*** are going to be ***updated*** every n-steps (that's why eligibility traces rather called n-steps eligibility trace) and **n** is the number after which the ***Q-values*** are going to be ***updated***.
* In our model we are going to have **n=10**. So in our case there will be ***10 steps eligibility trace***, therefore we will update and ***learn*** the Q-values every **10 steps** after ***accumulating*** the ***rewards*** on these ***10 steps***.
* That ***eligibility trace*** will make our model even more powerful and in the end we will get outstanding results.
* Doom Environment: We'll use the following code to get the Doom Environment :

*# -------- Getting the Doom environment --------*

doom\_env = **image\_preprocessing.PreprocessImage**(

**SkipWrapper**(4)(**ToDiscrete**("minimal")(**gym.make**("ppaquette/DoomCorridor-v0"))),

width = 80, height = 80, grayscale = **True**)

doom\_env = **gym.wrappers.Monitor**(doom\_env, "videos", force = **True**)

number\_actions = doom\_env.action\_space.n

* We are just using the **image\_preprocessing.py** ***external file*** from our working directory folder.
* So basically the order is take this line of code:

**gym.make**("ppaquette/DoomCorridor-v0")

* DoomCorridor-v0 is the name of the environment of the game we're playing.
* So first we import the environment with this **gym.make**("ppaquette/DoomCorridor-v0"). The details you can find on the *OpenAI's page and tutorials*.
* But then we use this **PreprocessImage** class, which is a class from **image\_preprocessing** to pre-process the images that will come into the Neural Network.
* We preprocess them so that they have a square format with the dominations because in our Neural Network we've set our input images to have the dimensions (1, 80, 80). Remember **1** is the ***number of channels*** i.e. we're working with black and white images. And **80** by **80** means that the dimensions of our input images will be **pixels**. Since we set that in our Neural-Network, now we need to specify this when we inputting the images from the Doom-Environment in **PreprocessImage** class.
* So that's why we used ***grayscale***, ***width*** & ***height*** like this: width = 80, height = 80, grayscale = **True**
* After we import the environment with the ***right format*** of the ***input images***, next we import the whole game with the videos with following line of code:

doom\_env = **gym.wrappers.Monitor**(doom\_env, "videos", force = **True**)

* So that in the end we'll see the videos of our AI playing Doom, how it will kill the monsters and try to reach the vest. These videos will be stored inside the ***videos folder***.
* Number of actions: Remember that our Neural Network (CNN class) takes number\_actions as input, it is *the number of actions* from the *Doom-Environment*.
* This enables us to test our AI easily on several environments i.e. several Doom Environments. Since the different environments have different number of actions, we specified this number\_actions variable as the input of the CNN class (the brain).
* To get this number\_actions we'll use the Doom-environment **doom\_env** that we just important.

number\_actions = doom\_env.action\_space.n

* Here **action\_space** is the ***set of actions*** (have a look at the OpenAI tutorials to see how OpenAI-gym environments works). From this set of actions we can access the ***number of actions*** in the environment, which is **n**.
* So it'll return 7 actions. (in the OpenAI's page it is specified 6-actions, but beside move forward, move left, move right, turn left, turn right and shoot, you can run, that makes seven actions).
* Now we get our ***Doom-environment***, the ***number of actions***; so we have **so-far everything** that we need for our AI-brain.
* Next we'll create a brain object (i.e. an object of CNN class) by inputting the number\_actions. Then of course we'll create the body and eventually the AI.

Now we can build as many AIs as we want.

**3.12 step 12: Building an AI: cnn, body, AI (obj)**

We'll create three objects of the three classes: cnn, softmax\_body, AI. Using these three objects we'll create our AI with a brain & a body.

# -------- Building an AI --------

cnn = **CNN**(number\_actions)

softmax\_body = **SoftmaxBody**(T = 1.0)

ai = **AI**(brain = cnn, body = softmax\_body)

* First we need to create a brain and a body because, when we create an AI we need to input a ***brain*** and a ***body***.
* Brain (cnn): cnn is the brain; it contains a ***Convolutional-Neural-Network*** and it's the object of **CNN class**.

cnn = **CNN**(number\_actions)

* We input the number\_actions as input to **CNN** class (it is the parameter of **CNN**'s **\_\_init\_\_()** function). From our ***Doom-Environment*** we already have this number\_actions.
* Body: To make the body, we're going to create an object of the **SoftmaxBody** class and we're going to call this object softmax\_body.

softmax\_body = **SoftmaxBody**(T = 1.0)

* The **SoftmaxBody** class requires ***temperature*** as an argument, and we set T = 1.0.
* We start with 1, it's a small temperature but this might work very well. We can increase it, then the action with the highest Q-value will have a higher probability to be selected.
* But the exploration will be decreased because the other actions which will have ***lower probabilities*** to be ***selected*** and therefore they will be ***less explored***.
* Make the AI: Now we have a brain and a body, it's time to make the final AI. We simply need to create a new object "**ai**" from our AI class. And since the AI is composed of a ***brain*** and a ***body*** we input the object of the brain (convolutional neural network) and the object of the body (**SoftmaxBody**).

ai = **AI**(brain = cnn, body = softmax\_body)

So now we have an AI ready to be trained.

* Next we'll set up the ***Experience Replay***. We'll launch the whole ***Deep Convolutional Q-Learning*** process with Experience Replay and Eligibility Trace on ***10 steps*** and eventually once we have all this, we will train the AI to make it smart.
* We're not going to implement ***Experience Replay*** all over again like the self-driving car. We have it already implemented in ***experience\_replay.py*** file. We'll create an object of the **ReplayMemory** ***class*** of this experience\_replay.py file.

**3.13 step 13: Experience Replay**

Now our AI it is ready to be trained. And the first step of the training is to set our ***experience replay***.

# Setting up Experience Replay

n\_steps = **experience\_replay.NStepProgress**(env = doom\_env, ai = ai, n\_step = 10)

memory = **experience\_replay.ReplayMemory**(n\_steps = n\_steps, capacity = 10000)

* We'll now use the ***implemented*** version of ***experience replay*** which is ***adapted*** to ***eligibility trace***.
* ***Why we're using a separate file for experience replay?:*** Eligibility trace is a technique that: instead of learning the Q-values every transition we learn it ***every 10 transitions***.
* Instead of having a ***single target*** and a ***single reward*** for ***each step*** we're going to have accumulative target of ten steps and accumulative reward of 10 steps and we will ***learn on the 10 steps*** each time.
* So we are learning on ***10 transitions***, ***10 steps*** instead of 1 (which we did for self-driving-car). As a benefit, the ***training*** will take ***much less time***.
* But we have to **specify** this in ***Experience Replay*** that: we are learning every 10 steps.

That's the reason why this experience replay (and the reason we're using a separate .py file) is not a classic implementation of experience replay, like the one for the self-driving car.

* It is a ***modified experience replay implementation*** taking into account this 10 steps learning (eligibility trace).
* There are two classes in this experience\_replay.py file, one class (first class NStepProgress) that makes your ***AI progress doing 10 steps*** so that it can ***sum the rewards*** observed on these 10 steps.
* We need this class because we need to include these 10 steps in the ReplayMemory **class** which is the class we ***implement*** for ***experience replay*** and that's how we make sure that the ***memory*** also takes into account the fact that we're ***learning on 10 steps***.

That's why there are two classes in this implementation of experience replay to take into account that ***we're learning in 10 steps*** and that must be ***taken into account*** also in the ***memory***.

* Memory: memory is going to be an object of the **ReplayMemory** class which is a class in experience\_replay.py.

memory = **experience\_replay.ReplayMemory**(n\_steps = n\_steps, capacity = 10000)

* We have to input two arguments, the first argument is **n\_steps** which corresponds exactly to the *number of steps on which we're going to learn the Q-values*. It is the number of steps on which we ***accumulate*** the ***target*** and the ***reward***. We're going to have a **cumulative target** and the **cumulative reward**.
* The second argument is the capacity i.e. the ***size of the memory***. We set it 10,000 to make a ***memory of size 10000*** and therefore that means that we will get a ***memory of the 10000 last steps*** performed by the AI.
* But again we're ***not*** going to learn ***every transition***. We're going to learn every ***10 steps*** *among these* ***last 10000 steps*** of the memory.
* That's how we apply the ***eligibility-trace*** with every ***10 steps*** (trick of learning every 10 steps) along with ***replay-memory*** trick.
* We're going to *learn every 10 steps* and we're going to do it in the *memory composed of the last 10000 steps*. And ***experience-replay*** combined to ***eligibility-trace*** with 10 steps will considerably improve the training performance.
* The memory will contain the ***last 10000 steps*** performed by the ***AI*** and that will allow us to generate some ***mini-batches*** with a ***sample function***. That is we'll sample these ***mini batches*** of ***10 transitions*** in the memory composed of the ***10000 last steps***. (In the self-driving car we sampled only one transition).
* n\_steps: ***n\_steps*** is an object of the class **NStepProgress** in experience\_replay.py file. It allows the AI to make progress during 10 steps.

n\_steps = **experience\_replay.NStepProgress**(env = doom\_env, ai = ai, n\_step = 10)

* And remember, during the ***10 steps*** we will sum the ***rewards*** on the ***10 steps*** to get the cumulative rewards over ***10 steps***. And that is exactly ***eligibility trace***.
* We have to put three arguments:
* ***env:*** It is the environment here that we imported. doom\_env is our doom environment.
* ***ai:*** Is our AI and this will be of course the AI that we just built in previous section, i.e. ai the object of AI. (Notice **ai** on the ***left*** is also the named-parameter of **NStepProgres** class, **ai** on the ***right*** is our AI-class object)
* ***n\_step:*** That's where we will specify that we want to learn every 10 steps (i.e. every 10 transitions).

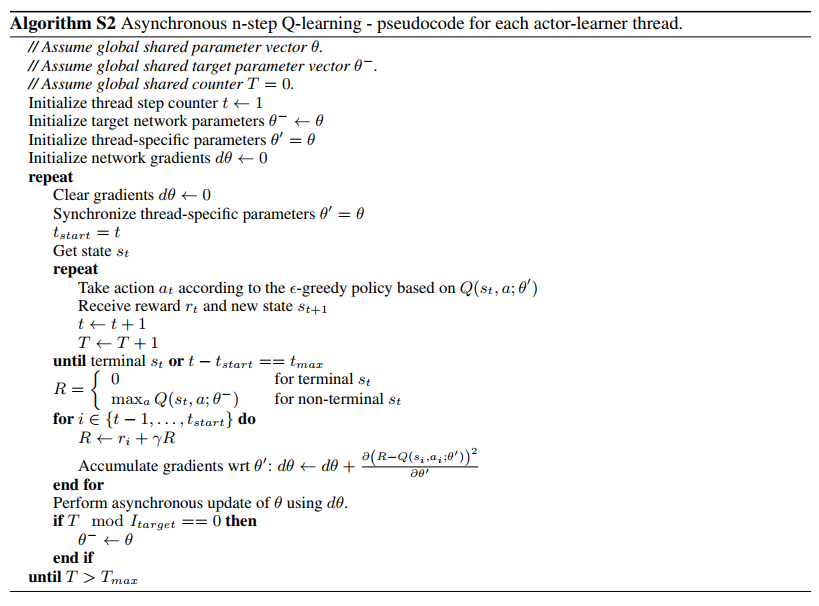
So right now we are just taking into account in the memory that there is a *learning on 10 steps* and this *learning on 10 steps* is called eligibility trace.

In next step we'll implement the eligibility trace.

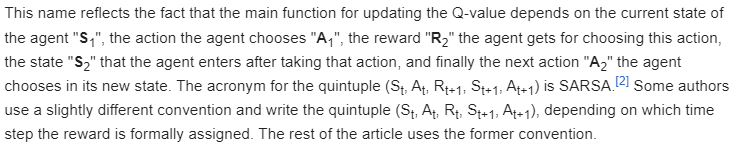
**3.14 step 14: eligibility\_trace() : part 1**

Here we're about to implement eligibility trace (SARSA) is actually an algorithm of the ***Asynchronous Actor Critics Agents Algorithms*** but we cannot consider it ***A3C*** because we're still going to have ***one agent***.

* However we're about to implement Asynchronous n-step Q-learning algorithm from the paper ***"Asynchronous Methods for Deep Reinforcement Learning"*** of Volodymyr Mnih Et. Al. Also from this paper we'll implement A3C algorithm in the Next chapter.
* ***Asynchronous n-step Q-learning*** almost the ***A3C*** but with one agent. The powerful thing about this is the ***n-step Q-learning*** & we're going to learn ***accumulative rewards*** and learn ***accumulative target*** on ***n steps*** (instead of one step like previous self-driving car).
* It will make the training much more performant and therefore our AI much more powerful.
* Following is the algorithm that we're about to implement. But with only ***one agent***.



* This one is called the ***Asynchronous n-step Q-learning***. But we don't have the right to say ***Asynchronous*** because we only have ***one agent***. Therefore we can call it ***n-step Q-learning eligibility trace*** or even ***SARSA***.
* **The difference is that:** here they take an action according to the -Greely (epsilon-greedy) policy based on the Q-values for the current state () and the action played .
* But in our case we didn't implement an -Greely policy, we implemented a **Soft-Max**. But the rest is the same. We'll basically follow the above pseudo-code. For example:
* We are going to compute the ***accumulated reward*** on n-steps (10 steps).
* We will get the ***maximum*** of the ***Q-values*** for the current state and the current action. Here is just a target parameter.
* ***Asynchronous one-step SARSA:*** The architecture of ***asynchronous one-step SARSA*** is almost similar to the architecture of ***asynchronous one-step Q-learning***, except the way target state-action value of the current state is calculated by the ***target*** ***network***. Instead of using the maximum Q-value of the next state s' by the target network, SARSA uses epsilon-greedy to choose the action a' for the next state s' and the Q-value of the next state action pair.
* SARSA: State–action–reward–state–action (SARSA) is an algorithm for ***learning*** a ***Markov decision process policy***, used in the ***Reinforcement Learning*** area of machine learning. It was proposed by ***Rummery*** and ***Niranjan*** in a technical note[1] with the name ***"Modified Connectionist Q-Learning" (MCQ-L)***. The alternative name ***SARSA***, proposed by ***Rich Sutton***, was only mentioned as a footnote (Wikipedia).



* Implementation: To implement ***eligibility trace*** we don’t need to implement a ***class***. We'll implement it as a ***function***. Because we don't really need to create *objects* of *eligibility trace* to make a model; a function will be enough.
* Basically what we want to do with this function is: to return the inputs and the target so that later when *training the AI* we are ready to ***minimize*** the ***squared distance*** *between the* ***predictions*** *and the* ***target*** *(i.e. predictions minus targets)*.
* To get the predictions we need the inputs because we are going to apply AI-brain (CNN) on the input to get the output signals as our predictions.
* Once we have our ***predictions*** and ***targets*** we will be ready to ***train the AI*** by trying to ***minimize*** this ***squared distance*** between the ***predictions*** and the ***target***.

**def** **eligibility\_trace**(batch):

    gamma = 0.99

    inputs = []

    targets = []

* We're going to name this function **eligibility\_trace** (we can also call it sarsa or n\_step\_Qleraning). This function is going to take one argument: a batch. Because we're going to get some inputs and some targets because we're going to ***train*** the ***AI*** on the ***batches***.
* Hence the ***inputs*** and the ***targets*** will go ***inside*** some ***batches*** and therefore we use **batch** as the input argument. This argument will contain *several* ***inputs*** and then *several* ***targets*** that we'll compute.
* Decay Parameter: We're going to use the Gamma () as the decay parameter. We'll start by introducing a variable for this Gamma parameter:

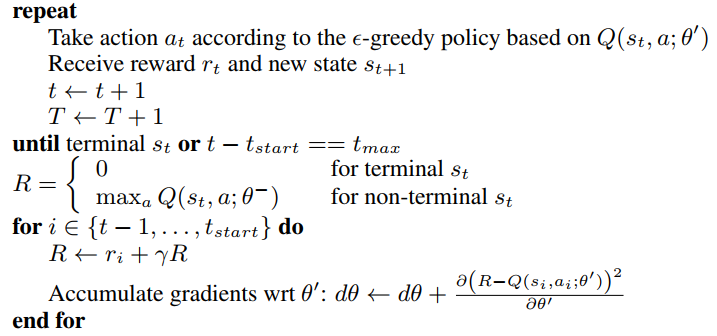
gamma = 0.99

* We choose 0.99, that's a classic good value for the Gamma and also is a good value for our AI.
* Prepare inputs and targets: We'll prepare our input and our targets that we want to return to train our AI.

inputs = []

targets = []

* We initialize them with ***empty lists*** because these inputs inside the batch, we're going to have several inputs & targets all into a list.
* We fill up those two lists because our function **eligibility\_trace**() will return several inputs and the associated several targets.
* According to the pseudo-code, we'll now implement a for-loop (this is our outer for-loop, but in pseudo-code inner repeat). [In the pseudo-code repeat means for-loop].



* We are going to compute the cumulative reward R, which is accumulated over the 10 steps.



* How is it computed: Well in ***each step*** that is not the ***last step***, we're going to get the ***maximum of the Q-values*** of the ***current-state*** we're enduring this ***n-step run***.
* And if we reach to the ***last state of the 10 steps***, this R will be equal to zero. That is we don't want to update it anymore.
* Then we have this second for-loop in which we'll update R by multiplying it by the decay parameter gamma and adding the reward.



* For-loop (outer): Let's start our outer for-loop, the iterative variable is going to be our 10 step series from the batch (sequence of 10 transitions).

**for** series **in** batch:

**input** = **Variable**(**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32)))

        output = **cnn**(**input**)

        cumul\_reward = 0.0 **if** series[-1].done **else** output[1].**data.max**()

**for** step **in** **reversed**(series[:-1]):

            cumul\_reward = step.reward + gamma \* cumul\_reward

        state = series[0].state

        target = output[0].data

        target[series[0].action] = cumul\_reward

**inputs.append**(state)

**targets.append**(target)

* series represents the series of 10 transitions in our input batch (i.e. the batches on which will train the AI).
* Accumulative reward: From the pseudo-code we see that, to get *accumulative reward* we need:
* ***State*** of the ***first transition*** of the series and
* ***State*** of the ***last transition*** of the series.
* input: So the first thing is to get these input states and we are going to put these two states into a variable that we call input .
* We'll put these two input states (first one and the last one of the series) into a Numpy-array later we'll convert that into a Torch-variable.

**input** = **Variable**(**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32)))

* ***First input:*** which is the *input-state* of the *first transition* of the *series* (to take the first transition, we use index 0 and then we access it by taking its state attribute).
* Because in our ***experience\_replay.py*** file we defined a special structure for each of the transition like below:

state = state, action = action, reward = r, done = is\_done

Each transition is composed of a **state**, an **action**, a **reward** and **is\_done**. So this special structure comes from the way we ***defined*** the transition and experience replay.

* Numpy array of first & last transition.

**np.array**([series[0].state, series[-1].state])

* From series[0].state we get the input state of the ***first transition***.
* ***Last input:*** And from series[-1].state we get input state of the ***last transition*** of the series. '-1' for the last index of the series.
* We put series[0].state, series[-1].state inside [] to apply **np.array()** function.
* Since we are going to convert that into a torch-tensor in a torch-variable, and a torch-tensor is a special array containing one single type.
* So we choose dtype = np.float32 to force to having one single type.

**np.array**([series[0].state, series[-1].state], dtype = np.float32)

* Now we can convert it into a torch-tensor in a torch-variable.
* Convert into a torch tensor using **torch.from\_numpy()**:

**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32))

* Finally we convert this **torch-tensor** into a **torch-variable** using **Variable** class.

**input = Variable**(**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32)))

So input will be an object of the Variable class. Now we have our two inputs that we need i.e. the *input state of the first transition* and the *input state of the last transition*.

* Output: Now get the output (i.e. the prediction) from our AI-brain using that input.

output = **cnn**(**input**)

* To get the outputs we use our **cnn** class (i.e. the brain) to return the prediction (i.e. outputs) we use the **input**.
* Cumulative reward: In this step we start to ***compute*** the ***cumulative reward***. Here we're going to do exactly the same as our S2 algorithm or the SARSA or n-step-Q-learning.
* We're going to introduce the cumulative reward variable which will be **cumul\_reward**.

cumul\_reward = 0.0 **if** series[-1].done **else** output[1].**data.max**()

* In the pseudo-code we can see that, to get this cumulative reward R,



* at each ***step*** of the ***10 steps run*** we need to update it by adding a 0 to R if we reached the *last state of the series* or
* the ***maximum*** of the Q-values if we haven't reached the last state of the series (i.e. all the steps except the last step).
* So we set **cumul\_reward = 0.0**, if we *reached the last state* (i.e. **series[-1]**) i. e. the last transition of the series.

|  |  |
| --- | --- |
| * We'll use **series[-1].done** because done actually is an attribute of our pre-defined ***transition structure*** that we defined in experience\_replay in ***experience\_replay.py*** file. * Actually this done comes from the OpenAI structure. In the web-page of **DoomCorridor-v0**, if we go to *documentation* we can see that *our observations i.e. our transitions* are defined by: an observation, reward, done and info. * Here **done** means that a *transition or a step is over*. * So we're going to use this **done** for our IF-condition. |  |

* **series[-1].done** means if the *last transition of the series* is ***completed***.
* And if we *haven't reached the last transition*, the cumulative reward is going to be updated with the *maximum of the Q-values* i.e. **output[1].data.max()**.
* And since this output is the *output of the brain* i.e. the ***predictions*** of the ***neural network*** and those are the ***predicted Q-values***. We used ***index 1***, because this *structure contains the Q-values* at ***index 1***.
* We used **data** to access the *data of this output structure* because it has the *special structure of a torch-Variable*, so with **data** we get our ***Q-values*** and then we use **max()** to take the maximum of those Q-values.

That’s how we get the *maximum of the Q-values* for the *non-terminal state* st.

**3.15 step 15: eligibility\_trace() : part 2**

* Now we're going to implement a ***second For-loop*** i.e. *for the 10 steps of the series we are going to* ***update*** *the* ***cumulative reward*** by multiplying it by (decay parameter, gamma = 0.99) then we add the reward .



* In the pseudo-code they start from the last step to the first step . That's exactly what we're going to do because in the end we want to get the ***cumulative reward*** being equal to

where are the ***rewards*** obtained in ***each of the N-steps*** of the series.

**for** series **in** batch:

**input** = **Variable**(**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32)))

        output = **cnn**(**input**)

        cumul\_reward = 0.0 **if** series[-1].done **else** output[1].**data.max**()

**for** step **in** **reversed**(series[:-1]):

            cumul\_reward = step.reward + gamma \* cumul\_reward

        state = series[0].state

        target = output[0].data

        target[series[0].action] = cumul\_reward

**inputs.append**(state)

**targets.append**(target)

* So this loop starts from the right and goes to the left. The iterative variable is the step, because we'll go from the last step to the first step of a ***series of transitions***. To do that we're going to use reversed().
* In the pseudo-code we go from to . i.e. we don't go from the ***actual last step*** (terminal step) but the step before that i.e. . However, is the ***actual first step***.
* That's why we used series[:-1], which means start from the *first element* up to the *element before the last element*. And then we *reverse* it to go from the *right* to the *left*.
* Inside this loop we're going to ***update*** the ***cumulative reward*** by multiplying it by the decay parameter gamma and adding the ***reward*** *obtained in the current step* (that is in the step of the FOR-loop).

cumul\_reward = step.reward + gamma \* cumul\_reward

* Since step is a ***transition***, we used step.reward to access the reward.
* step.reward: is the reward in the step we are in right now the loop.
* gamma \* cumul\_reward: decayRate \* previous cumulative reward before it's updated.
* Next, we go back to the first FOR-loop because this inner-for-loop is just to compute the cumulative reward.
* Remember, the goal of doing all of this is to get our inputs ready and our targets ready so that we can *minimize the squared difference* between the two for the training. So the next thing to do is that to get these inputs and targets ready.
* Preparing the input state: First we add the first state of the series in our input's list. So far this ***input state*** is in this **input** variable, but that was just to compute the **output** (what we did at the beginning of the outer FOR-loop).
* So we're going to prepare this ***input state*** *of the* ***first step*** separately, after preparation we'll append this to our **input** list.

        state = series[0].state

        target = output[0].data

        target[series[0].action] = cumul\_reward

* **state**: We use same approach as we did before to get the *input-state* of the *first transition* of the *series* to get the **input**. We'll take the first index of the series which contains the first transition and then we get the state of this first transition.

state = series[0].state

* **target**: Similarly we're going to get separately the targets associated to the *input state of the first transition*, which will be the *Q-value of the first step*.

target = output[0].data

And since the ***Q-value*** is *returned by the CNN* and its contained in the output "output = **cnn**(**input**)" and since this output is associated to the input (which contains the first element of the transition), so we can get this Q-value just taking the ***index zero***. And then we add data to get the Q-value of the input state of the first transition and that is exactly the target Q-value.

* Then we are going to ***update*** this target but only for the ***action*** that was selected in the first step of the series.

target[series[0].action] = cumul\_reward

* + To access the *action* corresponding to the *first step of the series*, we use "series[0].action". This is because we're using an ***attribute*** ***structure*** that we defined for the series and ***action*** is an ***attribute*** of the ***first step (first transition) of the series***. Because *each* *transition* of the *series* has the following structure: **state**, **action**, **reward** and **done**.
  + "series[0].action" means that we're simply getting the *action of this first state*.
  + Now the target for that specific action of the *first step* needs to be updated by the cumulative reward.
  + The target associated to the action that was played in the *first step of the series* is the cumulative reward *that we just computed*.
* Updating inputs and targets lists: Now we update the two lists: **inputs = []**, **targets = []** that we defined at the beginning of our **eligibility\_trace()** function. We update those lists by appending the ***input*** and the ***target*** that we're just computed for the *first step (first transition of the series)*.
* It's important to understand that, we only need to ***update*** *the* ***first step*** *of the series* because we train the AI on 10 steps and *we don't* ***need*** *to get any* ***inputs*** *or any* ***targets*** *in the* ***following steps*** of the 10 steps because basically the *learning happens 10 steps afterwards*.
* That's why right now we only getting the state and the target of the *first step of the series*.
* To do that we have to input them (append) in our list of inputs and our list of targets.

**inputs.append**(state)

**targets.append**(target)

* Now we need to return the **inputs** and the **targets** that are now updated. First we get outside the FOR-loop,
* inputs:
* np.array(inputs, dtype = np.float32): We convert the inputs to numpy array with data-type float32.
* torch.from\_numpy(np.array(inputs, dtype = np.float32)): we convert np.**array**(inputs, dtype = np.float32)into a torch tensor using **torch.from\_numpy**().
* targets:
* torch.stack(targets): We're going to stack the targets together and to do this we use **torch.stack**()

**return** **torch.from\_numpy**(**np.array**(inputs, dtype = np.float32)), **torch.stack**(targets)

So this line of code basically returns the inputs and the targets that were just updated through the ***Eligibility Trace SARSA algorithm*** (or we can call it ***n-step-Q-learning***)

Next we do the final training which consists of minimizing the squared differences between the ***predictions of our inputs*** and the ***targets***.

# *Implementing Eligibility Trace*

**def** **eligibility\_trace**(batch):

    gamma = 0.99

    inputs = []

    targets = []

**for** series **in** batch:

**input** = **Variable**(**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32)))

        output = **cnn**(**input**)

        cumul\_reward = 0.0 **if** series[-1].done **else** output[1].**data.max**()

**for** step **in** **reversed**(series[:-1]):

            cumul\_reward = step.reward + gamma \* cumul\_reward

        state = series[0].state

        target = output[0].data

        target[series[0].action] = cumul\_reward

**inputs.append**(state)

**targets.append**(target)

**return** **torch.from\_numpy**(**np.array**(inputs, dtype = np.float32)), **torch.stack**(targets)

**3.16 step 16: Moving Average: MA-class**

Now we are ready to *train the network* to minimize the squared distance between the **outputs** and the **target** thanks to what we did with ***Eligibility Trace*** in the previous section.

* Basically we are ready to start the whole training by: getting inputs, targets, predictions then computing the loss error between the **predictions** and the **target** and then doing the backward propagation with ***Stochastic Gradient Descent*** to *update* the *weights*.
* But, since we want to ***compute*** the moving average on **100 steps** to *keep track of the* ***average*** *during the* ***training***. So we are going to make one more class that will get this moving average on 100 steps. We'll make it *a class with three functions* what we will do all this.
* **class MA:** This MA class is for moving average. We'll implement it with these three functions \_\_init\_\_(), add() and average().
* **\_\_init\_\_( self, size):** This function is going to take two arguments.
* **self:** Represents the *MA class object itself*.
* **size:** Corresponds to the *size of the list* of the ***rewords*** of which we're going to compute the ***average***. It's going to be 100.
* Now we have to *initialize the variables* specific to this MA class's object. And these are: self.list\_of\_rewards and self.size.

**class** MA:

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

        self.list\_of\_rewards = []

        self.size = size

* **self.list\_of\_rewards:** It's a list of rewards, that's going to be the list containing the ***100 rewards*** of which we're going to compute the ***average***. We simply initialize this by an empty list.
* **self.size:** Corresponds to the *size* of the *list of the rewords*.
* **add(self, rewards):** Next function is the **add()** function and it'll *add the cumulative rewards* (it's not the *simple reward*, because we are doing *Eligibility Trace* and we *learn every 10 steps*). So this **add()** function will add the *cumulative reward* to that list of rewards list\_of\_rewards. This function takes 2 arguments:

**def** **add**(self, rewards):

**if** **isinstance**(rewards, **list**):

            self.list\_of\_rewards += rewards

**else**:

            self**.list\_of\_rewards.append**(rewards)

**while** **len**(self.list\_of\_rewards) **>** self.size:

**del** self.list\_of\_rewards[0]

* **self:** Represents the *MA class object itself*. Also we need to access the self.list\_of\_rewards and self.size. Because we are going to ***append*** the ***cumulative reward*** to this list\_of\_rewards.
* **rewards:** It will represent the ***cumulative reward***.
* ***Adding the reward to list\_of\_rewards:***Whenever we get a cumulative reward (a new one) when we progress on 10 new steps, we'll add this to the list.
* Since we're working with *batches*, the reward will be in some *lists* but in some *other cases* the reward can also be as a *single element*. I.e. in some cases the **rewards** variable can be a *list* or a *single element*. So we'll use two conditions to add these rewards to list\_of\_rewards.
* If **rewards** is a list itself, we just simply *add (concatenate)* it to the list\_of\_rewards.
* If **rewards** is a single element, then we use append() to append it to list\_of\_rewards.
* First we check the if it is a list or not using **isinstance**(). If it is, then we concatenate to list\_of\_rewards. Otherwise, if it's a single element (a single cumulative reward), we'll append it to list\_of\_rewards

**if** **isinstance**(rewards, **list**):

            self.list\_of\_rewards += rewards

**else**:

            self**.list\_of\_rewards.append**(rewards)

* Making A Sliding Window: What does happen when this list\_of\_rewards gets more than 100 (self.size) elements?
* In that case what we have to do is: delete the first element of list\_of\_rewards to make sure that it contains no more than 100 elements. (same thing we did for the self driving car by making a score and reward\_window).

**while** **len**(self.list\_of\_rewards) **>** self.size:

**del** self.list\_of\_rewards[0]

* So we'll use a while-loop to check if the size of list\_of\_rewards is more than 100 (i.e. the size of list\_of\_rewards is more than self.size). In that case we'll ***delete the first element*** of our list\_of\_rewards.
* Average: We'll implement a function to compute the average of our list of rewards (the list will contains on the run 100 elements). With the help of the sliding window and the average function, we will compute the moving average of 100 steps each time.
* The average() function will take one argument.
* **self**: Because we still need to use **self.list\_of\_rewards** which is a variable of our object.

**def** **average**(self):

**return** **np.mean**(self.list\_of\_rewards)

* np.mean(self.list\_of\_rewards): We'll return the average directly and we'll also use the mean() function from numpy (i.e.np.mean) to compute the ***average of a list***.

Now we have our average on 100 steps.

* Now we get the instructions on how to obtain a moving average of 100 steps. And since we're going to use *one moving average object* when doing the training, let's create this moving average object.

ma = **MA**(100)

* Notice we set the size to 100 because we want to ***compute*** the ***moving average*** on one 100 steps.

# *Making the moving average on 100 steps*

**class** MA:

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

        self.list\_of\_rewards = []

        self.size = size

**def** **add**(self, rewards):

**if** **isinstance**(rewards, **list**):

            self.list\_of\_rewards += rewards

**else**:

            self**.list\_of\_rewards.append**(rewards)

**while** **len**(self.list\_of\_rewards) **>** self.size:

**del** self.list\_of\_rewards[0]

**def** **average**(self):

**return** **np.mean**(self.list\_of\_rewards)

ma = **MA**(100)

Now we are ready to train our AI to finally be intelligent. From this point our AI-agent will become smart. After done training, our agent will be fully ready and therefore we will execute the code.

And then the AI will play Doom and eventually we'll watch the videos of our AI playing Doom. Also we will see if it manages to reach the vest.

**PART 2 Codes : Implementing Eligibility Trace**

# *Part 2 - Training the AI with Deep Convolutional Q-Learning*

# *Getting the Doom environment*

doom\_env = **image\_preprocessing.PreprocessImage**(**SkipWrapper**(4)(**ToDiscrete**("minimal")(**gym.make**("ppaquette/DoomCorridor-v0"))), width = 80, height = 80, grayscale = **True**)

doom\_env = **gym.wrappers.Monitor**(doom\_env, "videos", force = **True**)

number\_actions = doom\_env.action\_space.n

# *Building an AI*

cnn = **CNN**(number\_actions)

softmax\_body = **SoftmaxBody**(T = 1.0)

ai = **AI**(brain = cnn, body = softmax\_body)

# *Setting up Experience Replay*

n\_steps = **experience\_replay.NStepProgress**(env = doom\_env, ai = ai, n\_step = 10)

memory = **experience\_replay.ReplayMemory**(n\_steps = n\_steps, capacity = 10000)

# *Implementing Eligibility Trace*

**def** **eligibility\_trace**(batch):

    gamma = 0.99

    inputs = []

    targets = []

**for** series **in** batch:

**input** = **Variable**(**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32)))

        output = **cnn**(**input**)

        cumul\_reward = 0.0 **if** series[-1].done **else** output[1].**data.max**()

**for** step **in** **reversed**(series[:-1]):

            cumul\_reward = step.reward + gamma \* cumul\_reward

        state = series[0].state

        target = output[0].data

        target[series[0].action] = cumul\_reward

**inputs.append**(state)

**targets.append**(target)

**return** **torch.from\_numpy**(**np.array**(inputs, dtype = np.float32)), **torch.stack**(targets)

# *Making the moving average on 100 steps*

**class** MA:

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

        self.list\_of\_rewards = []

        self.size = size

**def** **add**(self, rewards):

**if** **isinstance**(rewards, **list**):

            self.list\_of\_rewards += rewards

**else**:

            self**.list\_of\_rewards.append**(rewards)

**while** **len**(self.list\_of\_rewards) **>** self.size:

**del** self.list\_of\_rewards[0]

**def** **average**(self):

**return** **np.mean**(self.list\_of\_rewards)

ma = **MA**(100)

**All code at once**

# *Importing the libraries*

**import** numpy **as** np

**import** torch

**import** torch.nn **as** nn

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

**import** torch.optim **as** optim

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

# *Importing the packages for OpenAI and Doom*

**import** gym

**from** gym.wrappers **import** SkipWrapper

**from** ppaquette\_gym\_doom.wrappers.action\_space **import** ToDiscrete

# *Importing the other Python files*

**import** experience\_replay, image\_preprocessing

# *Part 1 - Building the AI*

# *Making the brain*

**class** CNN(nn.Module):

**def** **\_\_init\_\_**(self, number\_actions):

**super**(CNN, self).**\_\_init\_\_**()

        self.convolution1 = **nn.Conv2d**(in\_channels = 1, out\_channels = 32, kernel\_size = 5)

        self.convolution2 = **nn.Conv2d**(in\_channels = 32, out\_channels = 32, kernel\_size = 3)

        self.convolution3 = **nn.Conv2d**(in\_channels = 32, out\_channels = 64, kernel\_size = 2)

        self.fc1 = **nn.Linear**(in\_features = self**.count\_neurons**((1, 80, 80)), out\_features = 40)

        self.fc2 = **nn.Linear**(in\_features = 40, out\_features = number\_actions)

**def** **count\_neurons**(self, image\_dim):

        x = **Variable**(**torch.rand**(1, \*image\_dim))

        x = **F.relu**(**F.max\_pool2d**(self**.convolution1**(x), 3, 2))

        x = **F.relu**(**F.max\_pool2d**(self**.convolution2**(x), 3, 2))

        x = **F.relu**(**F.max\_pool2d**(self**.convolution3**(x), 3, 2))

**return** **x.data.view**(1, -1).**size**(1)

**def** **forward**(self, x):

        x = **F.relu**(**F.max\_pool2d**(self**.convolution1**(x), 3, 2))

        x = **F.relu**(**F.max\_pool2d**(self**.convolution2**(x), 3, 2))

        x = **F.relu**(**F.max\_pool2d**(self**.convolution3**(x), 3, 2))

        x = **x.view**(**x.size**(0), -1)

        x = **F.relu**(self**.fc1**(x))

        x = self**.fc2**(x)

**return** x

# *Making the body*

**class** SoftmaxBody(nn.Module):

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

**super**(SoftmaxBody, self).**\_\_init\_\_**()

        self.T = T

**def** **forward**(self, outputs):

        probs = **F.softmax**(outputs \* self.T)

        actions = **probs.multinomial**()

**return** actions

# *Making the AI*

**class** AI:

**def** **\_\_init\_\_**(self, brain, body):

        self.brain = brain

        self.body = body

**def** **\_\_call\_\_**(self, inputs):

**input** = **Variable**(**torch.from\_numpy**(**np.array**(inputs, dtype = np.float32)))

        output = self**.brain**(**input**)

        actions = self**.body**(output)

**return** **actions.data.numpy**()

# *Part 2 - Training the AI with Deep Convolutional Q-Learning*

# *Getting the Doom environment*

doom\_env = **image\_preprocessing.PreprocessImage**(**SkipWrapper**(4)(**ToDiscrete**("minimal")(**gym.make**("ppaquette/DoomCorridor-v0"))), width = 80, height = 80, grayscale = **True**)

doom\_env = **gym.wrappers.Monitor**(doom\_env, "videos", force = **True**)

number\_actions = doom\_env.action\_space.n

# *Building an AI*

cnn = **CNN**(number\_actions)

softmax\_body = **SoftmaxBody**(T = 1.0)

ai = **AI**(brain = cnn, body = softmax\_body)

# *Setting up Experience Replay*

n\_steps = **experience\_replay.NStepProgress**(env = doom\_env, ai = ai, n\_step = 10)

memory = **experience\_replay.ReplayMemory**(n\_steps = n\_steps, capacity = 10000)

# *Implementing Eligibility Trace*

**def** **eligibility\_trace**(batch):

    gamma = 0.99

    inputs = []

    targets = []

**for** series **in** batch:

**input** = **Variable**(**torch.from\_numpy**(**np.array**([series[0].state, series[-1].state], dtype = np.float32)))

        output = **cnn**(**input**)

        cumul\_reward = 0.0 **if** series[-1].done **else** output[1].**data.max**()

**for** step **in** **reversed**(series[:-1]):

            cumul\_reward = step.reward + gamma \* cumul\_reward

        state = series[0].state

        target = output[0].data

        target[series[0].action] = cumul\_reward

**inputs.append**(state)

**targets.append**(target)

**return** **torch.from\_numpy**(**np.array**(inputs, dtype = np.float32)), **torch.stack**(targets)

# *Making the moving average on 100 steps*

**class** MA:

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

        self.list\_of\_rewards = []

        self.size = size

**def** **add**(self, rewards):

**if** **isinstance**(rewards, **list**):

            self.list\_of\_rewards += rewards

**else**:

            self**.list\_of\_rewards.append**(rewards)

**while** **len**(self.list\_of\_rewards) **>** self.size:

**del** self.list\_of\_rewards[0]

**def** **average**(self):

**return** **np.mean**(self.list\_of\_rewards)

ma = **MA**(100)