Chapter – 3 (implementation – Part 5 – Train & Test)

**Artificial Intelligence**

**AI-app (CNN-DQ): DOOM Game-play** (part 5)

Apply training mechanisms

Test the AI

**Train the AI**

**3.17 Step 17: Apply training mechanisms**

In this part we're going to make our AI smart by training. We will train it's intelligence to reach the goal (i.e. reaching the vest without getting killed).

* To do this we're going to basically train the neural network to *output the right predictions*.
* And everything is already ready because these ***output signals*** from the ***brain*** already have the *right transmission* to the ***body*** to *play* the *final actions*.
* Basically now what we're about to do is:
* take some ***random batches*** from the ***memory***,
* get our ***inputs*** from these ***samples***
* get the ***outputs***
* get the ***target***
* get the ***predictions***
* compute the ***loss-error*** between the predictions and the target
* perform ***backward propagation*** with ***stochastic gradient descent*** to ***update the weights*** according to how much they contributed to that loss-error.
* We already have all the tools to implement the training, not only *PyTorch tools* (*optimizer*, *loss-functions*) but also we have all the *classes that we made* before (*CNN brain to get the predictions, experience replay implementation, eligibility trace*) and all these tools combined to the PyTorch tools we'll make the training super performant.

Therefore eventually we will get a super powerful AI.

* Loss-function: First thing is to get the loss-function that we'll use during the training when computing the error and an optimizer.
* **loss** is the variable for the loss function, (we'll use "mean squared error") using **nn.MSELoss()** from torch.nn.

loss = nn.MSELoss()

* We use MSE because basically our predictions are ***Q-values*** of the different actions. And since these Q-values are ***real numbers*** and we're kind of doing some *Neural Network for* ***Regression***, therefore the loss function is the "mean squared error". That's the loss function we use in general for regression.
* Optimizer: We'll use Adam optimizer, as we did for self-driving car. We have to input *two* essential *arguments*.
* ***The first argument:*** **cnn.parameters**() will make the connection between the *optimizer* and the *parameters of our neural network* (i.e. the weights of the neurons of AI's brain, cnn class).

optimizer = **optim.Adam**(**cnn.parameters**(), lr = 0.001)

* **cnn.parameters**() makes the connection between the ***optimizer*** and the ***weights*** *of the neurons* in the brain of our AI.
* ***The second argument*** "lr = 0.001" is a ***learning rate***. We have to take a small learning rate because we *don't want to* ***converge*** *too fast* and we want to have some exploration. Recall we also used *similar learning rate* for the *self-driving car*.
* Number of epochs: All right so now we have a loss-function, an optimizer so we are almost ready to start the FOR-loop. But before that we need to decide the ***number of epochs*** that we will be training the AI.

nb\_epochs = 100

* We set 100 epochs. That will be way enough to train the AI. We hope the AI will manage to reach the vest way before 100 epochs (like 20 or 30). We can also increase it later.
* FOR-loop (outer): Now we have our ***number of epochs***, so we can start the FOR-loop, which we iterate over the epochs.

**for** epoch **in** **range**(1, nb\_epochs + 1):

* We used **nb\_epochs + 1** because we start at 1 and the end-point of **range**() is not inclusive. Hence to go up to 100 we need to add 1 with nb\_epochs.
* Now we're going to do *200 runs of 10 steps*. So each epoch will be 200 runs one after the other of 10 steps.

**memory.run\_steps**(200)

* To do this, we have to use **run\_steps()** function from our **experience\_replay** class (i.e. we'll use the **memory** object with n\_steps = 10 steps and a capacity of 10000) and we specify *200 successive runs of 10 steps*.
* Sample some batches: Now we have these 200 steps running at each epoch. So it's time to ***sample some batches*** from these runs. To sample these batches we'll use another function from **experience\_replay** class (i.e. the **memory** object) which is **sample\_batch**() and that will *generate some batches* from those 200 runs.
* But remember these batches are the batches of series of transitions (i.e. series of 10 *steps* / *transitions*) as opposed to before (self-driving-car) where the batches were just some batches of ***single transition***.

**memory.sample\_batch**(128)

* For the batch size, we can take **32** or **64** or even **128**.
* It's a common practice to use 32. That's what you will see in general in the *Neural Networks Architectures* when doing some *Batch Learning*.
* But our case is quite different; we're *just sampling* some batches of 10 steps. So it's better to take batches with larger sizes. That's why we choose 128.
* Inner FOR loop: We'll use a FOR-loop because we want to take several batches and we're taking them in what is returned by **memory.sample\_batch**(128). So this For-Loop:

**for** batch **in** **memory.sample\_batch**(128)

means that every *128 steps*, our memory will give us a *batch of size 128* which will contain the last 128 steps that we're just run.

* We're just getting some *batches of size 128* and the *learning* is going to *happen* on these *batches*.
* And inside these batches we will have eligibility trace running in order to learn every 10 steps.
* inputs & targets: Note that we're still in one epoch, but the inner-loop runs over different batches. Now we're going to get our inputs and our targets separately.
* To do this, we're gonna use eligibility trace. We implemented eligibility\_trace() as a function and it takes batches and outputs the targets and inputs. We'll apply eligibility\_trace() to the current batch of the *inner FOR-loop*.
* So we create two new variables to store the inputs and the targets from eligibility\_trace().

inputs, targets = **eligibility\_trace**(batch)

* Next, we convert those in to Torch-Variable:

inputs, targets = **Variable**(inputs), **Variable**(targets)

Now the ***inputs of the brain*** are converted into some Torch-Variable and the ***targets*** also are converted into some Torch-Variable.

* Predictions: We now feed the above inputs to our NN (cnn, brain of our AI agent) to get the predictions. Because we have to calculate the loss between the predictions and the targets.

predictions = **cnn**(inputs)

* Loss: We have the predictions and the targets, so we can get the loss. To get the loss error we'll apply our predictions and targets to the loss() function.

loss\_error = **loss**(predictions, targets)

* Back propagate the loss\_error: After getting the loss\_error, we'll *back propagate* it back into the *neural network* to *update* the *weights* and we do that with *Stochastic-Gradient-Descent (SGD)*.
* To perform SGD we need our optimizer (we've used Adam-optimizer).
* Initialize: But first we need to initialize it, we take our optimizer object and apply **zero\_grad**() method to initialize it.

**optimizer.zero\_grad**()

* ***Back propagate:*** Now we back propagate the loss error back into the *neural network* and to do this we'll use **backward**() method from **nn.MSELoss** (by using **loss\_error**  which is an object of **nn.MSELoss**).

**loss\_error.backward**()

That’s how we apply the backward propagation.

* ***Update the weights:*** Since the **loss\_error** is back propagated into the *neural network*, now we can update the weights with Stochastic-Gradient-Descent (SGD).

**optimizer.step**()

To do this, we take our optimizer and we apply the **step()** method.

* Average reward: We are going to print the average reward every epoch. So that we can keep track of how the AI's training is going.
* We want to see the average reward increasing over the epochs.
* ***Exploitation phase:*** The average reward might not increase at the beginning but then once it reaches the *exploitation phase*, then it'll definitely increase. It will increase up to a certain level: when it reaches the vest as fast as possible.
* We compute the average reward outside the batch-loop (inner FOR-loop) but inside the epoch-loop (outer FOR-loop) because now we have our batch is sampled and we have a training happening in the batch.
* But now that forward-propagation and the backward-propagation are done in the batch. So we are getting back into the epoch-loop.
* We can now compute the cumulative rewards using our **n\_steps** object (from experience\_replay) because our n\_steps object contains this rewards\_steps() function that allows us to get the cumulative rewards *happening in the steps* during ***n-steps run***.
* So we are going to use rewards\_steps() right now to update the *new rewards of the steps*

rewards\_steps = n\_steps.rewards\_steps()

* Then we will update the *moving average* (**ma**) object by adding above cumulative rewards (rewards\_steps) to the **ma** object.

**ma.add**(rewards\_steps)

this will add the rewards of the steps into the moving average.

* After that we'll compute the *average* using **average()** to get the ***average reward***.

avg\_reward = **ma.average**()

* We convert the **epoch** and **avg\_reward** to string during print because we use **%s** in formatted printing.

**print**("Epoch: %s, Average Reward: %s" % (**str**(epoch), **str**(avg\_reward)))

# *Training the AI*

loss = **nn.MSELoss**()

optimizer = **optim.Adam**(**cnn.parameters**(), lr = 0.001)

nb\_epochs = 100

**for** epoch **in** **range**(1, nb\_epochs + 1):

**memory.run\_steps**(200)

**for** batch **in** **memory.sample\_batch**(128):

        inputs, targets = **eligibility\_trace**(batch)

        inputs, targets = **Variable**(inputs), **Variable**(targets)

        predictions = **cnn**(inputs)

        loss\_error = **loss**(predictions, targets)

**optimizer.zero\_grad**()

**loss\_error.backward**()

**optimizer.step**()

    rewards\_steps = **n\_steps.rewards\_steps**()

**ma.add**(rewards\_steps)

    avg\_reward = **ma.average**()

**print**("Epoch: %s, Average Reward: %s" % (**str**(epoch), **str**(avg\_reward)))

* Stop the Training (optional): From several testing, we observed that: The AI reaches the vest for sure when avg\_reward reaches 1500.

**if** avg\_reward **>=** 1500:

**print**("Congratulations!! Your AI wins!")

**break**

* If the average reward, avg\_reward reaches 1500 or higher, we can be 100% sure that the AI reaches the vest.
* To stop the training and to end all the process we'll use a **break** statement.
* Close the DOOM environment (optional): When the game finishes and the AI wins, we can close the DOOM environment.

# *Closing the DOOM environment*

**doom\_env.close**()

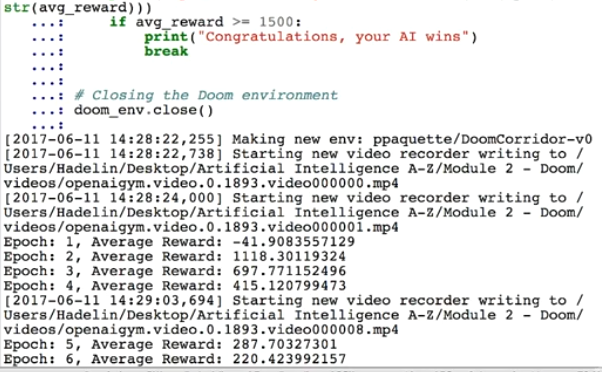
* To do this we take our DOOM environment **doom\_env** and apply **close()**. It's better to do once you've finished the game.
* Congratulations!! We're all done now. It's going to be pretty exciting!!! It's time to play with it and have fun!
* Let's hope we can reach the vest as fast as possible. That's the goal of this DOOM environment.

**Test the AI**

**3.18 Step 18: Test the AI**

To get the results we simply *select the whole code* and *execute*. Note that, we have also the *"videos" folder*, where we can watch our AI playing DOOM. This folder is going to be populated with some videos in real time.

* This is the first attempt that our AI playing the game:



* Notice the *first average reward* is ***negative*** it's got -41.9. That's pretty bad but that's totally normal, because the AI is ***exploring*** and ***training***. Remember our goal is to get the average reward of 1500.
* Also notice we've got 1118.3 in the second epoch, which is just by ***luck***. However the reward *decreases* and then it'll *increase*.
* What does 1500 mean? If we look at the OpenAI's page, we'll see if we reach to the vest as soon as possible, we'll get the maximum reward.
* So if the average reward reaches 1500, it means we're 100% sure that out AI reached the vest. This is a special case, that the AI figured out the best strategy that: It doesn't have to kill the monsters, just reach the vest as soon as possible.
* Or maybe the strategy is to *avoid the monster* or to *run very fast avoiding the monster* and reach the vest. But the main goal is: *reach the vest as soon as possible*.
* We noticed after second epoch (where we got average reward of 1500 by luck) the average reward is decreasing. It's because there is the training and AI is playing the game and exploring the strategies.
* Also we're using a continuous reward, when the *AI is approaching towards the vest* it's getting a *continuous positive reward*.
* If it's getting *further away from the vest* it's getting it's getting the *negative rewards*.
* For example at the epoch 1 we got -41.9 it got further away from the vest or it got killed.
* Quick overview of the code: First we import all the required packages and libraries.

Libraries and Packages:

# *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

* Brain of the AI: Then we make the brain of the AI. Inside the \_\_init\_\_() we defined the CNN (the eye of the AI). And forward() that propagates the signals to the brain.

Brain of the AI:

# 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

* AI body: Next we build the body of the AI, which is the part of the AI that plays the action. We used softmax to play the actions (benefit of softmax: since it uses probabilistic approach, the best actions will be played and the other actions also explored).
* Because we want to combine exploration and exploitation (otherwise we'll got stuck in local-minimum or local-maximum).

AI body (playing the actions):

# Making the body

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

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

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

        self.T = T

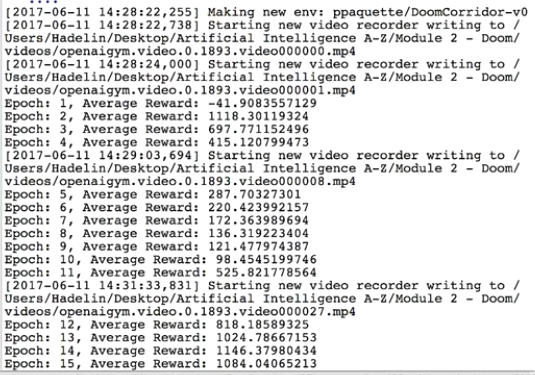
**def** **forward**(self, outputs):

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

        actions = **probs.multinomial**()

**return** actions

* Now notice the Exploration Phase is over; because notice: from **Epoch 2** to **Epoch 10** the *average reward is* ***decreasing***. But after **Epoch 11** the *average reward is* ***increasing***. So we can say from Epoch 1 to Epoch 10 was the *big* ***exploration*** *phase*. However during the exploitation phase we'll observe some small explorations also.



* So the ***exploration phase*** *is not totally over*, we can say: *"it's less likely to explore"*. Because dong some random exploration, the AI might find a ***better pattern***, that would lead it to a better reward.
* Full AI: After making the brain and the body, we've assemble those to make the whole AI. The \_\_call\_\_ is kind of ***forward*** ***function*** that propagates the whole signals from the input image (game screen), then through the brain and after that through the body to ***play*** the ***action***.

Full AI:

# 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**()

* Training Mechanisms: In this part we have implemented the Eligibility trace which is *n-step deep convolutional Q-learning*. Instead of learning on *each step* (instead getting reward, target every step), we compute a *cumulative reward* and a *cumulative* *target* on n-steps (we set it n=10, i.e. 10 steps) it improves the training.
* Then we make a moving average (MA class), it's similar to a score function

Training Mechanisms

# *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)

* Finally we train the AI, which is a (stochastic thing).
* Our inputs go into the NN and then we get an error, we compare the prediction and the targets and generates a loss,
* That loss is then back-propagated to the NN.
* Then we apply SGD to update the weights that try to minimize the squared distance between the predictions and the targets.
* Finally we print the progress to keep track of how our AI's doing.

Training

# *Training the AI*

loss = **nn.MSELoss**()

optimizer = **optim.Adam**(**cnn.parameters**(), lr = 0.001)

nb\_epochs = 100

**for** epoch **in** **range**(1, nb\_epochs + 1):

**memory.run\_steps**(200)

**for** batch **in** **memory.sample\_batch**(128):

        inputs, targets = **eligibility\_trace**(batch)

        inputs, targets = **Variable**(inputs), **Variable**(targets)

        predictions = **cnn**(inputs)

        loss\_error = **loss**(predictions, targets)

**optimizer.zero\_grad**()

**loss\_error.backward**()

**optimizer.step**()

    rewards\_steps = **n\_steps.rewards\_steps**()

**ma.add**(rewards\_steps)

    avg\_reward = **ma.average**()

**print**("Epoch: %s, Average Reward: %s" % (**str**(epoch), **str**(avg\_reward)))

**if** avg\_reward **>=** 1500:

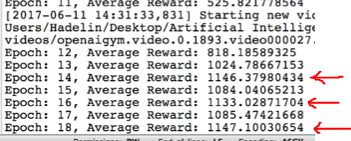
**print**("Congratulations!! Your AI wins!")

**break**

# *Closing the DOOM environment*

**doom\_env.close**()

* Notice the fluctuations, that means our AI is still doing some exploration



* It is still exploring a little bit because it's definitely reaching an average reward near 1200. It’s trying to reach 1500.
* ***Now let's observe the videos:*** Our video will record randomly.
* ***1st video:*** We see our AI is able to kill some zombie but then it failed to pass other zombies.



* ***2nd video:*** It might not be much better. But we've got lucky, our AI is able to kill all the zombies but failed to reach the vest:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

* Remember, we've got average reward 1118 at epoch 2, (which was a random luck). Above video is from that epoch. Notice that, our AI agent was *able to kill all the zombies* but it *failed to reach the vest*.
* ***However, the best strategy is:*** *reach the vest as fast as possible avoiding all the zombies* (i.e. you don’t need to kill the zombies). Our AI agent doesn't know this strategy yet.
* ***3rd video:*** It's similar to the first video, where our AI is *failed to kill* all the zombies and *failed to reach the vest*. We also observed that our AI agent also *done better* compared to the first video.



* ***4th video:*** We notice that, the AI agent gets to the vest pretty much right away. Because the *faster* you get to the vest the *more* *positive reward* you get.
* ***5th video:*** We have the same thing here; the AI gets to the vest as fast as possible.
* Now we can reduce our average reward limit from 1500 to 1250, because from those observations we know now that our AI is able to reach the vest if it got 1250 average reward or more. However, from 25-30 epoch it's possible to reach 1500 average reward.
* ***Implement ϵ-greedy policy (advanced):*** We can also implement -greedy policy instead of soft-max. We can do that later for practice.
* ***Now one point is:*** Big gaming companies don't implement AI like ours to their games (for example FarCry, Death Stranding etc) because humans cannot beat an AI opponent, and if a customer (human) cannot win a game, they lose their interest to that game. And that is bad for the Gaming Companies.

In the next chapter we'll build a NN with A3C model to beat the BREAKOUT game. Note that the Breakout game is more difficult compared to other 2 AI we built , because the Deep-CNN has to detect the small ball (in Doom, the monsters/zombies were big and easily detectable).

**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)

# *Training the AI*

loss = **nn.MSELoss**()

optimizer = **optim.Adam**(**cnn.parameters**(), lr = 0.001)

nb\_epochs = 100

**for** epoch **in** **range**(1, nb\_epochs + 1):

**memory.run\_steps**(200)

**for** batch **in** **memory.sample\_batch**(128):

        inputs, targets = **eligibility\_trace**(batch)

        inputs, targets = **Variable**(inputs), **Variable**(targets)

        predictions = **cnn**(inputs)

        loss\_error = **loss**(predictions, targets)

**optimizer.zero\_grad**()

**loss\_error.backward**()

**optimizer.step**()

    rewards\_steps = **n\_steps.rewards\_steps**()

**ma.add**(rewards\_steps)

    avg\_reward = **ma.average**()

**print**("Epoch: %s, Average Reward: %s" % (**str**(epoch), **str**(avg\_reward)))

**if** avg\_reward **>=** 1500:

**print**("Congratulations!! Your AI wins!")

**break**

# *Closing the DOOM environment*

**doom\_env.close**()