Chapter 4

**AI: A3C (Asynchronous Advantage Actor Critic)**

**"Breakdown" Game playing AI**

The three A's in A3C

Actor-Critic

Asynchronous

Advantage

LSTM Layer

**4.1 Objectives of A3C**

A3C stands for "Asynchronous Advantage Actor Critic".

1. Three A's of A3C:

* We'll discuss the abbreviation stands for and how to deal with the three A's.
* We'll look at the research paper that gave birth to A3C, the Google Deep Mind Research Paper.
* Why A3C is the cutting edge most advanced algorithm on the planet for AI.

1. Actor-Critic:

* It is about the first A of A3C which is Actor-Critic.

1. Asynchronous:

* Then we'll talk about the Asynchronous Element in the **A3C** algorithm, where we'll put everything together and all start making sense.

1. Advantage:

* In this step all pieces of A3C come together.

1. LSTM Layer:

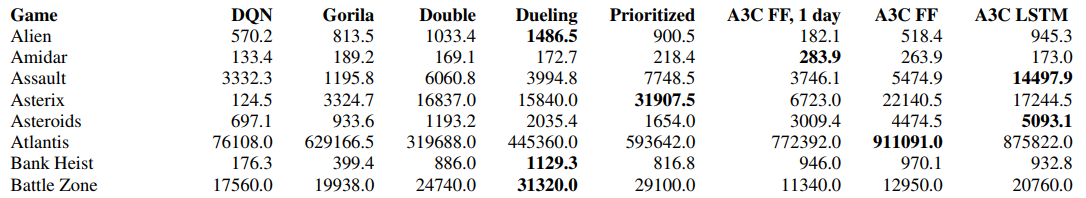
* We will add the long short term memory to our A3C algorithm. This will showcase how the A3C algorithm can be modified to suit certain purposes.

**4.2 The three A's in A3C**

* A3C stands for "Asynchronous Advantage Actor Critic". This is an algorithm which was developed at Google Deep Mind in ***2016*** by a group of researchers and it's the cutting edge algorithm for artificial intelligence.
* It has multiple modifications, but nevertheless this algorithm blows everything else including Deep Convolutional Q-Learning Networks.
* **A3C** is **faster**.
* It takes **less** **time** for **training** and
* Gets **better results**.

|  |  |
| --- | --- |
| * Reference: The paper that first introduced A3C is called the ***Asynchronous Methods Of Deep Reinforcement Learning*** by Volodymyr Mnih and others from *Google Deep Mind*. |  |

* In this research-paper you can see the even in Google Deep Mind they're training/evaluating their algorithms on games. DQN, Gorila and A3C FF or A3C LSTM are different algorithms and their performance is tested with different games.
* The bold results are best results and most of the best-results appeared in A3C-LSTM.



* Because Games are a Simulated Environment or a *small confined environment* with *certain rules* and we want to understand how well this artificial intelligence doing in those games.
* Lot of those games you can find on OpenAi GYM. In our case we'll work with the game called BREAKOUTS.

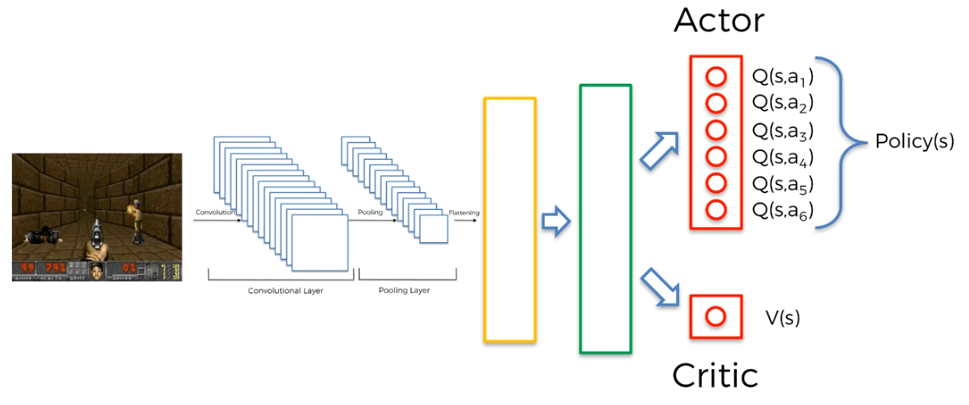
So you can see that A3C-LSTM is a really powerful algorithm. It is indeed at the forefront of artificial intelligence and that's exactly what we will be implementing. Later in the practical section we'll be working with their pseudo code and implement it.

**4.3 Actor-Critic**

In Deep-Convolutional Q-learning, our computer was seeing the actual image and pixels not just a vector. For simplicity's sake in our image, we represent the layers by Boxes and the networks by Arrows. It doesn't change the essence. This is just the representation. This is still Deep-Convolutional Q-learning.

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

* **Actor and Critic:** In previous Deep-Convolutional Q-learning we had our ***predicted Q-values*** at the end of the NN.
* Actor: In the " Actor-Critic model " we also have all the ***predicted Q-values*** and these are called the "**ACTOR**". And in addition we’ll have an extra value, which is called "**CRITIC**".
* Critic: In the " Actor-Critic model " **NN** will also output the "**State-value**" (recall from the previous chapters, we called it **V(s)**). This state-value is called "**CRITIC**".



* So the Neural Network is now predicting the *all possible* Q-values and **V(s)** the ***value*** of the ***state***.
* The top outputs are the Q-values for the actions. They're the same thing as we discussed in Deep-Convolutional Q-learning. This part is called "Actor", because that's the part where the agent chooses the action.
* The bottom part which is just one value and that is **V(s)** so that is the ***value*** of the ***state*** **s**. And this part is called "Critic".
* In some literature or blogs or some kind of discussions you might find that the **Q-values** are called **Policies(s)** (also denoted by **P(s)**) of state **s**.
* Remember that the policy is all the possible actions together and then the agent is deciding which action to take.

**4.4 Asynchronous**

Now we discuss about Asynchronous part of A3C algorithm. We're going to find out what ASYNCHRONOUS here stands for and what it means.

* First let's go back a step. Let's look at what we've started in this whole course for *Reinforcement Learning*, what it's all about.
* What we've done before: In reinforcement learning, our agent is in a certain state.
* They observe the state.
* They make certain decisions, they take actions in that state and
* Then the agent gets into a new state with a reward (for taking that action or it could be a penalty as well).
* And based on that new state, now they take another action again. They get a reward and end up in another new state and so on.

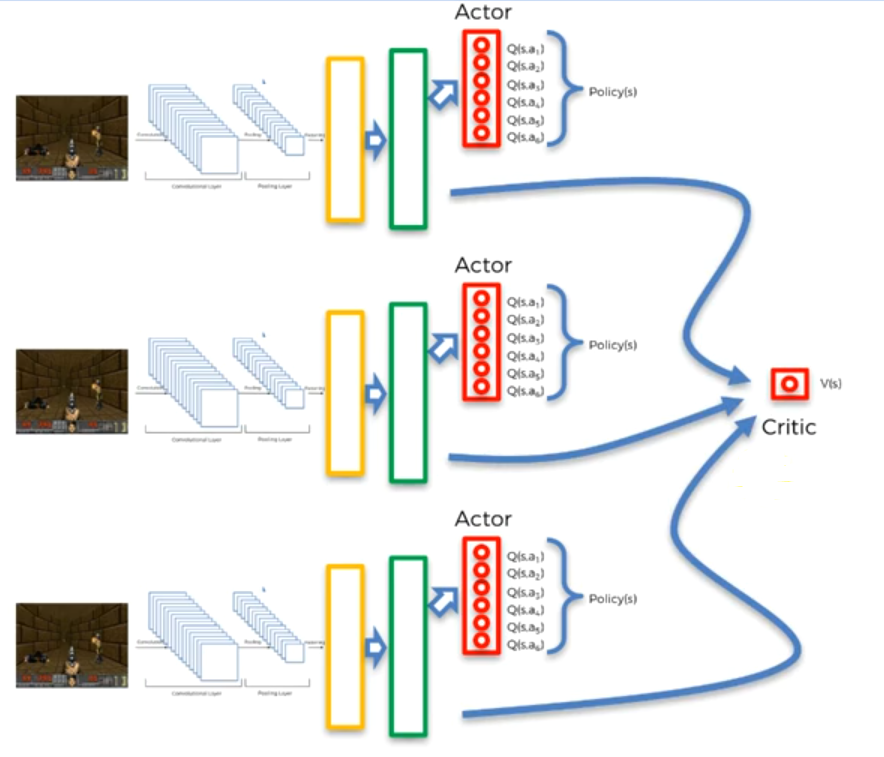
This idea we've been using in **Q-learning**, in **Deep Q-learning** and in **Deep Convolutional Q-learning** and that has allowed our agents to beat gradually more complex and more complex environments.

|  |  |
| --- | --- |
| * MULTI-AGENT concept: There is even better concept that takes above idea even further level. The idea is, ***instead*** of having ***one agent*** acting in the environment, we can have ***several agents*** acting in the ***same*** ***environment***. * Asynchronous: It's called the ASYNCHRONOUS because those agents were initialized differently. So their ***start*** ***inputs*** are different. |  |

* They will *go through environment* in different ways, *explore* in *different ways* and then in the next iterations they are also going to explore in different ways.
* Advantages:
* More Experience: So if we have three agents, we're getting triple the amount of experience to understand how to operate it in that environment.
* Avoid local Maxima: Each one of them getting experience and apart from having a broad range of experience it ***also reduces*** the chances of one agent getting stuck in a local maximum.
* The likelihood of several agents getting stuck in that same local maximum is decreases with the number of agents.
* The probability of one agent getting stuck in a certain local maximum might be high but the probability of all three agents getting stuck in that local maximum is much lower.
* Help each other: As long as they **share experience between each other** they can help each other. If one agent got stuck in a local maximum, from other agent's information it can find out the better options.
* **Sharing the experience:** We're going to replicate the actor-critic-model for each of our agents. Because now we have multiple agents.
* They are sharing their experience between each other.
* They're all independent and separately playing the same game, but their initials (starting values) are different.
* Indeed you'll get *more* *experience* and *more variety* especially if they're *initialized differently*.
* So it's not going to be identical training/learning from this game.

It's more like the agents are playing on three different computers. That’s how you're going to have a *bigger variety of possible solutions*.

|  |  |
| --- | --- |
| * **Why sharing experience is better:** If they're not sharing the experience among each other or not learning from each other, they don't have the advantage. Because, they can't leverage each other's strengths and can't mitigate each other's weaknesses. * So if you put these three agents playing on three computers side by side, indeed you'll have more experience, more possibility to get a better solution. * But it'll be even better if they start sharing that experience. * **Share the experience through "critic V(s)":** We have **two** kind of **V(s)**, One **V(s)** is output *from the Neural Network*, another **V(s)** is calculated from the *environment* (as we discussed in the Q-learning: chapter 1). * This critic **V(s)** is output *from the Neural Network* is actually like follows: |  |



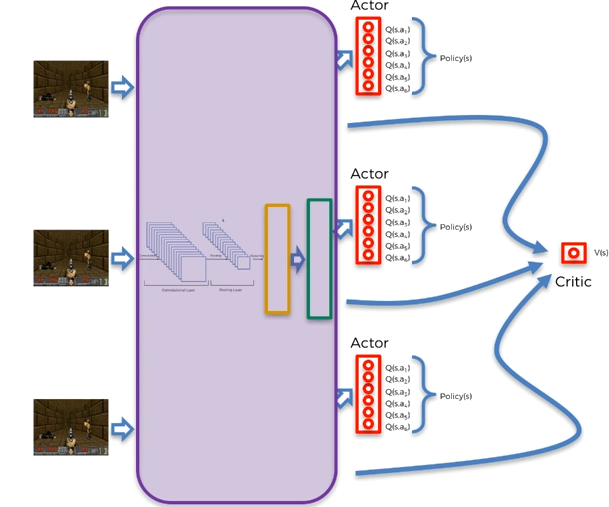
* All these agents are contributing to the same critic. They don't have separate critics they have a common critic.
* That's the key of how the Actor-Critic ties in with the Asynchronous.
* How do we calculate the V(s): As you remember from the Chapter 1 : Q-learning, we can get **V(s)** through the *rewards* that we get through the *environment*.
* As the agents explore their environment they predict the **V(s)** from the NN and
* They also have the **V(s)** that they can calculate (Expect through the rewards from the *environment* that they've *already* *explored*. However, as they explore more of course that value can change).
* As they're going through the learning process they're going to adjust the **V(s)** from their neural networks in order to better match that expected **V(s)** that *calculated from the environment*.
* So the Critic-Part is shared between the agents and that is how they share the information between each other. And that's how they are able to kind of see what is going on in the *environment shared with each other*.
* Then the agents use that information in the Advantage-Part in order to *optimize* how they're *behaving* *the* *environment*.

8:25

* Single NN A3C: *Above A3C-model* is the core of the A3C. However, there is a modified version of A3C created by the Creator of the Pytorch.
* This version of A3C gas *one NN for all agents*, i.e. all agents share the same NN instead of separate NN for each.
* We also apply this kind of A3C in our practical implementation part.

|  |  |
| --- | --- |
| Separate NN | Shared NN |
|  |  |

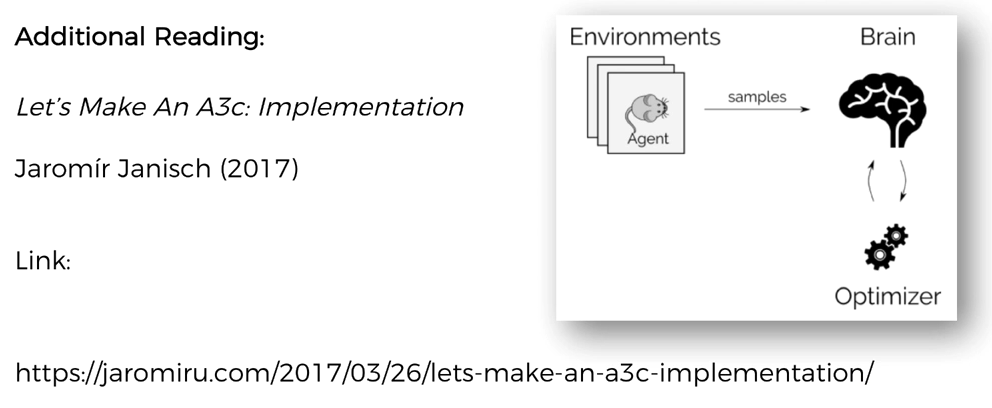
* So ultimately there's only one neural network here shared among the agents. They all have one neural network which is shared for the different actors and a common critic. This is basically the architecture that we're going to be using in the practical implementation.



* It would make more sense after "Advantage" in the following section. But generally speaking, there's only one network which each of the agents use or share.
* That means that ***they share*** the ***weights*** *of the network are* shared between agents and they update the whole network not just their own network.

The main take away from this Asynchronous section is that the **CRITIC** is shared. And that's how the *agents* are able to make sure that they are *cooperating together* in order to get the result much faster.

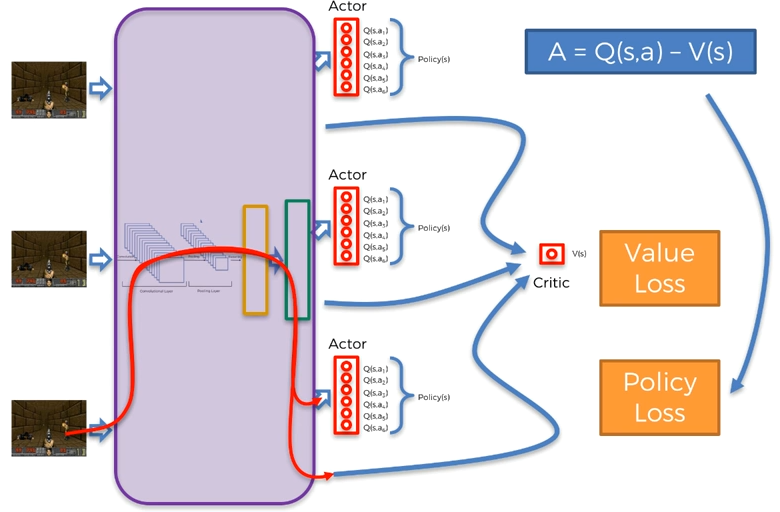
* Additional reading: So this is a blog by Jaromir Jansch. There are two parts of A3C implementation and theory. We'll try to implement it in our practical implementation part.



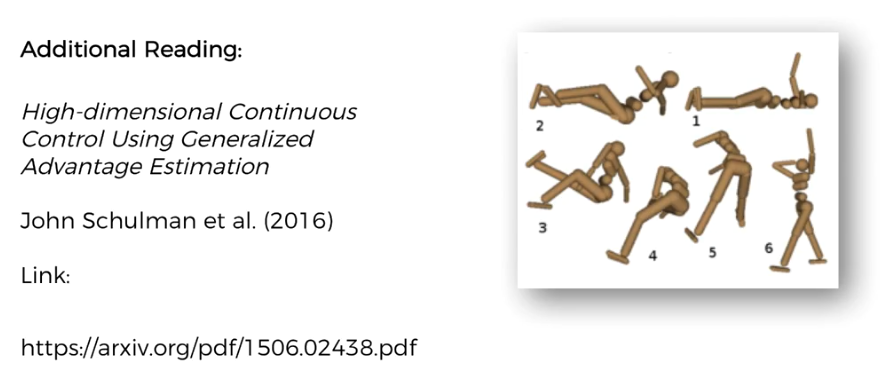
**4.5 Advantage**

Now let's discuss the "Advantage" part of the A3C-model. In the previous section we've got a neural network which is shared between the agents and we've got the *critic* which is also shared between agents.

* Let's consider the third agent of our example, the information goes the convolutional layer, pooling layer, flattening layer, hidden layers of the neural network and then as output we get ***policies*** and critic value ***V(s)***.



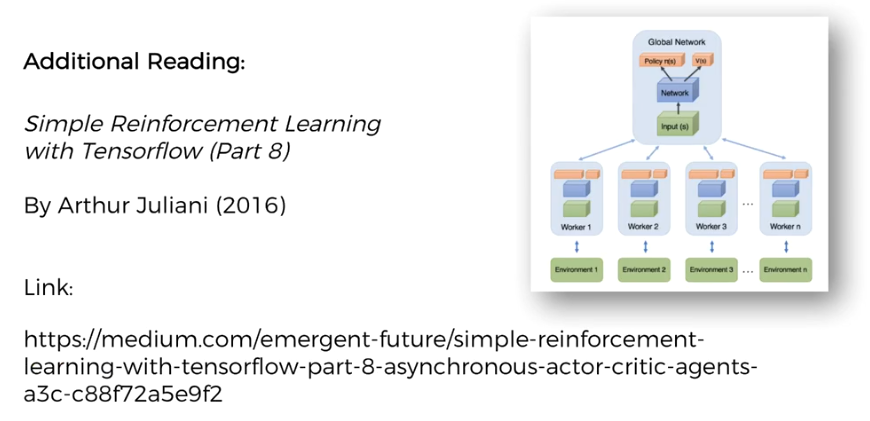
* Two kind of loss: The point is, when back-propagation happens we calculate two kind of loss to update the weights.
* **Value-Loss on V(s)** and
* **Policy-Loss**
* Value-Loss on V(s): We’ve discussed before in the previous chapters. It calculated from rewards, discount factor it's very similar what we've discussed in Deep-Q-Learning.
* Basically the network predicts a certain value **V** and at the same time we can estimate a value for **V** based on what we know about the environment so far. By comparing the two we can calculate the value-loss and then back propagator trough the network to update the weights.
* Policy-loss: This is the part where the critic is shared between the actors or the agents.
* So to understand Policy-loss we need to introduce a value called ***Advantage*** hence the name of this part. The ***Advantage*** is calculated as below
* It is the difference of the ***Q-value*** of the ***action*** that you chose to play in the state **s** that you are in and the value of that state **V(s)**.
* And this Advantage is used in the calculation of the Policy-loss.
* We don't go to the details of the calculation of Policy-loss because it's quite complex & it also uses entropy. We're going to discuss Policy-loss on an intuitive level. We'll try to find:
* Why are we doing Policy-loss
* Why are we calculating this ***Advantage*** and
* How is it going to help us.
* The Q value here comes from what the neural network predicted for this agent in this specific action and in this specific state for the action that it can play
* The agent can select one of the action from the policy (group of actions) and it can play it.
* Whereas the is the value called ***critic*** which shared by the *different agents*. This is how the ***critic*** comes into play.
* Because we've got *an action* that we choose to play for this agent in that state then the *critic* can tell us the *known value* of that state.
* It is the overall known value for the whole group of agents that are performing together.
* Because they're *sharing* the critic, they're all contributing to this to this values that are *being calculated for a different set*.
* That means, **critic** knows a **V(s)** value that how much better Q value **Q(s, a**) you're selecting compared to this known **V(s)** value.
* The agent selects a Q-value here based on a policy (Soft-max or Epsilon-Greedy or other) with exploration plus exploitation combined.
* The question is what is the **ADVANTAGE** that your selected action brings compared to the known value of that state? And that is the essence of the ***ADVANTAGE***.
* Basically ***advantage*** **A** is used to calculate the policy-loss and then the policy loss is then back propagated.
* Actually both VALUE-LOSS and POLICY-LOSS are back propagated through the NN and the weights are adjusted in order for the network to better represent the value of the critic V(s).
* And also when this POLICY-LOSS is back propagated through the NN, the weights are adjusted in such a way so that this ***advantage*** is maximized.
* That means when an agent takes a bad actions where the Q-value is ***less than*** the known value for the state.
* Then A3C algorithm detects that the newly taken action is worse than what we've already know about this whole environment.
* And therefore the weights are adjusted in a way so that this action happens rarer (i.e. a less frequent occurrence that we choose that bad action).
* On the other hand, if you choose a very good action where Q-value is ***greater than*** the known value V, then during this backwardation of the policy-loss, the weights are adjusted in such a way that the new taken action occurs more frequently.
* Because the ***advantage*** **A** was very high there.
* That's how the network is slowly going to construct itself into something that
* On one hand calculates the values correctly and
* On the other hand it encourages the actions which have a high advantage.
* The same thing happens to all other agents.
* Note that, the value of the state **V(s)** which comes from the shared critics is kind of observing all of these agents at the same time.
* All agents are contributing towards a critic **V(s)** to make sure that the critic is ***representative*** of what's going on in the ***actual*** ***environment***.
* The value-loss adjusts the weights in such a way that they reflect very well the actual situation of things in the environment.
* So that they can rely on this value **V(s)** and then use it in the calculation of the advantage **A**.
* All of these agents are contributing to this critic **V(s)**.
* But at same time the critic is observing the decisions or the policies of these agents. It criticizing their decisions through that advantage **A**.
* Basically this whole policy-loss helps the network adapt or morph in such a way that it'll take good actions more often and do less of the bad things.
* That's how these two losses, value-loss and policy-loss come into play and back propagated.
* Summery: That's how these asynchronous agents are playing or exploring the environment and all altogether and contributing to a ***critic*** which is observing their policies and the actors through the ***advantage*** and calculates value-loss and policy-loss which back propagate to adjust the network in order to improve the actors performances.
* Additional reading:
* The first one is "High-dimensional Continuous Control Using Generalized Advantage Estimation" by ***John Schulman et al. (2016)***.
* Here, you'll find all the different types of advantages, the *general advantage in estimation* with calculations.
* So if you want to find out more about advantage and exactly how it works, the formulas behind it then you must read this article.



* Another one is a series of blog post by Arthur Juliani (2016) which we've mentioned a couple times already. This is part-8 which is specifically about A3C.
* With a bit more mathematics you can pick up some additional things from here about what is going on. Keep in mind that this blog is in TensorFlow and we're using Pytorch.

Also we structured our approach is: Actor-Critic > Asynchronous > Advantage

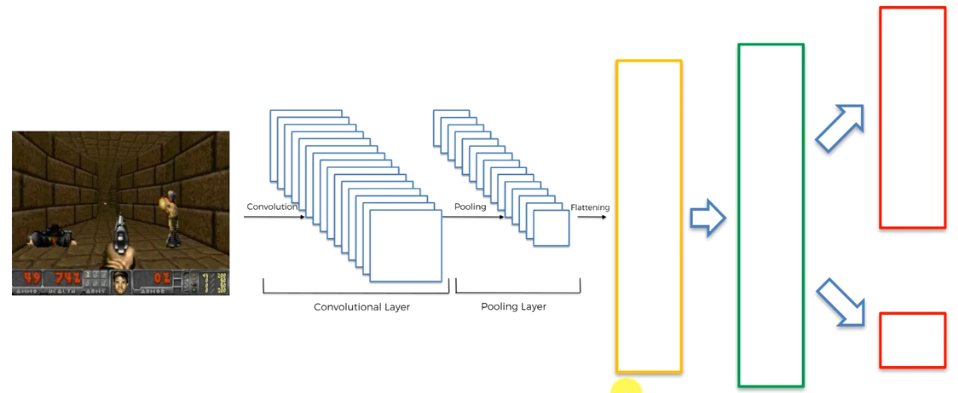
But in this blog, Arthur first talks about: Asynchronous > Actor-Critic > Advantage



**4.6 LSTM Layer**

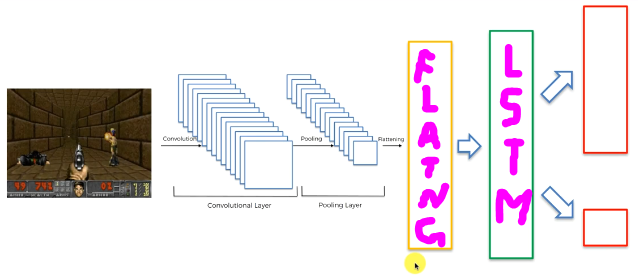
Previously we mentioned that we have multiple agents that exploring the same environment. Now we want to apply LSTM in those agents.

* For simplicity's sake, in this section, we're going to consider a single agent.



* What we've got here: The ***image*** goes through ***CNN*** and outputs the ***numerical-representation of the environment*** then those are propagated through the "network and hidden layers" (A3C) and then as the output we get the "policy or the ACTOR part" and they get the "value of the state or the CRITIC part".

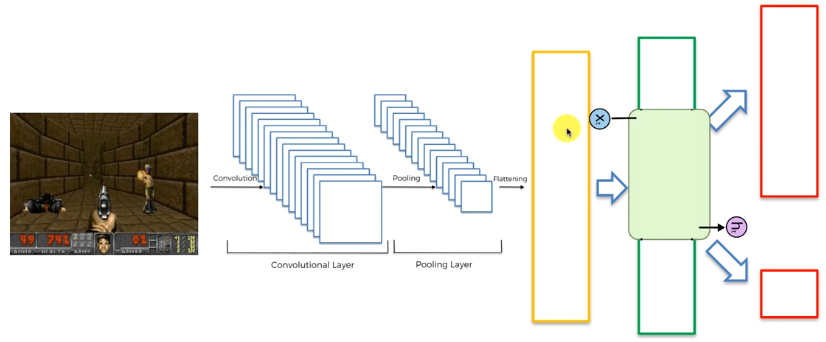
|  |  |
| --- | --- |
| * Where do we apply LSTM?: We'll apply LSTM-layer just before the hidden layers of our A3C model. LSTM allows your A3C algorithm to have memory, to remember what happened before. * Also note that, we don't even need any ***hidden*** ***layers*** after the ***LSTM***. * In our project, we'll apply ***LSTM*** just after the ***Flattening-layer*** of ***CNN*** and from the ***LSTM*** we'll directly get the outputs (policy and value). |  |



* But you are free to use the hidden-layers and LSTM-layers together. For example: *five hidden layers* with *several LSTM layers*. That's totally up to you and for you to experiment and explore.
* Other modifications:
* Shared CNN layer: All agents share the same CNN (main network).
* Separate CNN: Each agent will have separate CNN (main network).

In our application part, we'll use the shared-network, because it is more appropriate.

* LSTM as a feature: The LSTM layer allows the neural network to have memory about what happened in the previous iterations and it is often symbolized as below:



|  |  |
| --- | --- |
| * Here is a vector from previous layer (flattening-layer of CNN), LSTM layer takes it as a input and then it outputs . Note that these are not just vectors, they are layers. * Inside LSTM: There is the point-wise-Operation, layer –wise-operations if you'd like to learn more about LSTM you can read details from Christopher Ola's blog. (Also we've discussed LSTM in Intro ML-DL book in Chapter 12 : Part 2). * Basically this is the internal part of the system. We input a layer, LSTM process this layer and it outputs the processed layer. |  |
| * Additional inputs: There's actually additional inputs into this layer. * What we have now is called a memory cell. * is actually a Time-axis. This is unraveled in time. * and is from past iteration and and are going to the future. * *You could just think of it as a like some memory:* like a flash drive or something like that that it just remembers the previous value and then it can use that to add something more to it or read from it and so on. |  |

* is the hidden state it's another value that comes from the past and then is used inside LSTM. Then a new is output and same value also past forward.

|  |  |
| --- | --- |
| * So basically the LSTM remembers two things:  1. A constant value that acts like a flash drive so that you have the luxury of storing something in that memory and it will be passed onto the future (in the next iteration). 2. The other value LSTM will remember is its output .  * **Why do we need LSTM in A3C:** Why do we need memory in our A3C algorithm? * In case of playing video game, we can cosider it as a bunch of images. It's kind of film-reel, where each frame represents a state of continuous events. * Last's consider our A3C playing the "Breakout-Game". In which we break the little blocks using a little moving ball which we bounce using a moving flat platform. * Wherever the ball is flying it must catch the ball and bounce off the platform and go back and hit blocks of the walls as well. |  |

* If our A3C has no memory, it's just seeing the ball. It has no idea of *"ball's movement"* that is *"Is it going upward or downward?"*.
* The A3C has no idea about the ball's movement, that it's going somewhere and maybe it's flying towards or backward.
* A3C has no idea about the ball's movement from just one image. It is geometrically impossible to tell the direction of a moving ball from just one image (frame). To understand which way the ball is flying, A3C needs the information from the previous image of the flying ball.

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

* **Use the LSTM:** So without memory, our A3C cannot track the balls movement. For this reason we need to implement ***LSTM*** into ***A3C***, so that it keeps track of the ball's movement in the video-game.
* With the LSTM it can track the ball's movement based on the information it gets from the *previous point* of *time*.
* The information isn't suddenly coming from somewhere but from the previous point in time.
* This information of ball's movement helps the algorithm to make a decision.
* Note that, the **LSTMs are not 100% necessary**.
* There's another one that you will see in the practical tutorials where we add ENTROPY which is calculated through policy-loss.
* There are lots of different modifications that can happen in an A3C algorithm, it depends on what you want to achieve.