Chapter – 3 (implementation – Part 2 – CNN class)

**Artificial Intelligence**

**AI-app (CNN-DQ): DOOM Game-play** (part 2)

CNN class \_init\_ fn

count-neuron fn

forward fn

**CNN class**

**3.3** step 3: **CNN class, \_init\_ [p1]: 'create 5 variables'**

To build our AI, first we make the brain then we make the body and then we assemble the two to make the AI.

* The brain is the neural network
* In the body we'll define how the actions are going to be played.
* After implementing the brain and the body we will assemble them to make the AI (in the last section).
* We'll make a ***class*** for the ***brain*** in four steps. The class will contain ***several functions*** to implement ***CNN***, ***forward-propagation*** and output. Once we made that class, we'll be able to create as many brains as we want by just creating some objects of this class.
* CNN class: We named the class '**CNN**' because the brain is a CNN network. Like the previous ***self-driving car***, our class is going to inherit from the **torch.nn** module.

**class** CNN(nn.Module)

* **\_\_init\_\_():** This \_\_init\_\_() will define *all the variables* of the future brain, the CNN objects. We use following arguments:

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

* **self:** refers to the object itself.
* **number\_actions:** it's the number actions in the Doom environment.
* Actually this variable is not compulsory for the **init** function, we used it so that we can test our AI on other Doom-environments. We will import this **number\_actions** from the Doom-environments with **ToDiscrete** (imported from ppaquette\_gym\_doom.wrappers.action\_space) and when we doing that, we will input the name of the environment **doomv0**. We did imported **ToDiscrete** at the beginning

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

* If we want to test our AI on other Doom-environments and play other games we won't have anything to do because this **number\_actions** will directly get the *number of actions* from the *Doom-environments* you'll be playing with.
* Activate the Inheritance: The first thing to initialize is to activate the inheritance with the **super()** function.

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

* **CNN**: the class that will define the neural network.
* **self**: to refer the object.
* We apply the **init** function of super class. And now we can use all the tools from the **nn** module.
* NN architecture: We are going to build a convolutional neural network (CNN) because this time the AI will have eyes and these eyes will be the ***convolutional layers*** of this CNN.
* The *AI visualizes* the images with the *convolutional layers* it will pass on the signals into a classic *Artificial Neural Network (ANN)* with fully connected layers. And that's where it will try to predict the Q-values for *each possible actions* that we can play.
* This architecture will have some convolutional layers and then some fully connected layers and this will be the brain of our AI.
* ***Following variables are for the layers:*** We are going to build a CNN with *three convolutional layers*. Hence the first *three variables* represent *three convolutional layers*. Then we'll have *one hidden layer*, that means that we will need ***three*** *convolutional connections* and ***two*** *full connections*.
* Therefore we'll define five variables, three for the convolutional connections and two for the full connections.

self.convolution1

self.convolution2

self.convolution3

self.fc1

self.fc2

* self.convolution1, self.convolution2, self.convolution3, are the convolution connections.
* convolution1 will apply convolution to the input images to get the *first convolutional layer*.
* In this *convolutional layer* we'll get some *new images*, ***each*** of them ***detecting*** one ***specific feature*** in the input images.
* convolution2 will take the *first convolutional layer as* input and by applying again some *convolution*, it will create a *second* *convolutional layer*.
* We will apply convolution2 to connect those new images from ***first convolutional layer*** to some *new images of a second convolutional* *layer*. These new images from ***second convolutional layer*** again will detect some features in the *images of the first convolutional layer*.
* So it's just to reinforce the feature detection.
* Then to the images of the ***second convolutional layer*** we applied the ***third convolutional layer*** to get some more images that detect even more features inside the input images.
* So the more we apply convolutions to the different layers of images, more features we are able to detect.
* That's how by *detecting features* the AI will *understand* the *location* of the *monsters/walls/obstacles/path* and play the game.
* Flattening layers: After the convolutional layers we have to ***flatten all the pixels*** obtained by the *different series of convolutions*. By flattening all the arrays of pixels we'll get a very-large vector that will become the ***input*** of a classic ***artificial neural network (ANN)***.
* That's where we get our ***fully connected layers*** and therefore our ***full-connections***. self.fc1, self.fc2 are the variables for those fully-connected-layers.
* Because we're going to have one hidden-layer to implement ANN and therefore we need one full connection **self.fc1** for this.
* We also need a second full connection between the hidden layer and the output layer **self.fc2** composed of the output neurons that are the Q-values.

Next we'll define these five variables, using the classes of the **torch.nn** module.

**3.4** step 4: **CNN class, \_init\_ [p2]: 'define 5 variables '**

* **convolution1** applies convolution to the ***input images*** (i.e.original images) and we're going to use **nn.Conv2d** to do this, because we're working with ***2D images***.

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

* **in\_channels:** Correspond to the input of the convolution. That's going to be the number of channels in our images. In this Doom-game we are going to work with black & white images the AI is totally capable of recognizing the monsters by their shape in ***black*** & ***white***.
* Therefore we're going to use one channel in\_channels = 1. For *color images* you should use *three channels*.
* **out\_channels:** Correspond to the output of the convolution. out\_channels is equal to the ***images you want*** to have in the convolutional layer which is the output of this **self.convolution1**.
* Basically this is equal to the number of features you want to detect in your *original images*
* Because we will create "one image" per feature we want to detect. Because we applied one *feature detector* to the *input* *image* to detect a *specific feature* in the *input image* and
* Therefore the ***number of output images*** here is the ***number of features*** we want to detect. That's why out\_channels=32.
* A common practice is to start with ***32 feature detectors*** and so that will lead us to 32 processed images in this *first convolutional layer*.
* So the input is ***one black and white image*** *(a real image)* and the output in the first convolutional layer is 32 processed images.
* Here the "processed" mean that the *convolution* was *applied* to the *input image* to get ***32 new images*** *with* ***detected features***.
* **kernel\_size:** kernel\_size is the ***dimension*** *of the* ***square*** that will go through the ***original image***. In common practice, we use either or or .
* For the first layer we're going to use a dimension ***feature detector***. And then we will reduce the size of this ***kernel*** for the ***next convolutional layers***. So we set kernel\_size = 5.
* There are other parameters: stride, padding, bias etc, and we have default values for all these ones.
* **convolution2** : Its mostly similar to **convolution1**.

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

* **in\_channels:** The ***input channel*** of the second convolutional layer is the ***output channel*** of the first convolutional layer. Therefore we set in\_channels = 32. Because we have *32 images* in the *input convolutional layer* of **convolution2**.
* **out\_channels:** The second convolution **convolution2** is applied to this *second convolutional layer* "in\_channels = 32" to return a *third convolutional* layer "out\_channels = 32".
* i.e. it'll create 32 new images (32 is actually a very common number in CNN),
* **kernel\_size:** we need to reduce the kernel size i.e. the dimensions of our feature detector - square. We set a smaller size 3, kernel\_size = 3.

Therefore, our second convolution is ready. It takes ***32 processed images*** as ***inputs*** (output from first-convolution layer, each one detecting a feature of the original input image) and it creates 32 new images using the feature detector of reduced size.

* **convolution3** : Similar to **convolution2**. We'll create a third convolution to detect more-features. We'll take **32** as input images and now we'll output **64** processed image with smaller kernel size kernel\_size = 2.

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

So now we have a very classic architecture of a Convolution layer and it's very efficient to have a high level of feature detection inside images.

* ***Hidden layer*** fc1: Now we have our three convolutional layers with *three convolution connections*, now it's time to build two full connections.
* After applying three convolutions, we've got ***total*** ***processed images*** (each detecting a specific feature).
* After flattening those processed images we'll obtain a huge vector: We ***flatten*** all the ***pixels*** of these images and we obtain one ***huge vector*** that will become the input of a new *Fully Connected Neural Network*.
* So we have to make the ***full-connections*** between this ***huge vector*** and a ***hidden layer***.
* Then a ***second full-connection*** between the ***hidden layer*** and the ***output layer*** composed of the output neurons (each neuron corresponding to a Q-value of the possible actions).
* We'll use **nn.Linear** because the full connection is an object of the Linear class. Notice we have different parameters:
* **in\_features:** It's the ***number of pixels*** in the *huge vector* that we got after *flattening* all the *processed images*. We don’t know the numbers right now, so well make a function later to calculate this number. For now we call it number\_neurons.

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

* **out\_features:** out\_features is the number of neurons in a hidden layer. We set this value ourselves, it depends on the architecture of the neural network. In this case, for example 40 neurons might be fine.
* We can increase it later. If the training is not too slow we can increase it to improve the predictions.
* Here we need to understand the ***mindset of programming***,
* To overcome current obstacle, you can simply ***input any name*** here that will represent the number of neurons
* Then afterward we will simply make a function that will return this number\_neurons variable, the number of pixels that we're looking for.
* We are totally allowed to do that even if the function comes afterwards.
* It's a typical programming thinking you must have when you get this kind of obstacle. Where you can make a function to get what you're missing.
* ***Hidden layer*** fc2: For the second full connection, i.e. the connection between the hidden layer and the output layer.
* **in\_features:** It is 40, the out\_features of the previous layer.
* **out\_features:** It is the ***number of output neurons***. Since each output neuron corresponds to ***one Q-value*** and one Q-value corresponds to ***one action***, therefore the output neurons here is equal to the *number of actions* number\_actions parameter of the current CNN class.

That’s the architecture of our neural network. Our neural network is composed of three convolutional layers and one hidden layer in one big CNN. This CNN will detect the features in the game so that the AI will know what it has to do, where it has to go and where it needs to shoot.

Next we're going to get the number\_neurons using a function.

**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 = number\_neurons, out\_features = 40)

        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)

**3.5** step 5: **CNN class, count-neuron()**

Now we make a function that will count the neurons in the huge vector after the convolutions are applied. We call this function **count\_neurons**().

* arguments:
* self: indicating object itself.
* image\_dim: The number of output neurons in the flattening layer depends on the dimensions of the original input image.
* The **dimensions** of the **input images** that we input in the CNN from **Doom** is going to be .
* Actually we're going to ***reduce*** the size of the original images to before feed into the CNN.
* So image\_dim is going to be a tuple (1, 80, 80), here **1** corresponds to **Black & White** image i.e. one channel.

That's the only argument we need. Now let's count the neurons.

* Create a fake image: We don't have any input image right now, i.e. *doom image* that we can *import*. We're going to do that later.
* So we have to create a ***fake image*** that has dimensions with ***fake pixels*** and that will still give us the desired number. Because that number ***only depends*** on the dimensions and ***not on*** the pixels inside the images.

So firstly we create a fake image, and then we will compute the number of neurons that we want.

* We call this fake image **x**. The trick to create a fake image is to use the torch.rand(). Because we're going to put some random pixels in these images.
* Then inside of torch.rand() we're going to input the damnations i.e. (1, 80, 80)of the images.
* But since we're going to ***input*** this image into the ***Neural Network*** and as you remember the neural network can only accept batches of input states i.e. the batches of input images.
* We are going to create that fake dimension directly into this rand() function.

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

* Here 1 correspond to the batch and \*image\_dim. Since the dimensions are contained in image\_dim argument of the **count\_neurons()** function.
* But in order to pass the elements of the touple, we used '\*' before image\_dim . It'll allow us to pass the elements of the image\_dim touple as arguments to rand() function.
* Lastly, we need to convert this x into a torch variable because this is going to be the input vector of the neural network. So we've used Variable().

All right so it'll create an image of fake pixels that will have *nothing to do with the real-doom-images*. Using this fake-image we'll be able to get the final number of neurons.

* This fake image **x** now represents an ***input image of random pixels*** that was just converted into a *Torch-Variable* to feed into the *Neural* *Network (convolutional layers)*.
* Since we only need the ***number of neurons*** after the ***convolutions*** are applied, we will just go up to the convolutions 3. We will not go into the *two full connections*.
* The number of neurons that we want is between self.convolution3 and self.fc1.
* Propagate the fake image through the convolutional layers: Now we have *one input image* with the *right dimensions*, it's time to ***propagate*** this image into the ***neural network*** to reach the ***flattening*** ***layer***.
* Then we're going to get the neurons in the flattened layer and we'll get the ***number of neurons*** in this ***flattening layer***.
* Here we have to do is exactly what we do in a ***forward function***. We need to propagate the signals into the ***NN*** but only in the ***convolutional layers*** until we reach the flattening layer.
* First convolutional layer: We're going to update **x** (input) and it'll will become the *first convolutional layer*. We have to do this in three steps process:

Step 1: We apply ***convolution*** to the *input images*.

Step 2: We apply ***max-pulling*** to the *obtained convoluted images*.

Step 3: We ***activate the neurons*** in this *pulled convoluted images*

**x** will become this ***First Convolutional Layer*** composed of all these ***Poolled Convoluted Images***.

* Step 1: So let's do this first step apply the first convolution **convolution1** to the input image **x**.

self**.convolution1**(x)

* Step 2: We are going to apply max-pooling using **F.max\_pool2d()** to our convoluted images returned by self**.convolution1**(x). Here **F** is the functional module.

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

But this **F.max\_pool2d()**takes additional arguments which are:

* kernel size: That's the ***size*** of the ***window sliding*** through your images and that will take the ***maximum*** of the ***pixels*** in each slide. It will still detect the features because the features are associated to a high value of the pixel in the arrays. We choose **3**, it’s a common choice for the kernel size.
* Strides: Are the *Steps* at which we're moving this whole filter-window kernel. It determines how many pixels the window going to slide in the images. We are going to take a stride of **2**, i.e. the window moves 2 pixels per slide. Again that's a common choice.
* Step 3: activate all the neurons in this pooled and convoluted images in this first convolution layer. To do this we are going to apply **F.relu()** another function from functional module. It's the ***Rectifier Activation Function***. So we apply **relu** to our pooled convoluted images.

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

* ***Remember the way:*** first we ***apply the convolution*** to our input images then we ***apply Max pulling*** to our convoluted images obtained with a convolution. And then we ***activate the neurons*** in all this *pooled convolutional layer* with the ***rectifier activation*** function .

So we get our first convolutional layer on which we applied max-pulling and the neurons are now activated. Basically it ***propagates the signals*** from the *first convolutional layer* to the next one.

* Second convolutional layer: We're going to do the same thing to the ***second convolutional-layer*** as we just did on the first convolutional-layer.
* We'll propagate the signals further into the neural network through the second convolutional layer by activating the neurons of the layer.
* Currently x is the ***first convolutional-layer***, so we are going to apply convolution to this **x** and apply ***max-pulling***. After that we'll apply ReLu to activate the neurons. As a result we'll get the ***second convolutional-layer*** as **x**.

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

With this line we propagate the signals from the ***second convolutional-layer*** to the ***third convolutional-layer***.

* Third convolutional layer: We're doing the same thing again to get the ***third convolutional layer***.

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

* Now we have our signals propagated up to the third convolutional layer and that leads us to the flattening layer. So it's time to get our ***flattening layer***.
* We're going to flatten all the pixels of this ***third convolutional layer*** (i.e. ***all*** the ***pixels*** of all the channels and put them *one after the other* in a ***huge vector***).
* This ***huge vector*** is the flattening layer and at the same time we will use a trick to get the number of neurons in this flattened layer. That's what we're looking for. That's the number of neurons we're missing.
* Therefore we directly ***return what we want*** and in this **return** we're going to flatten the third convolutional layer and get the ***number of neurons*** at the same time in this flattening layer.

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

* Here x represents the ***third convolutional layer***.
* We're going to take all the channels of the ***third convolutional layer*** and we're going to use a function **size()**to flatten all the pixels of all those ***channels*** in one ***same huge vector***.
* **x.data** takes all the data of **x**, because x is a torch variable and it has *special structure*, therefore first we need to access it with **data**. Then we apply view() to look what's inside of it.
* Basically using **x.data.view**(1, -1).**size**(1) we take *all the pixels of all the channels* and we *put* them *one after the other* in this huge vector (that's basically what the size(1) does) which will be the input of the fully connected network.
* And from this huge vector we can get the number of neurons that we're looking for.

**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)

* ***Use the returned number of neurons in* \_\_init\_\_*:*** Now we go back to **\_\_init\_\_()** function and modify the following line of code

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

to

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

**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 = number\_neurons, out\_features = 40)*

        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)

* Because now self**.count\_neurons**((1, 80, 80)) returns the number of neurons. Where the image is of the size: and in ***one channel*** i.e. black & white.
* Don't forget use self because **count\_neurons()** is actually a method of our CNN class.

Finally we get the ***architecture of the neural network*** and with **count\_neurons()** we can try some other architectures and we don't need to count the number of neurons manually.

* Just use **count\_neurons()** function ***apply*** it to the ***format of your images***. And this will get you the number of neurons in the flattening-layer without having to do anything, and whatever the architecture is.

The next function we'll implement is going to be the main forward function.

So we are going to propagate the signals ***from the beginning*** of the brain (i.e. from the eyes of the AI) to the ***output layer*** i.e. ***after*** the ***second full-connection***.

**3.6** step 6: **CNN class, forward()**

The forward function will propagate the signals in all the layers of the neural network including the ***3-convolutional layers*** and the ***fully*** ***connected layer***.

* It is similar to the ***self-driving car*** *(previous app)*, but this time we have to *propagate* the *signals* in the convolutional layers before the fully connected layer.
* The good news is we already *propagated* the *signals* in the convolutional layers in **count\_neurons**() function (previous step). So we'll also use that code in this forward() function.
* Then combining that code with the following code of **self-driving car**, we'll get the forward function for our CNN brain.

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

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

        q\_values = self**.fc2**(x)

**return** q\_values

* **forward(self, x):** For our CNN class we'll create **forward(self, x)**. Instead of state (self-driving car) we use **x**, which will be the input images at the beginning and then **x** ***will be updated*** as the signal is propagated into the neural network.
* We copy and ***re-use*** following code from **count\_neurons**(), which will *propagate* the *signals* in the convolutional layers.

**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))

* So now we need to propagate the signal from the convolutional layers to the hidden layer and then eventually to the output layer (i.e. end of the NN).
* To do this we first need to flatten the *third convolutional layer*, because the ***input image*** **x** is updated through the 1st, 2nd and 3rd *convolutional layers*. So right now at this stage x is the *third convolutional layer*.
* We'll do something quite similar as we did for **count\_neurons**(), only this time we *don't need the* ***number of neurons*** we ***simply*** need to ***flatten the channels*** in the third convolutional layer. So this will be quite more simple but very similar.
* To become the flattening layer, we update **x** as below:

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

* We're just updating **x** previous layer (third convolutional layer) usiing **x.view**(). We apply two arguments:
* **x.size(0):** It takes *all the pixels of all the channels* in the ***third convolutional layer*** and we put them one after another in this huge vector **x**, and it will become the input of the ***fully connected network***.
* **-1**: using **-1** is a pytorch trick to flatten a convolutional layer composed of several channels by using the size() function.
* Now **x** is our flattening layer, it is going to be the input of a classic fully connected network with a *simple linear transmission* of the signal.
* In this case, we don't use a ***convolution function*** to pass on the signal, we're going to use a ***linear transmission*** *with a* ***linear class*** and then to break the linearity (because images have non-linear relationships) we're going to use a ***rectifier function*** to be able to learn these ***non-linear relationships***.
* We'll update **x** again because we want to get the ***hidden layer*** now.
* We'll use our full connection **fc1**, because **fc1** connects the ***flattening layer*** to the ***hidden layer*** and therefore we need to take **fc1** and apply it to the current **x** (flattening layer).

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

* Don't forget to use self because fc1 is a variable of our \_\_init\_\_ function.
* self**.fc1**(x) ***passes on linearly*** the signal from the *flattening-layer* to the *hidden-layer*.
* But we need to activate these neurons while at the same time breaking the linearity. To do that we use the ***rectifier activation function***.
* We use **F.relu**() from functional module **F** i.e. the relu() is the *rectifier activation function*. We then pass self.fc1(x) as an argument of **F.relu**().
* In this line of code: x = **F.relu**(self**.fc1**(x)), first we propagate the signals from the ***flattening layer*** to the ***hidden layer*** of the *fully connected network*. And then we activate the neurons of this hidden layer by breaking the linearity with this *rectifier activation function* and we get our hidden-layer (i.e. updated x here).
* Now we have to propagate the signal from the hidden layer to the output layer with the final output neurons. That's *similar* to what we did with the *self-driving-car*.
* We take our second full collection fc2 and we *apply* it to of course the *neurons of the hidden layer* (i.e. **x** from fc1).
* So **x** here is the neurons of the hidden-layer (fc1).

x = self**.fc2**(x)

And now **x** becomes of course the output neurons of the output layer (fc2) containing the Q-values.

* And finally we simply ***return*** the ***output*** neurons (i.e. **x**) with the Q-values.

**return** x

Finally we just made a brain (CNN class) of our AI. Next we'll make the body that is defining how we're going to play the action after all the signals are processed in the brain.

**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:

*# 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