Chapter – 4 (implementation – Part 1 – Library & Environment)

**Artificial Intelligence**

**AI-app (CNN-A3C) "Breakdown" Game play** (part 1)

Project overview

libraries & packages

normalized\_columns\_initializer()

weights \_\_init()\_\_

**Getting Started**

**4.0: Project overview**

We are about to work on one of the *most powerful model* in ***Artificial Intelligence***, which is: A3C - Asynchronous Advantage Actor-Critic. Welcome to the State-of-the-Art in Machine Learning.

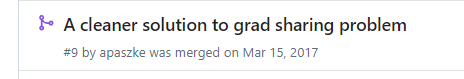
* ***A3C*** stands for *Asynchronous Advantage Actor-Critic*. It's a ***reinforcement learning*** algorithm used to train *deep neural networks* for decision-making in various environments. Here's a breakdown of the key components:
* Asynchronous: *Multiple actors* learn *simultaneously* and update a *shared model* ***asynchronously***, making it efficient for *parallel processing*.
* Advantage: The critic estimates the advantage, which is the ***difference*** *between the* ***actual reward*** *and the* ***expected reward***, guiding the *actor* towards *better actions*.
* Actor-Critic: This refers to the two components of the algorithm:
* Actor: Makes *decisions* (takes actions) in the environment.
* Critic: *Evaluates the actions* taken by the actor and provides feedback.

By combining these elements, A3C allows agents to learn effectively in complex environments with large action spaces.

* We are going to implement the most powerful version of this A3C algorithm (i.e. the most optimized implemented version)
* Consider A3C as the heart of the algorithm and then we'll use lot of tools to optimize the whole model, make the model super powerful.
* ***Why we do we need most optimized A3C model?***

Because solving BREAKOUT is more challenging than DOOM. One challenge is: we have to deal with a tiny ball in BreakOut (in DOOM the monsters are big and therefore easier to detect and kill/avoid).

* The version we're about to implement was first developed by ***Ilya Kostrikov*** and then corrected by the writer of the PyTorch, ***Adam*** ***Paszke***. And this A3C implementation is more perfect.
* About this **implementation**:
* It's originally developed by **Ilya Kostrikov** (<https://github.com/ikostrikov/pytorch-a3c>).



* Which originally didn't work well for breakout but then ***Adam*** ***Paszke*** made a pull request, for "A cleaner solution to grad sharing problem" (by apaszke).
* Thanks to Adam Paszke for fixing the gradient sharing problem. We can also see the contributors of this most powerful A3C implementation:

|  |  |
| --- | --- |
| * Many people tried different approach to make the AI to play the Breakout, for example: modify the breakout game by making the ball bigger (for better image processing) in kivy but the AI took too much time (compared to Paszke's implementation) to train on a powerful computer. | <https://github.com/ikostrikov2>  <https://github.com/ethancaballero>  <https://github.com/apaszke>  <https://github.com/beduffy>  <https://github.com/atgambardella>  <https://github.com/nadavbh12>  <https://github.com/lucasb-eyer> |

* So in this chapter, we'll implement the most updated version of A3C to play the breakout. We're basically going to reimplement all those codes, note that we'll try explain most of the code and focus on A3C.
* Folder Structure: Let's see which ones are *directly* ***related*** *to* ***A3C***, and which ones we're going to *implement*. There are actually two files:
* model.py :The first one is model.py, and we'll re-implement it line by line because that's where we make the A3C brains.
* The most important thing to understand here is that: we will have a shared model which will have the ***same updates*** *of the weights* for the ***actor*** and the ***critic***. That's a part of this special version of the A3C, the *shared model* with the *shared* *updates* of the weights
* train.py: we'll also implement it line by line because right after we made the brain of the A3C, we have to train them. So we'll implement the training mechanism in this train.ai file.
* It will contain the *heart of the* ***A3C*** *model* which will have ***2 losses*** to reduce
* Value loss: It is the loss related to the ***predictions*** *of the* ***critic***
* Policy loss: which is the loss related to the ***predictions*** *of the* ***actor***.
* Because A3C basically working with several agents each one having their *own copy of the environment* as well also have a *fully connected layer* that outputs a ***value*** *of the* ***V-function*** and that basically is a *common vision* of what's happening in the game.
* envs.py: It is an improvement of the gym environment with universe. envs.py allows us to have an *optimal pre-processing* of the *images* and also to ***normalize*** *all the values of the environment* such as: the *colors-intensities* or the *rewards-intensities*.
* main.py: This will execute the whole program. It'll *create the brain*, *train the brain* and *output the video*. Basically it'll run all the other codes.
* my\_optim.py: It's a special optimizer. It's basically the Adam-optimizer but adapted to the shared model that we're implementing.
* test.py: In this file we will implement a test agent. This test agent will play the breakout *without updating the model*. It's totally independent from the training.

**4.1: libraries & packages**

Let's start with model.py, here we'll implement the ***brain*** of the ***A3C model***. We'll make the *neural network* with *convolutional-layers* because our AI will have eyes. Also we'll integrate everything that is related to the A3C model.

* Moreover, our NN will contain RNN (more specifically LSTM) so that we can learn the temporal properties of what's going on in the environment (game).
* By learning the temporal properties of the input, the predictions can be better.
* So we are implementing a very powerful model that combines basically all the neural networks that we've learned so far in Deep-Learning (i.e. ANN, CNN & RNN).
* And at the heart of all these networks there is A3C model that will make the AI very powerful.
* Essential Libraries for Neural Networks: These libraries are crucial for building our AI's neural network:

# *Importing the librairies*

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

**import** torch

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

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

* numpy (np): Used for array manipulation.
* torch (torch): The core framework for deep learning.
* torch.nn: Provides building blocks for neural networks (e.g., convolutional layers).
* torch.nn.functional (F): Offers activation functions (e.g., ReLU) and other functionalities for neural networks (e.g., max pooling).

**4.2 normalized\_columns\_initializer()**

* We're going to make two functions for initialization. One will *initialize normalized columns* and other will *initialize the weights* for the Neural Network. These functions will help us to integrate the whole A3C model very easily. Once we're done with these two functions, we will start implementing the neural network.
* normalized\_columns\_initializer: This function will not only initialize some weights but sets a specific variance of tensor of weights.
* weights\_init: It will basically initialize the weights in an optimal way for the learning.
* normalized\_columns\_initializer: This function will take two inputs.
* First argument is the weights we want to initialize
* Second argument std is the standard deviation because we want to set a *specific variance* for our *tensor of weights*.
* ***Why we have to do this?***
* It's because when we make the neural network, there will be the actor and the critic. So we'll make *two* separate *fully connected layers*, one for the actor and another for the critic.
* These two fully connected layers will have weights and we will set a standard deviation for each of these two *groups of weights*.
* We will set a ***small*** *standard deviation* for the actor it'll be around 0.01 and a ***big*** *standard deviation* for the critic which will be around 1.0.
* That's why we're making this function so that we can set the standard deviation for the weights for the actor and the critic separately.

**def normalized\_columns\_initializer**(weights, std=1.0):

* However, we set the *default value* for the *standard deviation* std=1.0, we can change it afterwards when we initialize the weights.
* Output: We first prepare the output. That "out" is what will be returned by this function. Now we initialize this "out", it will be a *tensor of weights* that will have a *specific standard deviation*.
* But before we take care of *setting* the *standard deviation*, we need to *initialize* it and then we'll set the *standard deviation* using the parameter std.
* From torch library, we'll use randn(), which will initialize the torch-tensor with *random weights* from a ***normal*** *distribution*.
* We input the *number of elements* of our *tensor of weights*, i.e. the *total number of weights* that we're going to initialize. To get the total number of the weights, we'll use **size()**, because the parameter **weights** is a ***tensor***. We are actually *initializing* a *tensor of weights*.

out = **torch.randn(weights.size())**

* **weights.size**() will give us the *number of elements* in **weights** *tensor* so that it will create the *torch tensor* of the same number of elements of our weights and it will be initialized with *random weights* following a *normal distribution*.
* Now we set the standard deviation using the parameter **std=1.0**. We're going to do a *simple normalization*, i.e. we'll normalize the *torch-tensor* of the *weights*.
* We update *out* by multiplying it by *std/torch.sqrt(out.pow(2).sum(1).expand\_as(out))*.

out \*= std / torch.sqrt(out.pow(2).sum(1).expand\_as(out))

* *out.pow(2).sum(1).expand\_as(out)* is the *squared sum of the weights* of our vector.
* In *sum(1),* 1 is the index.
* To get all the weights separately, we use *expand\_as(out)*, it'll get the weights of *out* which was initialized as a torch-tensor of weights.
* *out.pow(2)* squares the weights.
* Then we take the square-root by using *torch.sqrt()*. To apply the normalization.
* Finally we divide std by that square-root. So that .
* This *std/torch.sqrt(out.pow(2).sum(1).expand\_as(out))* will make sure that this *tensor of weights* that we *initialized* will have a *variance* that will be equal to the *square of the standard deviation* *std* that we put as argument i.e. **std=1.0**.
* That's how we can set a *specific standard deviation* for our future actor and critic. We will choose a *small standard deviation* for the *actor* and a *larger* one for the *critic* and we will do this very easily using this **normalized\_columns\_initializer** function (that we're now implemented).
* Finally we return the output that is now normalized with the specific standard deviation std.

# *Initializing and setting the variance of a tensor of weights*

**def** **normalized\_columns\_initializer**(weights, std=1.0):

    out = **torch.randn**(**weights.size**())

    out \*= std / **torch.sqrt**(**out.pow**(2).**sum**(1).**expand\_as**(out))

    # *thanks to this initialization, we have var(out) = std^2*

**return** out

Next we'll make the weights\_init() function to *initialize* the *weights* to make the learning *optimal*.

**4.3: weights\_init()**

Now let's make this weights\_init() function to initialize the weights. It will take m as an argument; m represents the object of the neural network.

* ***Different kind of initialization:*** Inside this function, we'll *initialize* the *weights* of the neural network in such a way that we get an *optimal learning*.
* Note that the weights will be *initialized in a specific way* that we *haven't seen before*; it's based on ***research papers*** and ***experiments***. So this will not seem particularly intuitive.
* So we'll implement it without getting into the details of why we initialize the weights that way.
* Getting the class-name: It's a python trick to get the *class name* of the *object* m (parameter). That *class name* will be used later to make the distinction between the convolution and the full-connection. Since our AI will have eyes and therefore it will have some ***convolutional layers***. Also it'll have some ***fully-connected layers***.
* But we will have a *different initialization* of the *weights* for these two types of connections. So we're going to use this python-trick to separate these two kinds of connections.
* And then we'll use an if-condition to get the *different initialization* for each of these connections.
* To get the class name we'll apply m.\_\_class\_\_.\_\_name\_\_ to the *neural network object* m.

Initialization of the **convolution connections**

* Conditional initialization: The first case is: if the connection is a *convolution* (we check against "-1", it means "not found").
* So "**if** **classname.find**('Conv') **!=** -1" means if we have a *convolution connection*.
* Weight shape: weight\_shape will be a list that will basically contains the *shape of the weights* in our neural network m.

weight\_shape = **list(m.weight.data.size()**) # list containing the shape of the weights in "m"

* We're using the **list()** function to create a list. To get the shape of the m.weight, we need to use data.size() attribute. **m.weight.data.size()** will get us the shape of those *weights* in the *convolution connection*.
* Fan-in and Fan-out: To initialize the weights of this convolution connection, we're going to need two values.
* First is the product of the *first dimension* by the *second dimension* by the *third dimension*.
* Then we'll also get the *zero dimension* times the *second dimension* times the *third dimension*.

We'll use these two values in the computation of how we initialize the weights.

* Fan-in: We're we going to use the NumPy **prod()** function. We used the dimensions **1**, **2** and **3** of our weight-shape. We just need the 3-indexes of weight.shape(),

fan\_in = **np.prod**(weight\_shape[1:4]) # *dim1 \* dim2 \* dim3*

* Fan-out: We'll do the same for **fan\_out**, it's going to be the product of the dimensions **0**, **2** and **3**. So we use **prod()** for dimension **2** to **3**, and then manually multiply with dimension **0**.

fan\_out = **np.prod**(weight\_shape[2:4]) \* weight\_shape[0] # *dim0 \* dim2 \* dim3*

* Weight bound: Now we're going to use those two values fan\_in and fan\_out to proceed to the initialization. We're going to compute a new value: w\_bound, which will be equal to square-root of 6.0/(fan\_in + fan\_out).

w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

* **w\_bound** represents in some way the *size of the tensor of weights*.
* Why did we get this? Because we want to *generate* some *random weights* that are *inversely proportional* to the *size of the tensor of weights*.

**if** **classname.find**('Conv') **!=** -1: # *if the connection is a convolution*

        weight\_shape = **list**(**m.weight.data.size**()) # *list containing the shape of the weights in "m"*

        fan\_in = **np.prod**(weight\_shape[1:4]) # *dim1 \* dim2 \* dim3*

        fan\_out = **np.prod**(weight\_shape[2:4]) \* weight\_shape[0] # *dim0 \* dim2 \* dim3*

        w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

# *generating random weights of order inversely proportional to the size of the tensor of weights*

**m.weight.data.uniform\_**(-w\_bound, w\_bound)

**m.bias.data.fill\_**(0) # *initializing all the bias with zeros*

* Generating random weights: Because indeed what we're about to do
* Now we get the ***weight*** of our neural network ***m***, access its data (i.e. the tensor itself).
* Then from that *tensor of weights* we're going to generate some *random weights* that are *inversely proportional* to the *size* of the *tensor weights*.
* We'll use uniform\_() function to do that. We set **-w\_bound** and **w\_bound** to the *lower bound* and *upper bound* respectively.
* Initializing bias: We are going to initialize all of them with zeros using the **fill\_()** method. It'll fill the *tensor of bias* with zeros.

To summarize we *generate some* ***random weights*** inversely proportional to the *size of the tensor weights* and we *initialized the* ***bias*** with zeros.

Initialization of the **full connection**

* ***Conditional initialization & weight shape:*** We do the same for the full connections. First we check is the connection is a full connection (i.e. a linear connection in a classic artificial neural network). We check against "-1". Then we initialize the **weigt\_shape** as we did above (convolution connections):

**elif** **classname.find**('Linear') **!=** -1: # if the connection is a full connection

weight\_shape = **list**(**m.weight.data.size**()) # list containing the shape of the weights in "m"

* ***Fan-in and fan-out:*** We'll now calculate the fan\_in and fan\_out slightly differently for the full-connections. There is less dimension for a full connection than for a convolution.
* **fan\_in**: This time it won't be equal to the product of dim1 \* dim2 \* dim3, actually this time it will be equal to simply the dim1. That's because in the ***full-connection*** there is *less connections* than a *convolution connection*.

fan\_in = weight\_shape[1] # *dim1*

* **fan\_out**: Similarly for full-copnnection the fan\_out is dim0 of the weight\_shape.

fan\_out = weight\_shape[0] # *dim0*

* **w\_bound** , generating random weights and initializing the bias for *full-connection* are the same as we did for *convolution connection*.

w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

m.weight.data.uniform\_(-w\_bound, w\_bound)

**m.bias.data.fill\_**(0) # initializing all the bias with zeros

That is, as previously we use the uniform\_() function for the random weights and the fill\_() function for the bias to get the same kind of initialization, but this time with a different fan\_in and fan\_out giving different w\_bound and therefore we get *different weights* for full-connection.

So it's basically the same principle. The only thing that changes here is that we have *less dimensions* for the *full connection* and therefore more *simple computation* for w\_bound to generate these random weights.

**elif** **classname.find**('Linear') **!=** -1: # *if the connection is a full connection*

        weight\_shape = **list**(**m.weight.data.size**()) # *list containing the shape of the weights in "m"*

        fan\_in = weight\_shape[1] # *dim1*

        fan\_out = weight\_shape[0] # *dim0*

        w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

# *generating random weights of order inversely proportional to the size of the tensor of weights*

**m.weight.data.uniform\_**(-w\_bound, w\_bound)

**m.bias.data.fill\_**(0) # *initializing all the bias with zeros*

* Now our weights\_init(m) function is ready, So we have now 2 tools, normalized\_columns\_initializer() and weights\_init(). We are ready to start building the brain.

# *Initializing the weights of the neural network in an optimal way for the learning*

**def** **weights\_init**(m):

# *python trick to check class-name of object "m" (convolution or full connection)*

    classname = m.\_\_class\_\_.\_\_name\_\_

**if** **classname.find**('Conv') **!=** -1: # *if the connection is a convolution*

        weight\_shape = **list**(**m.weight.data.size**()) # *list containing the shape of the weights in "m"*

        fan\_in = **np.prod**(weight\_shape[1:4]) # *dim1 \* dim2 \* dim3*

        fan\_out = **np.prod**(weight\_shape[2:4]) \* weight\_shape[0] # *dim0 \* dim2 \* dim3*

        w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

# *generating random weights of order inversely proportional to the size of the tensor of weights*

**m.weight.data.uniform\_**(-w\_bound, w\_bound)

**m.bias.data.fill\_**(0) # *initializing all the bias with zeros*

**elif** **classname.find**('Linear') **!=** -1: # *if the connection is a full connection*

        weight\_shape = **list**(**m.weight.data.size**()) # *list containing the shape of the weights in "m"*

        fan\_in = weight\_shape[1] # *dim1*

        fan\_out = weight\_shape[0] # *dim0*

        w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

# *generating random weights of order inversely proportional to the size of the tensor of weights*

**m.weight.data.uniform\_**(-w\_bound, w\_bound)

**m.bias.data.fill\_**(0) # *initializing all the bias with zeros*

In Next steps, we'll start by making

* The *eyes* of our AI add an
* LSTM to learn the temporal properties of the input
* And then we'll take care of the actor and the critic. And that's where we'll use this 2 function normalized\_columns\_initializer() and weights\_init().

**All code at once**

# *AI for Breakout*

# *Importing the librairies*

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

**import** torch

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

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

# *Initializing and setting the variance of a tensor of weights*

**def** **normalized\_columns\_initializer**(weights, std=1.0):

    out = **torch.randn**(**weights.size**())

    out \*= std / **torch.sqrt**(**out.pow**(2).**sum**(1).**expand\_as**(out)) # *thanks to this initialization, we have var(out) = std^2*

**return** out

# *Initializing the weights of the neural network in an optimal way for the learning*

**def** **weights\_init**(m):

    classname = m.\_\_class\_\_.\_\_name\_\_ # *python trick that will look for the type of connection in the object "m" (convolution or full connection)*

**if** **classname.find**('Conv') **!=** -1: # *if the connection is a convolution*

        weight\_shape = **list**(**m.weight.data.size**()) # *list containing the shape of the weights in the object "m"*

        fan\_in = **np.prod**(weight\_shape[1:4]) # *dim1 \* dim2 \* dim3*

        fan\_out = **np.prod**(weight\_shape[2:4]) \* weight\_shape[0] # *dim0 \* dim2 \* dim3*

        w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

**m.weight.data.uniform\_**(-w\_bound, w\_bound) # *generating some random weights of order inversely proportional to the size of the tensor of weights*

**m.bias.data.fill\_**(0) # *initializing all the bias with zeros*

**elif** **classname.find**('Linear') **!=** -1: # *if the connection is a full connection*

        weight\_shape = **list**(**m.weight.data.size**()) # *list containing the shape of the weights in the object "m"*

        fan\_in = weight\_shape[1] # *dim1*

        fan\_out = weight\_shape[0] # *dim0*

        w\_bound = **np.sqrt**(6. / (fan\_in + fan\_out)) # *weight bound*

**m.weight.data.uniform\_**(-w\_bound, w\_bound) # *generating some random weights of order inversely proportional to the size of the tensor of weights*

**m.bias.data.fill\_**(0) # *initializing all the bias with zeros*