Chapter – 3 (implementation – Part 3 – SoftmaxBody & AI)

**Artificial Intelligence**

**AI-app (CNN-DQ): DOOM Game-play** (part 3)

SoftmaxBody class: \_init\_ fn, forward fn

AI class: \_init\_ fn, \_\_call\_\_ fn

**Making the AI Body: SoftmaxBody class**

**3.7** step 7: **SoftmaxBody class [p1]: \_init\_()**

Previously we made the brain of our AI, now let's make the body.

* Body: The body is the part where we define how the actions are going to be played.
* Our signals are coming from the AI brain. The output signals returned from the forward function of CNN class (the brain).
* First the *images* go into the *eyes* of the *neural network* (i.e. the ***three convolutional layers***) and then with the ***fully connected layers*** we get the ***output signal*** from the ***brain*** which contains the Q-values.
* Next, that output signal should be forwarded to the body and the *body will play* the action. And that's exactly we're going to do right now.
* We're going to ***implement the way*** the body will play the action. We'll use softmax method to do that (as we did for self-driving car).
* We'll create a class called SoftmaxBody which will correspond to the *body of the AI* (we cannot give the class-name Softmax, because it's a function from PyTorch from nn module).

Now our CNN ***convolutional neural network class*** is the brain and SoftmaxBody is the body of the AI.

* ***Inheritance:*** We'll inherit from nn.module, actually we're not going to use it but we can still inherit from it - in case we want to improve this SoftmaxBody class and we can use the tools from nn.module.
* **\_\_init\_\_**: We'll use following parameters:
* **self**: Refer to the SoftmaxBody object itself.
* **temperature**: Temperature **T** will allow us to ***modulate*** how the ***neural network will be sure*** of which action it should decide to play.
* This parameter will be a positive number and:
* The closer it is to zero ,the neural network will be less sure when playing an action
* Higher the temperature parameter, neural network will be more sure when playing an action

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

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

        self.T = T

* Since we inherit from nn.module we're going to use the **super**() again. We'll use **super**(SoftmaxBody, self).**\_\_init\_\_**()
* To set the temperature variable we use: self.T = T
* Next we'll implement the forward() function, because we have to forward the output signal from the brain. i.e. the Q-values contained in the ***output neurons*** of the ***output layer*** (of the brain/NN) to the body (which will play the action).
* So we are forwarding the output signal from the brain to the body that will play the action (actions like: moving forward/left/right, turn left, turn right or shoot).

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

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

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

        self.T = T

**3.8** step 8: **SoftmaxBody class [p2]: forward()**

Now we're going to make the forward function which will *propagate* the output signals of ***AI-brain*** to the ***AI- body*** so that it will play the right action to reach the vest.

* But there is no ***right-action*** yet because there is no training yet. We have not trained the AI yet but this is exactly what we will do in next part (**from step 11**) by Implementing Deep Convolutional Q-learning where we'll *Train the AI with Deep Convolutional Q-learning*.
* Let's forward the signal from the output layer of the NN (AI-brain) to the AI-body (SoftmaxBody).

***Arguments:***

* **self**: Refer to the SoftmaxBody object itself.
* **outputs**: We want to ***forward*** the ***output signal*** of the *brain* to the *body* and therefore the ***input*** will be the output signal of the brain.
* **outputs** corresponds to the *output signals of the brain* after the ***input images*** are ***propagated*** through all the CNN to reach the output layer which is **x**, returned by the **CNN**'s *forward function* of the *brain*.
* That output signal of the brain will be forwarded to the body with new forward function of SoftmaxBody.
* We're going to use **F.softmax()** method from functional module to play the action. Using the signals from the Brain of our AI the body will play the actions with the softmax technique.

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

* It is exactly the same as what we did for the self-driving-car. First, we're going to get our ***distribution of probabilities***. Then we're going to sample an action according to that *distribution of probabilities*.
* We copied following code from our previous self-driving car file. Which is the select\_action() function of the self-driving car. We are doing the same thing for the **forward()** of our SoftmaxBody class.

**def** **select\_action**(self, state):

        probs = **F.softmax**(self**.model**(**Variable**(state, volatile = **True**))\*100) # *T=100*

        action = **probs.multinomial**()

**return** action.data[0,0]

* Following is our **forward()** of our SoftmaxBody class.

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

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

        actions = **probs.multinomial**()

**return** actions

* probs is our probabilities. It is a ***distribution of probabilities*** for each of the **Q-values** which depend on the *input image* and each *action*.
* So we have **1 Q-value** for each of the six possible actions (moving forward/left/right, turn left, turn right or shoot) and therefore we get a ***distribution of 6 probabilities***. So there is a probability for each Q-value associated to each action.
* We're using **F.softmax()** from functional module F.

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

* outputs \* self.T are the elements for which you want to create a distribution of probabilities.
* outputs are the Q-values we're getting from the neural network (CNN, brain).
* ***Temperature T:*** We multiply the **Q-values** (outputs of our brain, CNN) with temperature self.T to customize the exploration.
* High temperature T means less exploration of the other actions. The best action will be selected with a higher probability as opposed to the other actions which will be selected with lower probabilities.
* That's exactly similar that we did for self-driving-car.
* We're creating this ***distribution of probabilities*** to be able to ***explore*** the ***different actions*** instead of directly picking the maximum Q value.
* If we directly pick the maximum Q-value (outputs of our brain, CNN), and we don't explore much the other actions, we might miss something.
* But with this **softmax()** method we can do some ***more exploration*** and therefore maybe find some ***hidden solutions*** in the patterns that might be much better.
* How are we going to use these probabilities?
* Well we're going to sample the final action to play from this distribution of probabilities. To do this, we have to use the multinomial() function to sample the action according to this distribution of probabilities (probs).

actions = **probs.multinomial**()

* The variable actions will become *actions that will be played* by the body of our AI.
* probs.multinomial()is the sampled actions from our probability distribution probs.
* Finally we return the actions.

**return** actions

So now the forward function, therefore the body of our AI is ready:

**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

* We have now our AI brain & body!! Next we'll assemble them to make the future AI. So that it'll have intelligence and a body to play the right actions.
* But also, we have to train it's intelligence to play the game of DOOM. That's what we'll do in: "Part Two- Training the AI" in next steps (**from step 11**) by Implementing Deep Convolutional Q-learning where we'll *Train the AI with Deep Convolutional Q-learning*.

Next we'll assemble the AI by creating an AI-class. It's again going to be a class of two functions.

**AI class (action)**

**3.9** step 9: **AI class [p1]: \_init\_()**

Now we have two classes: ***AI-brain*** and ***AI-body***. We're going to assemble them by creating the ***objects*** of those ***two classes*** and use those objects to create a bigger object. The bigger object will be the AI.

* We are going to make the AI by assembling the two classes that we created, i.e. the brain and the body.

class AI:

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

        self.brain = brain

        self.body = body

* **AI** is our ***new class*** which is our ***final assembled AI***.
* **\_\_init\_\_** is taking 3 arguments:
* **self** refers to the object itself.
* brain: It is an object of the **CNN** class, i.e. the brain of our AI.
* body: It is an object of the **SoftmaxBody** class, i.e. the body of our AI.

That's the two elements that we need to build an AI. We need a brain: *the neural network* and a body: *which will play the action* with soft-max.

**3.10** step 10: **AI class [p1]: \_\_call\_\_()**

Now we make a ***big forward function*** that will do the whole propagation, since our AI is assembled but we have *two forward functions* (one for the brain and one for the body), we're not going to use those two separately.

* In this **AI** class we are going to make a big forward function which will take the images as input then we ***propagate the signals*** in the brain.
* For this we'll use the *first forward function* (from CNN class) and then once we get the ***output signals*** of the ***brain*** we will *forward* these *output signals* into the ***body*** with the *forward function* of **SoftmaxBody** class (which will use the ***soft-max*** technique).
* And then eventually we will ***return*** the ***actions to play***.
* Note that: We're not going to call this next function forward.
* We're going to use the **\_\_call\_\_** function to call the two forward functions: from the brain and the body to propagate the signal from the *very beginning with the* ***input images*** to the *very end with the* ***actions to play***.
* So this the **\_\_call\_\_** function will combine the two forward functions of the brain and the body. Then we will have our AI ready.
* This forward function will *propagate* the *signals* from the *very beginning* when the *brain is getting the image* to the *very end* when the *AI* *plays the action*.
* This whole function is going to be our last step before we move to the next part: ***Training the AI with Deep Convolutional Q-Learning***.
* **\_\_call\_\_():** It is similar to the **\_\_init\_\_** function, i.e. it's an ***existing function*** but this time we use it to call some other functions. We'll use it to call the *forward function* from the brain and the other *forward function* from the body.
* \_\_call\_\_ is going to take two arguments:
* **self:** refers to the object itself.
* **inputs:** Since, we are doing the ***whole propagation*** this time, so the input will be the input images, because that's the *starting point* when the ***AI*** is ***playing*** the ***game***.

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

* It is first visualizing the images of the game
* then propagates the signals in the brain and
* then plays the action.

Therefore the second argument is going to be inputs. And now we are ready to make this whole propagation.

* Converting the image into the right format: The first step is receiving the input images from the game. And since these images are going to ***enter*** the ***neural network***, we have to format them in a **torch** structure.
* We will convert these images into a numpy array then we will convert the numpy array into a torch tensor and then finally we will put the torch tensor inside a torch variable that will contain both the tensor and a gradient.
* That's for our ***dynamic graphs*** to compute very efficiently the ***gradients*** later for ***Stochastic Gradient descent***.

Once we get the right format of our images, then we'll be able to enter it into the *neural network* and then we'll do the whole *propagation of the signals*.

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

* **input**: It is the real input of the neural network. We take the **\_\_call\_\_**()'s argument inputs (original image) and convert it to a numpy-array using np.array().
* **dtype = np.float32**: since the cells of the numpy-array will contain the pixels, it is actually safer to specify the data-type as float.
* So np.array(inputs, dtype = np.float32) will convert the original image inputs to a numpy-array of type float.
* Also note that, the ***tensors*** are by definition ***arrays of a single type***, so we choose **float32** to be a single type float.
* Convert the numpy-array to a torch-tensor: Now we convert the numpy-array np.array(inputs, dtype = np.float32) into a Torch-tensor.

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

* To do this we can use **torch.from\_numpy**() function.
* Convert torch-Tensor to torch-Variable: A torch-variable contains both the ***tensor*** and the ***gradient***. To do it- we take our Variable **class** and we use the torch-tensor as an input.

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

Finally the **input** are allowed to enter the neural network (i.e. first the eyes- CNN and then the fully connected layers to lead to the predictions of Q-values).

* ***Propagate the input (i.e. signals) through the Brain:*** We're going to propagate these formatted images into the CNN (eyes of the AI) i.e. through the ***three convolutional layers***.

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

* self.brain(input)will propagate the **input** inside the brain of AI (i.e. *convolutional layers* and *fully-connected layers*) and since the forward function of the brain returns the Q-values as *output-signal*, our variable output will store those *output signals* from the *brain of* *AI*.
* ***Propagate*** *the* ***output*** *(i.e. Q-values) through the* ***Body of AI****:* Now we have to propagate the output signals (i.e. Q-values) from the Barin to the Body of the AI.
* To do this we're going to use the second forward function from the body (*forward function* of **SoftmaxBody** class).
* The forward function of the body takes the ***output signals*** of the ***brain*** as ***input*** and then it returns the actions as output.

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

* Finally we return the action

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

Since the **actions** is in the ***Torch-variable*** format, we need to convert them back into **numpy** array. So we applied **numpy()** function on the **data** structure of the **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**()

So what we did above is: ***forward*** the input ***images*** through the brain (we propagated the signals inside the brain) and get **Q-values** as ***output*** and then we ***forward*** those ***Q-values*** through the body (we propagate the Q-values into the body) and we get the ***action to play*** as final output.

* Therefore we build the AI in three steps:

1. First we made the brain
2. Second we made the body and
3. Third we assembles the brain and the body and we propagated the whole signal from the eyes to the moment we play the action.

* Now we need to train AI to be intelligent.
* To do this we're going to use the ***rewords*** from the ***Doom-Environment***. Because it's ***learning*** from the ***rewards*** by being ***reinforced*** when it gets a ***good reward***, and by being ***punished*** or ***weakened*** when it's getting a ***bad rewards*** that's where the ***Q-learning*** will come into play.

And so that's exactly what we'll do in the next part: training the AI with deep convolutional Q learning.

**All code at once**

* Libraries:

# *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

* CNN class *(BRAIN of 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

* SoftmaxBody class *(BODY of AI):*

**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 *(assembling BRAIN and BODY & propagate the signals):*

# *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**()