Chapter – 4 (implementation – Part 2 – Actor Critic)

**Artificial Intelligence**

**AI-app (CNN-A3C) "Breakdown" Game play** (part 2)

ActorCritic class, \_\_init\_\_(), forward()

SharedAdam Optimizer

**Actor Critic**

**4.4: ActorCritic class \_init\_ pt1**

Here we're about to make the ***A3C-brain*** (i.e. the brain of our AI).

* Note that, previously in the ***Self-Driving car*** project, we've made a simple brain with only *fully-connected layers*.
* Then in the second project for ***DOOM*** we've made a brain that not only had *fully connected layers* but also eyes because we added the *convolutional layers* and it could observe the images and understand what's going on inside the game.
* Now we're going to make a brain that not only will have eyes and fully connect layers but also memory.
* Because we're going to add a ***Recurrent Neural Network (RNN)*** inside the AI's brain and that will give a long memory. So that it can understand the *temporal relationships* and the *temporal properties* of the *input images*.
* Now we can see by building AI's with *deep-learning* and *deep-reinforcement learning* is all about getting closer and closer to how human brain works.
* We started with the basic relationships of the
* Brain with the ***linear full-connections*** (self-driving car).
* Then we added ***eyes*** and (Doom playing)
* Now we add the ***memory*** (playing Breakout).

With ***fully-connected-layers***, ***eyes*** and ***memory*** we have a really good and functional brain.

* ActorCritic class: This class is going to have a lot of properties with the *convolutions* and *LSTMs*.
* We're going to make \_\_init\_\_ function to *initialize* and *create* all these connections.
* Then we'll make the *forward function* that will *propagate the signal* inside the *brain* to *produce the output*.
* This ActorCritic class will be the brain of our AI which is based on the *actor-critic principle*, with separately the actor and the critic.
* We'll make *one linear full connection* for the actor and *one linear full connection* for the critic.
* ***Inherit:*** This ActorCritic is going to inherit from the torch.nn.Module so that we can use all the PyTorch tools.

**class** ActorCritic(torch.nn.Module):

* **\_\_init\_\_:** This \_\_init\_\_ function is going to take three argument self, num\_inputs, action\_space.
* self: the object itself
* num\_inputs: It is the input shape, i.e. the dimensions of our input images.
* action\_space: It's basically the space that contains all the actions. Also from this action space we can get the ***number of possible actions***.
* ***Appling the inheritance:*** We have to use the super() function to activate the inheritance so that we can use all the tools from the torch.nn.Module.

**def** **\_\_init\_\_**(self, num\_inputs, action\_space):

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

* We input our class-name "ActorCritic" as an argument to super().
* **super**(ActorCritic, self).**\_\_init\_\_**() gives us all the tools that we need from torch to build our brain.
* Convolutions: Now we'll make the eyes of our AI, i.e. the *convolutional layers*. We've already explained this in details in the previous project (DOOM playing AI).
* We'll use a very simple architecture with *32 feature detectors* of size 3x3, stride of 2 and a padding of 1. That's a pretty classic architecture but enough to understand what's going on in the breakout Game.
* We'll use self because the convolutions will be variables of the object.
* There's going to be four convolutions.
* In the first convolution, we start with our input images that has num\_inputs *dimensions* and we get *32 convoluted images*, *each* one detecting a *specific feature*.
* Then from these *32 convoluted images (from first conv. layer)* we apply the second convolution to get *32 new convoluted images*.
* Then same from these 32 new convoluted images *(from 2nd conv. layer)*, we apply the third convolution to get *32 new convoluted* *images* again.
* And then eventually from the 32 convoluted images *(from 3rd conv. layer)*, we apply the fourth convolution to get final features.
* ***First convolution:*** One input image goes to the *first convolutional layer* and outputs *32 convoluted images*.

self.conv1 = **nn.Conv2d**(num\_inputs, 32, 3, stride=2, padding=1) # *first convolution*

* We first input the dimensions of our input images i.e. num\_inputs input shape of the images.
* 32 is the number of feature detectors are also the number of kernels.
* 3 is the *size of the* kernel (i.e.the number of cells that will slide over the input image, 3 /4 /5 are common choices).
* For stride size we set 2.
* And padding 1.
* ***2nd , 3rd and 4th convolutions:*** So now we are ready to make the 2nd , 3rd and 4th convolutions. Those are almost the same as the first convolution.

self.conv2 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *second convolution*

self.conv3 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *third convolution*

self.conv4 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *fourth convolution*

* They will all have 32 *feature detectors* and outputs 32 convoluted images. Also the *input-dimensions* (output from previous convolution- *the left part of the convolution is the right part of the previous convolution*) are the same, i.e. 32.
* We'll set stride of 2 for each, padding 1.

With those four convolutions, our AI will have supervision, in the ***BreakOut game*** it will detect the ball very well. So that's it for the eyes.

* Memory: Now we're going to implement memory to our AI's brain (which we didn't have in DOOM). This will give our AI not only supervision but also it will have a long memory because we going to implement LSTM *(Long short term memory, a special kind of RNN that gives to your model some kind of a long memory)* so that our AI can learn some *long temporal* *relationships* from the *past*.
* We call this variable "self.lstm" because this will correspond to the LSTN network inside the AI's brain.
* ***What this LSTM part of the brain will do:*** This LSTM is used to learn the *temporal properties* of the *input* (input images). For example if the ball hits a brick the LSTM will encode the balance.
* First thing to understand is that: It will kind of *encode* what's *happening* in the game.
* Next important thing to understand when we implemented that LSTM is that: we need to choose an order of the temporal dependencies.
* Since we're going to feed our neural network with a *sequence of four images*, then that means that we can already learn some *temporal dependencies of order 4* i.e. some *temporal dependencies* where what happens at t+1 ***depends*** on what happens at time t, t-1, t-1 and t-3.
* Moreover, we're going to use an **LSTM**, therefore we will be able to learn some even *more complex temporal relationships*.

***Eg:*** we can learn some temporal properties where what happens at t+1 will depend on what happens at time t, t-1 , t-2 , t-3 down to t-n.

And that's the **Long** part in the **Long Short Term Memory** with this LSTM we can learn some very complex temporal relationships.

* ***LSTM:*** We're going to use the nn.LSTMCell from the nn module, it will represent the LSTM part of the neural network, by creating an LSTM object.

# *making an LSTM to learn the temporal properties of the input*

# *we obtain a big encoded vector S of size 256 that encodes an event of the game*

self.lstm = **nn.LSTMCell**(32 \* 3 \* 3, 256)

* It's also important to understand that for our AI-brain we're making a CRNN *(convolutional recurrent neural network)* and the RNN part comes after the CNN part
* Therefore right now what we need to input in this nn.LSTMCell is:
* First the *size of the output* after the convolution (in our case it's 32\*3\*3). This 32\*3\*3is actually the output after the previous *four convolutions*. Now it becomes the input of the RNN- the LSTM network.
* Why is the output of the four convolutions have the size 32\*3\*3?

It's actually not a simple formula but there is a *formula to compute this number of output neurons*, after flattening the pooled and convoluted images of the convolutions.

* Note that, in previous project (Doom), we made a function to compute this number (we made a count\_neurons() function to do that). We can reuse it here if we want.
* The second argument is going to be the *number of* ***output neurons*** of the LSTM. And we're going to set it to 256.

It means that now we have a vector that encodes *each event of the game* or in other words we have an ***encoded state***. Which can be used to make networks for the actor and the critic separately.

* Now we can make the separation between the actor and the critic. We're going to make actually two separate neural networks, one for the actor and one for the critic but they will be the *same* ***encoding*** *of the* ***images*** and the *temporal relationships* for these two neural networks.

self.conv1 = **nn.Conv2d**(num\_inputs, 32, 3, stride=2, padding=1) # *first convolution*

self.conv2 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *second convolution*

self.conv3 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *third convolution*

self.conv4 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *fourth convolution*

# *making an LSTM to learn the temporal properties of the input*

# *we obtain a big encoded vector S of size 256 that encodes an event of the game*

self.lstm = **nn.LSTMCell**(32 \* 3 \* 3, 256)

* So this is the common part for those two neural networks. This will be the same beginning for the two neural networks.
* After this things are going to change for the actor and the critic because we're going to make one linear-full-connection for the actor and one *different* linear-full-connection for the critic.

**4.5: ActorCritic class \_init\_ pt2**

After we made the four convolutions and the LSTM, we now have an encoded state. This encoded state is going to be the input of the two neural networks that we're going to make for the actor and the critic.

* We just need to create 2 *linear -full-connections*, one for the actor and another for the critic.
* But before we do that, we need to get the *number of possible actions*. We make a new variable num\_output, but it's not going to be a variable of the object. So we're not going to use self.
* This variable will represent a *number of possible actions* and we can get it from the *action space*.
* ***Number of possible actions:*** So we take our action\_space (input variable of \_\_init\_\_) we use action\_space.n to get this number of possible actions.

num\_outputs = action\_space.n # *getting the number of possible actions*

* The actor and the critic will take separately the *same input* which is an *encoded state* (i.e. the output of the whole process- CNN and LSTM).
* But they'll have two different *linear-full-connections* so that we get two *Neural Networks* one for the actor and another one for a critic.
* But since encoding already did the most of the part, now we simply need to create two objects: one *linear full connection* for the actor and another *linear full connection* for the critic.
* Critic: We're going to call this object critic\_linear and to create this linear full connection, we simply need to use nn.Linear() class, we set the input-neurons, which are the outputs of all the encoding (convolution and LSTM) including 256 neurons.

self.critic\_linear = **nn.Linear**(256, 1) # *full connection of the critic: output = V(S)*

* also we're going to have *one output*, because the output of the neural network for the critic is the value of the ***V function*** applied to the input state (to the inputs encoded states that we made using the four convolutions and the LSTM).
* So if we call the input state S (i.e. the output of all the convolutions and the LSTM, we call this encoded vector S) then the *output* of the *neural network of the critic* will be V(S) and therefore it has one dimension (it's just a value), so that's why we used 1.
* And remember V(S) what is *shared* among the *actors* so that they can get some *common information* that they can use to *play* *their* *action* in a more relevant way.
* Actor: We call the object actor\_linear that represents the *neural network* of the *actor*. Again, we already have the inputs encoded States, so we simply need to add a linear connection. As before, we'll use nn.Linear() class.

self.actor\_linear = **nn.Linear**(256, num\_outputs) # *full connection of the actor: output = Q(S,A)*

* This neural-network of the *actor* will take the *encoded state* that has the size of *256* (similar to critic).
* The *output of the actor* is going to be different, because the *output* of the *neural network for the actor* are the Q-values of the input states (the *encoded state* and *action* to play).
* If we call the encoded state S an action played A the output of this neural network actually there will be Q(S,A). Since we have ***one*** ***Q-value*** for *each action* then we have num\_outputs of *Q-values*. So the output is going to be num\_outputs.
* num\_outputs is actually the number of Q-values.
* ***Initialize the weights and bias for actor and critic:*** Now we have two separate neural networks, one for the critic and one for the actor. Now we have to initialize all the weights and bias of those two neural networks.
* To do that we're going to use the two functions that we made earlier (i. e. normalized\_columns\_initializer and weights\_init).
* ***Initialize random weights:*** To do this we're going to apply the weights\_init function to our object.

self**.apply**(weights\_init) # *initilizing the weights of the model with random weights*

by doing this we are just initializing some random weights to get a future optimal learning of these weights.

* Now we have to make a special normalization for the actor and the critic. But remember we're *not going to set a same* ***variance*** for the *actor* and the *critic*.
* The actor will get a ***small*** *standard deviation*, small variance.
* The critics will get a ***big*** *standard deviation*.
* ***And why do we do this?*** What's the purpose of giving a *small standard deviation* of the ***weights*** for the *actor* and the *large* *standard deviation* of the ***weights*** for the *critic*?
* By giving *a small variance* to the *actor* in a *large variance* to *critic*, it allows to manage to deal ***exploration*** vs ***exploitation***.
* Actor weights: Now we set the weights of the neural network of the actor. To access the weights of the actor we'll use self.actor\_linear.weight.data. Where actor\_linear is the neural network of our actor.
* We're going to use our function: **normalized\_columns\_initializer**. This function takes two arguments:
* The weights we want to initialize (i.e. self.actor\_linear.weight.data)
* The second argument is the standard deviation that we want these weights to have. We want a small standard deviation for the actor, it's going to be 0.01.

self.actor\_linear.weight.data = **normalized\_columns\_initializer**(self.actor\_linear.weight.data, 0.01)

* Actor bias: Now let's take care of the bias of the neural network of the actor. As above, we'll use self.actor\_linear.bias.data to access the bias, then we're simply going to use **fill\_**(0), because we want all the bias to be initialized with zero.

# *setting the standard deviation of the actor tensor of weights to 0.01*

self.actor\_linear.weight.data = **normalized\_columns\_initializer**(self.actor\_linear.weight.data, 0.01)

self**.actor\_linear.bias.data.fill\_**(0) # *initializing the actor bias with zeros*

* However, this line is not necessary because our bias are already initialized to zero with this **fill\_**(0) function inside the weights\_init function.
* We're doing this just to make sure that the bias are actually initialized to zero. Now we are 100% sure.
* Critic weight & bias: We're going to do the same for the critic. The only thing that we have to change is the standard deviation for the weights of the neural network for the critic.
* This time we set a large standard deviation, so we set 1.0 instead of 0.01.

# *setting the standard deviation of the critic tensor of weights to 0.01*

self.critic\_linear.weight.data = **normalized\_columns\_initializer**(self.critic\_linear.weight.data, 1.0)

self**.critic\_linear.bias.data.fill\_**(0) # *initializing the critic bias with zeros*

* We have a small standard deviation 0.01 for the actor and a large standard deviation 1.0 for the critic.
* ***Initialize the bias of the LSTM:*** We use self**.lstm** because lstm belongs to our object. There are two types of bias in the lstm, bias\_ih and bias\_hh, both are going to be initialized to zero. So first we access to the data and then we use **fill\_**(0) to fill all these bias with zeroes.

self**.lstm.bias\_ih.data.fill\_**(0) # *initializing the lstm bias with zeros*

self**.lstm.bias\_hh.data.fill\_**(0) # *initializing the lstm bias with zeros*

* ***Put the module in train-mode:*** We'll use a method that is inherited from the **nn.module**, the train() method. This method puts the module in *train-mode*.
* It allows to *activate* ***drop-out*** (if there is any) in the *batch normalization*. To use it we just add self.train().

self**.train**() # *setting the module in "train" mode to activate the dropouts and batchnorms*

Now our \_\_init\_\_() function is complete. We have our *convolutions* and *LSTM*, we have our two separate neural networks for the critic and the actor and all the *weights* and *bias* are *well-initialized*.

**\_\_init\_\_()**

**class** ActorCritic(torch.nn.Module):

**def** **\_\_init\_\_**(self, num\_inputs, action\_space):

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

        self.conv1 = **nn.Conv2d**(num\_inputs, 32, 3, stride=2, padding=1) # *first convolution*

        self.conv2 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *second convolution*

        self.conv3 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *third convolution*

        self.conv4 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *fourth convolution*

       # *making an LSTM to learn the temporal properties of the input –*

# *we obtain a big encoded vector S of size 256 that encodes an event of the game*

self.lstm = **nn.LSTMCell**(32 \* 3 \* 3, 256)

        num\_outputs = action\_space.n # *getting the number of possible actions*

        self.critic\_linear = **nn.Linear**(256, 1) # *full connection of the critic: output = V(S)*

        self.actor\_linear = **nn.Linear**(256, num\_outputs) # *full connection of the actor: output = Q(S,A)*

        self**.apply**(weights\_init) # *initilizing the weights of the model with random weights*

# *setting the standard deviation of the actor tensor of weights to 0.01*

        self.actor\_linear.weight.data = **normalized\_columns\_initializer**(self.actor\_linear.weight.data, 0.01)

        self**.actor\_linear.bias.data.fill\_**(0) # *initializing the actor bias with zeros*

# *setting the standard deviation of the critic tensor of weights to 0.01*

        self.critic\_linear.weight.data = **normalized\_columns\_initializer**(self.critic\_linear.weight.data, 1.0)

        self**.critic\_linear.bias.data.fill\_**(0) # *initializing the critic bias with zeros*

        self**.lstm.bias\_ih.data.fill\_**(0) # *initializing the lstm bias with zeros*

        self**.lstm.bias\_hh.data.fill\_**(0) # *initializing the lstm bias with zeros*

        self**.train**() # *setting the module in "train" mode to activate the dropouts and batchnorms*

The next step is to make the forward() function that will of course *forward propagate* the *signal* from the very beginning (with the original input images) throughout all the brain until we get the output.

**4.6: ActorCritic class forward()**

Now we're going to make the **forward()** function that will *forward propagate* the *signal* through all the brain (CNN & LSTM) from the very beginning with the input images up to the outputs (*Q-values* for the actor and the *V-value* i.e. value taken by the *V-function* for the critic).

* It'll be quite similar to DOOM (previous project), but this time we'll apply LSTM. So we have to do something more to propagate the signal.
* There is another change compared to before is that we're not going to use a **ReLU** *activation function* (as nonlinear activation function) instead we're going to use the ELU which is kind of a more *sophisticated* ***ReLU*** function (more in Pytorch documentation).
* forward: we call this function forward(), this function is going to take 2 arguments:
* self: the object itself
* inputs: So what will these inputs be? This will not only be the *input images*, these inputs will also contain the *hidden nodes* and the *cell nodes* of the ***LSTM***.
* In the forward function, basically we're considering the *hidden-nodes* and *cell-nodes* of the LSTM. So we're going to separate these two (hidden-nodes and cell-nodes) from the argument inputs of the forward() function.
* How can we separate them?
* We use the new variable inputs (which will be the inputs images), and we separate them with the tuple (hx, cx) is the tuple of the hidden nodes (hx) and the cell nodes (cx) of the LSTM (nodes of LSTM sometimes called states).
* We then assign those to the inputs (the argument of the forward function).

# *getting separately the input images to the tuple (hidden states, cell states)*

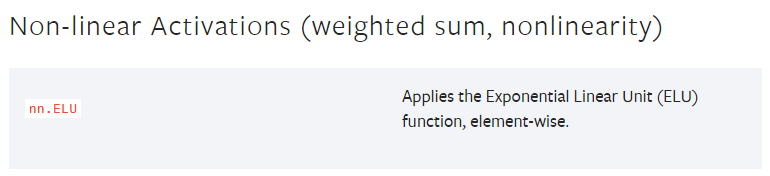
inputs, (hx, cx) = inputs

* ***Propagate to first layer:*** After that separation we now *propagate* the *signal* throughout the brain. To do that, we are going to get successively different layers from the first one to the last one, i.e. the *convolutions*, the *LSTM*, and the *linear connections* (the *full connections*).
* First we create a variable x, then we update it by our first-layer. To get this first layer we need to *propagate* the signal from the *inputs* to this *first layer*, we'll use first-convolution because the first convolution propagates the signal from the input images to the first layer.

# *forward propagating the signal from the input images to the 1st convolutional layer*

x = **F.elu**(self**.conv1**(inputs))

* We apply this first convolution to our input images inputs as self**.conv1**(inputs). It *propagates the signal* from the input images to the first layer.
* But also we have to use a *nonlinear activation function* to break the linearity in order to learn the *non-linear relationships inside images*. This time we are going to use elu activation function instead of **relu** (from the torch.nn.functional i.e. F).
* We apply elu over self**.conv1**(inputs) because we want to nonlinearly activate the neurons of this first layer (that we obtained by applying the first convolution on the inputs).
* elu in PyTorch documentation: (go to <https://pytorch.org/docs/stable/nn.html>). Then we search for "Non-linear Activations".



|  |  |
| --- | --- |
| * ELU means- Exponential linear unit | E:\1_Development_2.0\ML_phase_5_Ai_RL_note\cs_MATH\ELU.png |

|  |  |
| --- | --- |
| * **ELU vs ReLU:** The classical activation we've used is **ReLU**. It's just he max of 0 and x. It's equation is:      * So we can describe the ELU as (): * Notice from the **plots**, we'll get –ve output if the input is –ve. But for the **RelU**, the **output** is always non –ve. | E:\1_Development_2.0\ML_phase_5_Ai_RL_note\cs_MATH\ReLU.png |

* ***Propagate to second layer:*** We applied the elu on the *first convolutional layer*. Now x is the *first convolutional layer*. And by propagating the signal from the *first convolutional layer* to the *next layer*, x will become the next convolutional layer.
* To do the next forward ***propagation*** of the ***signal***, from the *first convolutional* *layer* to the *second convolutional* *layer*, we are going to update x again.

# *forward propagating the signal from the 1st convolutional layer to the 2nd convolutional layer*

x = **F.elu**(self**.conv2**(x))

* Notice, **conv1** was applied to inputs while the second convolution **conv2** is applied to x (i.e. the first convolutional layer), now x will be updated.
* ***Propagate to third layer:*** Now we get our *second convolutional layer* (updated x). Let's propagate the signal again from the second convolutional layer to the third one.

# *forward propagating the signal from the 2nd convolutional layer to the 3rd convolutional layer*

x = **F.elu**(self**.conv3**(x))

* ***Propagate to fourth layer:*** Similarly to propagate the signal from the third convolutional layer to the fourth one we can do the same

# *forward propagating the signal from the 3rd convolutional layer to the 4th convolutional layer*

x = **F.elu**(self**.conv4**(x))

* Recap:
* We start with input image, we then apply the *first convolution* to get the *first convolutional layer*.
* Then *second convolution to first convolutional layer* to obtain the *second convolutional layer*.
* Then *third convolution to the second convolutional layer* to obtain the *third convolutional layer*.
* Finally we apply the *fourth convolution to the third convolutional layer* to obtain the *fourth convolutional layer*.

And that's how the signal is propagated throughout the eyes of the AI.

* ***Flattening:*** Now we have the output signal after the four convolutions. We need to expand this whole output signal in *one dimensional vector* and that's the ***flattening step***.

# *flattening the last convolutional layer into this 1D vector x*

x = **x.view**(-1, 32 \* 3 \* 3)

* So we're going to update x again, it will become this flattened one dimensional vector.
* To do this we'll take x (fourth convolutional layer), we use view() function and in the first argument we put minus one (-1) because we want *one dimensional vector*.
* As a second argument we need to input the *number of elements* in this vector and it is 32\*3\*3.
* Why is the output of the four convolutions have the size 32\*3\*3 (we've discussed it previously in definition of memory of the LSTM in section 4.4)?
* It's actually not a simple formula but there is a *formula to compute this number of output neurons*, after flattening the pooled and convoluted images of the convolutions.

Now we have our flattened vector and the flattening step is done.

* ***LSTM propagation:*** LSTM takes the flattened vector (updated x, the one dimensional vector of 32\*3\*3 elements) as an input.
* But also we need to put the tuple of the hidden states (hx) and the cell States (cx). We can use hx and cx here because we've made that *separations* from the original inputs arguments of our forward() function. So we do that as:

self**.lstm**(x, (hx, cx))

* And self.**lstm**(x, (hx, cx)) will actually return a tuple of two outputs. Which will be the *output hidden-nodes* and the *output cell-nodes*.
* Therefore we're going to update *hidden-nodes* hx and *cell-nodes* cx because the LSTM will output these as a tuple.

# *the LSTM takes as input x and the old hidden & cell states and ouputs the new hidden & cell states*

hx, cx = self**.lstm**(x, (hx, cx))

* ***Getting the hidden states:*** Now we have the outputs of the LSTM, but we need only the ***hidden nodes***. So we need to update the x again with hx, which is the first element of the *output tuple* of the LSTM.

# *getting the useful output, which are the hidden states (principle of the LSTM)*

x = hx

* ***Returning the values:*** Since we have two brains: one for the actor and anotherfor the critic, therefore we have *return two output signals*.
* To return those, we simply need to take our *linear-full-connections* but *separately*, (i.e. a linear-full-connection of the *critic* and another linear-full-connection of the *actor*). We'll apply each of these full-connections to the output x (i.e. the hidden nodes of the LSTM).
* Linear-full-connection of the critic: self.**critic\_linear**(x)
* Linear-full-connection of the actor: self.**actor\_linear**(x)
* We're also going to return the tuple (hx, cx) of hx the *hidden node* and cx the *cell node* because we will be using them later in the retro loop of the LSTM.

# *returning the output of the critic (V(S)), the output of the actor (Q(S,A)), and*

# *the new hidden & cell states ((hx, cx))*

**return** self**.critic\_linear**(x), self**.actor\_linear**(x), (hx, cx)

* Now we're done with the brains for the actor and critic. And the good news is our *model.py* file is ready.
* In the next section we'll take care of the optimizer (i.e. **my\_optim.py** file) because we're going to make a separate optimizer. We're *not going to discuss each line of code* because a lot of it comes from the *research papers* and if we go into the deep details of what's going on with this optimizer this might be a little too overwhelming.
* We'll try our best to explain the code and so that we can understand the whole idea behind this optimizer.
* But we'll give more focus to make the **train()** because that contains the *algorithm of the A3C*.

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

        inputs, (hx, cx) = inputs # *getting separately the input images to the tuple (hidden states, cell states)*

        # *forward propagating the signal from the input images to the 1st convolutional layer*

        x = **F.elu**(self**.conv1**(inputs))

        # *forward propagating the signal from the 1st convolutional layer to the 2nd convolutional layer*

        x = **F.elu**(self**.conv2**(x))

        # *forward propagating the signal from the 2nd convolutional layer to the 3rd convolutional layer*

        x = **F.elu**(self**.conv3**(x))

        # *forward propagating the signal from the 3rd convolutional layer to the 4th convolutional layer*

        x = **F.elu**(self**.conv4**(x))

        # *flattening the last convolutional layer into this 1D vector x*

        x = **x.view**(-1, 32 \* 3 \* 3)

        # *the LSTM takes as input x and the old hidden & cell states and ouputs the new hidden & cell states*

        hx, cx = self**.lstm**(x, (hx, cx))

        # *getting the useful output, which are the hidden states (principle of the LSTM)*

        x = hx

        # *returning the output of the critic (V(S)), the output of the actor (Q(S,A)), and the new hidden & cell states ((hx, cx))*

**return** self**.critic\_linear**(x), self**.actor\_linear**(x), (hx, cx)

**All code at once (A3C brain)**

# *Making the A3C brain*

**class** ActorCritic(torch.nn.Module):

**def** **\_\_init\_\_**(self, num\_inputs, action\_space):

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

        self.conv1 = **nn.Conv2d**(num\_inputs, 32, 3, stride=2, padding=1) # *first convolution*

        self.conv2 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *second convolution*

        self.conv3 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *third convolution*

        self.conv4 = **nn.Conv2d**(32, 32, 3, stride=2, padding=1) # *fourth convolution*

        self.lstm = **nn.LSTMCell**(32 \* 3 \* 3, 256) # *making an LSTM (Long Short Term Memory) to learn the temporal properties of the input - we obtain a big encoded vector S of size 256 that encodes an event of the game*

        num\_outputs = action\_space.n # *getting the number of possible actions*

        self.critic\_linear = **nn.Linear**(256, 1) # *full connection of the critic: output = V(S)*

        self.actor\_linear = **nn.Linear**(256, num\_outputs) # *full connection of the actor: output = Q(S,A)*

        self**.apply**(weights\_init) # *initilizing the weights of the model with random weights*

        self.actor\_linear.weight.data = **normalized\_columns\_initializer**(self.actor\_linear.weight.data, 0.01) # *setting the standard deviation of the actor tensor of weights to 0.01*

        self**.actor\_linear.bias.data.fill\_**(0) # *initializing the actor bias with zeros*

        self.critic\_linear.weight.data = **normalized\_columns\_initializer**(self.critic\_linear.weight.data, 1.0) # *setting the standard deviation of the critic tensor of weights to 0.01*

        self**.critic\_linear.bias.data.fill\_**(0) # *initializing the critic bias with zeros*

        self**.lstm.bias\_ih.data.fill\_**(0) # *initializing the lstm bias with zeros*

        self**.lstm.bias\_hh.data.fill\_**(0) # *initializing the lstm bias with zeros*

        self**.train**() # *setting the module in "train" mode to activate the dropouts and batchnorms*

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

        inputs, (hx, cx) = inputs # *getting separately the input images to the tuple (hidden states, cell states)*

        x = **F.elu**(self**.conv1**(inputs)) # *forward propagating the signal from the input images to the 1st convolutional layer*

        x = **F.elu**(self**.conv2**(x)) # *forward propagating the signal from the 1st convolutional layer to the 2nd convolutional layer*

        x = **F.elu**(self**.conv3**(x)) # *forward propagating the signal from the 2nd convolutional layer to the 3rd convolutional layer*

        x = **F.elu**(self**.conv4**(x)) # *forward propagating the signal from the 3rd convolutional layer to the 4th convolutional layer*

        x = **x.view**(-1, 32 \* 3 \* 3) # *flattening the last convolutional layer into this 1D vector x*

        hx, cx = self**.lstm**(x, (hx, cx)) # *the LSTM takes as input x and the old hidden & cell states and ouputs the new hidden & cell states*

        x = hx # *getting the useful output, which are the hidden states (principle of the LSTM)*

**return** self**.critic\_linear**(x), self**.actor\_linear**(x), (hx, cx) # *returning the output of the critic (V(S)), the output of the actor (Q(S,A)), and the new hidden & cell states ((hx, cx))*

**4.7: code explanation: SharedAdam Optimizer**

In the previous section we've made the brains for the A3C, now we need to train this brain. In order to train this brains we need an optimizer. We'll use this optimizer in *stochastic gradient descent* to update the weights according to how much they contribute to the *error between the predictions and the targets*.

* Previously we've used the Adam-optimizer from *PyTorch* (**torch.optim.Adam**) in our training of Self-driving car and Doom-Playing-AI.
* But we're now dealing with a very challenging problem to play the *BREAKOUT* and *A3C algorithm* by itself is not enough to solve this problem.
* We need some customized optimizers and a lot of different tricks to solve this problem.
* So that's why we have a separate *custom optimizer based on the ADAM optimizer*.
* ***Custom Optimizer:*** It is based on ADAM optimizer and is contained in the **SharedAdam** class.
* Why Shared ADAM? Because it is actually the *ADAM optimizer* but it will work on *shared States*. So we're going to explain how it works (without coding ourselves).
* The class **SharedAdam** contains three functions:
* \_\_init\_\_()
* share\_memory()
* step()

**class** SharedAdam(optim.Adam): # *object that inherits from optim.Adam*

* The **SharedAdam** class is inherited from **optim.Adam** (which is of course the ADAM-optimizer, from the **optim** module from the **torch** library **torch.optim.Adam**). So we get all the tools related to the ADAM optimizer.
* \_\_init\_\_(): First we use the super() to inherit all the tools and all the basic parameters from the optim.Adam class and these basic parameters are: params, lr (learning rate), betas, eps (epsilon), weight\_decay. Notice we've used some default values because super() need them.

**def** **\_\_init\_\_**(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight\_decay=0):

# *inheriting from the tools of optim.Adam*

**super**(SharedAdam, self).**\_\_init\_\_**(params, lr, betas, eps, weight\_decay)

* Next, we start a FOR-loop, first one is "**for** group **in** self.param\_groups".
* "**self.param\_groups**" contains all the *attributes* of the *optimizer*. And among these attributes we have the *parameters* that we have to *optimize*.
* These parameters that we want to optimize are the *weights of the network* that are contained in **params** of **self.param\_groups**.
* So basically we go through **self.param\_groups** that contains all the parameters and
* For each group of parameters in **self.param\_groups** we are gonna go through the *parameters* that we want to *optimize*.
* Therefore, in the nested-FOR-loop, "**for** p **in** group['params']" means *for each Tensor of weights* that we want to *optimize*.

# *for each tensor p of weights to optimize*

**for** p **in** group['params']:

* What happens inside this loop with these following four lines of codes:

# *at the beginning, self.state is an empty dictionary so*

# *state = {} and*

# *self.state = {p:{}} = {p: state}*

state = self.state[p]

state['step'] = **torch.zeros**(1) # *counting the steps: state = {'step' : tensor([0])}*

# *update of the adam optimizer is based on an exponential moving average of the gradient (moment 1)*

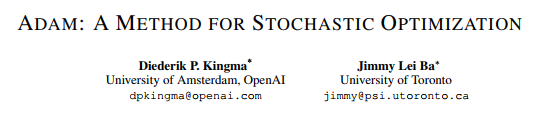
state['exp\_avg'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

# *update of the adam optimizer is also based on an exponential moving average of*

# *the squared of the gradient (moment 2)*

state['exp\_avg\_sq'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

* Basically there is an update made by the ADAM-optimizer based on an *exponential moving average* of the gradient. Here 'exp\_avg' is the *exponential moving average* of the *gradient of moment one* (i.e. of order 1).
* But the update made by ADAM is not only based on that, it is also based on an *exponential moving average* of the *squared of the gradient* (i.e. an exponential moving average of the gradient of moment 2 or order two).
* So here's happening the *exponential moving average of order-one* and also the *exponential moving average of order-two* i.e for each of them there's an EMA of the gradient.



|  |  |
| --- | --- |
| * ***Paper:*** If you want to get more in depth of how the exponential moving average works well, you can have a look at this research paper: *Diederik P. Kingma, Jimmy Lei Ba*, "Adam: A method for stochastic optimization". * Because basically the ADAM-optimizer that we're implementing right now is based on the following algorithm 1. * So if you want to have more details on how the algorithm works, this paper will be definitely helpful. Also you'll have some further explanations on the algorithm with the *ADAM's update rules*. So you can read this before we start the training phase. |  |

**class** SharedAdam(optim.Adam): # *object that inherits from optim.Adam*

**def** **\_\_init\_\_**(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight\_decay=0):

**super**(SharedAdam, self).**\_\_init\_\_**(params, lr, betas, eps, weight\_decay) # *inheriting from the tools of optim.Adam*

        # *self.param\_groups contains all the attributes of the optimizer, including*

        # *the parameters to optimize (the weights of the network) contained in self.param\_groups['params']*

**for** group **in** self.param\_groups:

**for** p **in** group['params']: # *for each tensor p of weights to optimize*

                state = self.state[p] # *at the beginning, self.state is an empty dictionary so state = {} and self.state = {p:{}} = {p: state}*

                state['step'] = **torch.zeros**(1) # *counting the steps: state = {'step' : tensor([0])}*

                # *the update of the adam optimizer is based on an exponential moving average of the gradient (moment 1)*

                state['exp\_avg'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

                # *the update of the adam optimizer is also based on an exponential moving average of the squared of the gradient (moment 2)*

                state['exp\_avg\_sq'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

* share\_memory(): The second function is for shared memory **share\_memory()**. The idea of this shared memory function is kind of like tensor.cuda() (CUDA is an accelerator based on GPU).
* And so basically what happens here is that we have these tensor.share\_memory\_(), (where the tensors are state['step'], state['exp\_avg'], state['exp\_avg\_sq'] ). These tensor.share\_memory\_() behave a little bit like tensor.cuda() (to make accelerated computations).

# *Sharing the memory*

**def** **share\_memory**(self):

**for** group **in** self.param\_groups:

**for** p **in** group['params']:

            state = self.state[p]

            state['step'].**share\_memory\_**() # *tensor.share\_memory\_() acts a little bit like tensor.cuda()*

            state['exp\_avg'].**share\_memory\_**() # *tensor.share\_memory\_() acts a little bit like tensor.cuda()*

            state['exp\_avg\_sq'].**share\_memory\_**() # *tensor.share\_memory\_() acts a little bit like tensor.cuda()*

* But the difference is that here the tensor.share\_memory\_() send the computation to a part of the GPU or CPU that is accessible to all the parallelized thread.
* So it's a little bit like tensor.cuda() but it's only sent to a part of the GPU or CPU accessible to the parallelized threads.
* step(): It's like the step() method of the ADAM optimizer that we already used in previous project. This is based on the *Algorithm 1* in *Diederik P. Kingma, Jimmy Lei Ba, "Adam: A method for stochastic optimization"* (<https://arxiv.org/pdf/1412.6980.pdf>).
* And besides what is done here is not totally compulsory because this is actually a *copy-paste* of the step() method of optim.adam class.
* So basically what is done here we could have done it by using our inheritance because we've already inherited from optim.adam. So to use our inheritance we could done

super(SharedAdam, self.step())

* Here self.step() is the step() method of optim.adam class. That's exactly the same. So we can replace all of following code using the above line of code.

# *Performing a single optimization step of the Adam algorithm*

# *(algorithm 1 in https://arxiv.org/pdf/1412.6980.pdf)*

**def** **step**(self):

    loss = **None**

**for** group **in** self.param\_groups:

**for** p **in** group['params']:

**if** p.grad **is** **None**:

**continue**

            grad = p.grad.data

            state = self.state[p]

            exp\_avg, exp\_avg\_sq = state['exp\_avg'], state['exp\_avg\_sq']

            beta1, beta2 = group['betas']

            state['step'] += 1

**if** group['weight\_decay'] **!=** 0:

                grad = **grad.add**(group['weight\_decay'], p.data)

**exp\_avg.mul\_**(beta1).**add\_**(1 - beta1, grad)

**exp\_avg\_sq.mul\_**(beta2).**addcmul\_**(1 - beta2, grad, grad)

            denom = **exp\_avg\_sq.sqrt**().**add\_**(group['eps'])

            bias\_correction1 = 1 - beta1 \*\* state['step'][0]

            bias\_correction2 = 1 - beta2 \*\* state['step'][0]

            step\_size = group['lr'] \* **math.sqrt**(bias\_correction2) / bias\_correction1

**p.data.addcdiv\_**(-step\_size, exp\_avg, denom)

**return** loss

In the next section we're going to move on to the train.py file and that will basically train our brains because we have our optimizer now.

* If you want to understand what happens behind you can look at this research paper Diederik P. Kingma, Jimmy Lei Ba, "Adam: A method for stochastic optimization" (<https://arxiv.org/pdf/1412.6980.pdf>).

**All code at once with comments (Shared ADAM optimizer)**

# *Optimizer*

**import** math

**import** torch

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

# *Implementing the Adam optimizer with shared states*

**class** SharedAdam(optim.Adam): # *object that inherits from optim.Adam*

**def** **\_\_init\_\_**(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight\_decay=0):

**super**(SharedAdam, self).**\_\_init\_\_**(params, lr, betas, eps, weight\_decay) # *inheriting from the tools of optim.Adam*

        # *self.param\_groups contains all the attributes of the optimizer, including*

        # *the parameters to optimize (the weights of the network) contained in self.param\_groups['params']*

**for** group **in** self.param\_groups:

**for** p **in** group['params']: # *for each tensor p of weights to optimize*

                state = self.state[p] # *at the beginning, self.state is an empty dictionary so state = {} and self.state = {p:{}} = {p: state}*

                state['step'] = **torch.zeros**(1) # *counting the steps: state = {'step' : tensor([0])}*

                # *the update of the adam optimizer is based on an exponential moving average of the gradient (moment 1)*

                state['exp\_avg'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

                # *the update of the adam optimizer is also based on an exponential moving average of the squared of the gradient (moment 2)*

                state['exp\_avg\_sq'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

    # *Sharing the memory*

**def** **share\_memory**(self):

**for** group **in** self.param\_groups:

**for** p **in** group['params']:

                state = self.state[p]

                state['step'].**share\_memory\_**() # *tensor.share\_memory\_() acts a little bit like tensor.cuda()*

                state['exp\_avg'].**share\_memory\_**() # *tensor.share\_memory\_() acts a little bit like tensor.cuda()*

                state['exp\_avg\_sq'].**share\_memory\_**() # *tensor.share\_memory\_() acts a little bit like tensor.cuda()*

    # *Performing a single optimization step of the Adam algorithm (see algorithm 1 in https://arxiv.org/pdf/1412.6980.pdf)*

**def** **step**(self):

        loss = **None**

**for** group **in** self.param\_groups:

**for** p **in** group['params']:

**if** p.grad **is** **None**:

**continue**

                grad = p.grad.data

                state = self.state[p]

                exp\_avg, exp\_avg\_sq = state['exp\_avg'], state['exp\_avg\_sq']

                beta1, beta2 = group['betas']

                state['step'] += 1

**if** group['weight\_decay'] **!=** 0:

                    grad = **grad.add**(group['weight\_decay'], p.data)

**exp\_avg.mul\_**(beta1).**add\_**(1 - beta1, grad)

**exp\_avg\_sq.mul\_**(beta2).**addcmul\_**(1 - beta2, grad, grad)

                denom = **exp\_avg\_sq.sqrt**().**add\_**(group['eps'])

                bias\_correction1 = 1 - beta1 \*\* state['step'][0]

                bias\_correction2 = 1 - beta2 \*\* state['step'][0]

                step\_size = group['lr'] \* **math.sqrt**(bias\_correction2) / bias\_correction1

**p.data.addcdiv\_**(-step\_size, exp\_avg, denom)

**return** loss

All code at once no comments (Shared ADAM optimizer)

# *Optimizer*

**import** math

**import** torch

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

**class** SharedAdam(optim.Adam):

**def** **\_\_init\_\_**(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight\_decay=0):

**super**(SharedAdam, self).**\_\_init\_\_**(params, lr, betas, eps, weight\_decay)

**for** group **in** self.param\_groups:

**for** p **in** group['params']:

                state = self.state[p]

                state['step'] = **torch.zeros**(1)

                state['exp\_avg'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

                state['exp\_avg\_sq'] = **p.data.new**().**resize\_as\_**(p.data).**zero\_**()

**def** **share\_memory**(self):

**for** group **in** self.param\_groups:

**for** p **in** group['params']:

                state = self.state[p]

                state['step'].**share\_memory\_**()

                state['exp\_avg'].**share\_memory\_**()

                state['exp\_avg\_sq'].**share\_memory\_**()

**def** **step**(self):

        loss = **None**

**for** group **in** self.param\_groups:

**for** p **in** group['params']:

**if** p.grad **is** **None**:

**continue**

                grad = p.grad.data

                state = self.state[p]

                exp\_avg, exp\_avg\_sq = state['exp\_avg'], state['exp\_avg\_sq']

                beta1, beta2 = group['betas']

                state['step'] += 1

**if** group['weight\_decay'] **!=** 0:

                    grad = **grad.add**(group['weight\_decay'], p.data)

**exp\_avg.mul\_**(beta1).**add\_**(1 - beta1, grad)

**exp\_avg\_sq.mul\_**(beta2).**addcmul\_**(1 - beta2, grad, grad)

                denom = **exp\_avg\_sq.sqrt**().**add\_**(group['eps'])

                bias\_correction1 = 1 - beta1 \*\* state['step'][0]

                bias\_correction2 = 1 - beta2 \*\* state['step'][0]

                step\_size = group['lr'] \* **math.sqrt**(bias\_correction2) / bias\_correction1

**p.data.addcdiv\_**(-step\_size, exp\_avg, denom)

**return** loss