Shagun Maheshwari

184 Followers About Follow

Implementing the A3C Algorithm to train an Agent to play Breakout!

Shagun Maheshwari Oct 13, 2018 · 7 min read

Deep reinforcement learning is dominating the AI world with it's psychology based model mimicking the way we humans learn and get better at a task.

This differs from supervised learning ML models, as their training data comes with an answer key from some "Godlike" supervisor in order for the model to learn and output accurately. (Only if life worked that way! Lol)

Reinforcement Learning is more realistic in the sense that the agent in the environment learns from experience and exploration. There is no magical answer key that is rewarded to the agent, but rather smaller keys in the form of rewards. The agent will interact with it's environment, look at the **state** it is in, and then carry out an action that results in a new state. After carrying out an action and entering a new state, the agent will get a reward from the environment depending how good or bad that action was. This process turns into a loop allowing the agent to improve it's actions based on the positive/negative reinforcement from the rewards when entering a new state. Reinforcement learning is goal oriented, the goal being to maximize long-term reward in order for the agent to learn through trial-and-error and output it's desired task accurately.

My Implementation: A3C to play Breakout

A3C is currently one of the most powerful algorithms in deep reinforcement learning. It is basically like the smarter, cooler, hotter sibling of Deep Q networks. I was fascinated by the capabilities with this algorithm so I implemented it to learn and play an ol Atari game called Breakout. My purpose was to see if the A3C model could indefinitely play the game as well as a human, and even surpass it's capabilities.

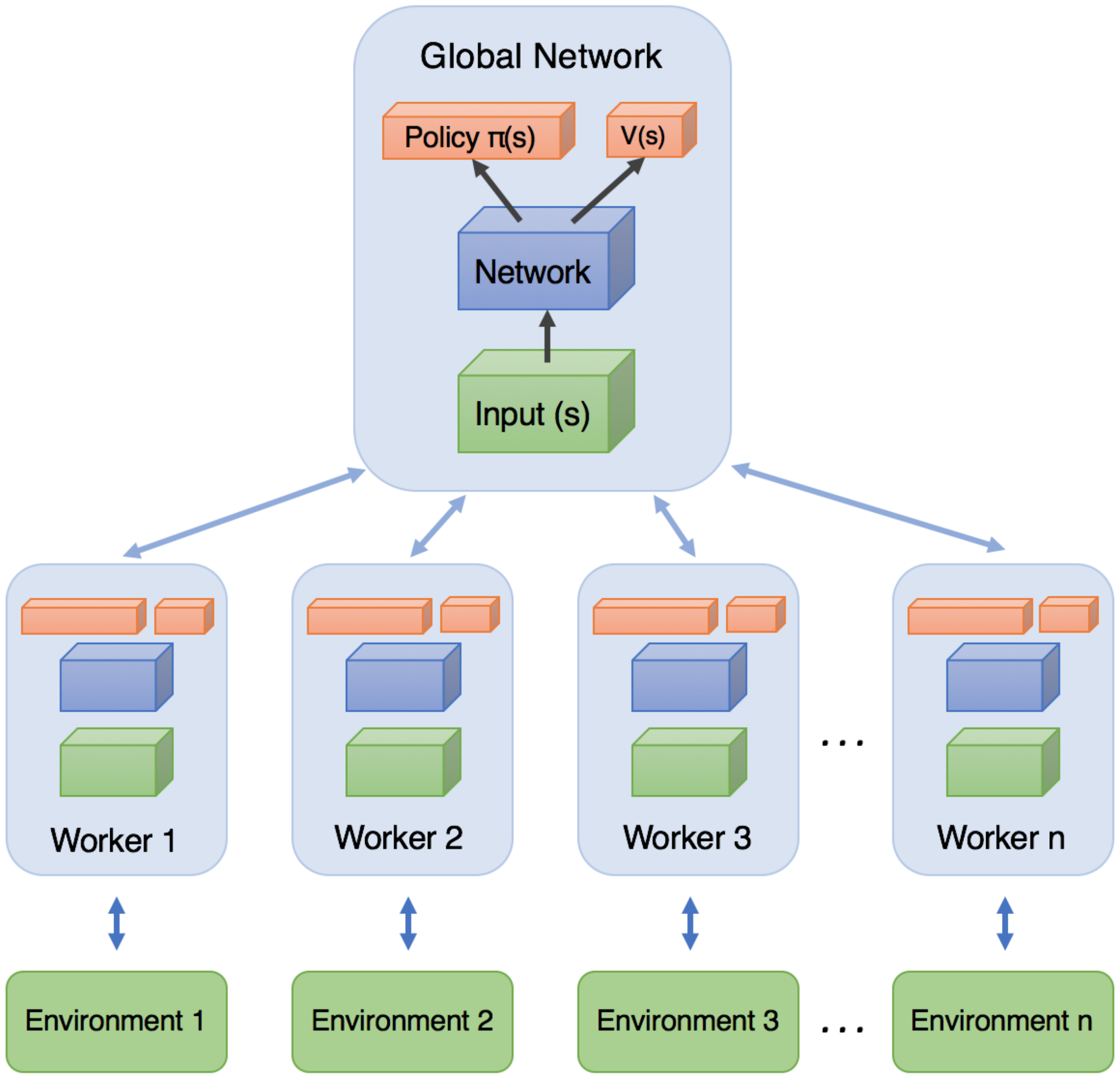


Diagram of A3C high-level architecture.

The 3 A's of A3C: Asynchronous Advantage Actor-Critic.

In order to understand the A3C model, we first have to understand the core of it, which is **Deep Convolutional Q Learning**.

Deep Convolutional Q Learning

^*That's kind of a mouthful as well, there's going to be a lot of mouthfuls during this article.*

Deep Convolutional Q Learning is basically Q-learning but with deep neural networks.

Q-Learning is a reinforcement learning method that uses the Bellman's equation to update it's Q-values. The Q-value [Q(s,a)] determines the **value** of the agent being in a certain **state** and taking a certain **action** to go to that state.

This algorithm calculates the Q-values of taking every action at every state, helping the agent make decisions on it's next move.

$$\underbrace{NewQ(s,a)}_{\text{New Q value for that state and that action}} = \underbrace{Q(s,a)}_{\text{Current Q value}} + \underset{\text{Learning Rate}}{\alpha}[\overbrace{\underbrace{R(s,a)}_{\text{Reward for taking that action at that state}} + \underset{\text{Discount rate}}{\gamma}\, \underbrace{max\,Q'(s',a')}_{\text{Maximum expected future reward given the new s' and all possible actions at that new state}} - Q(s,a)}^{\text{Update Value}}]$$

The Bellman's equation.

Only problem with this is that the Bellman equation only works best and is used in problems/scenarios where there are a small amount of states for the algorithm to calculate.

If we want our agent to be able to preform tasks and solve complex challenges like a human and even outperform a human, we need our model to mimic more human behaviour.

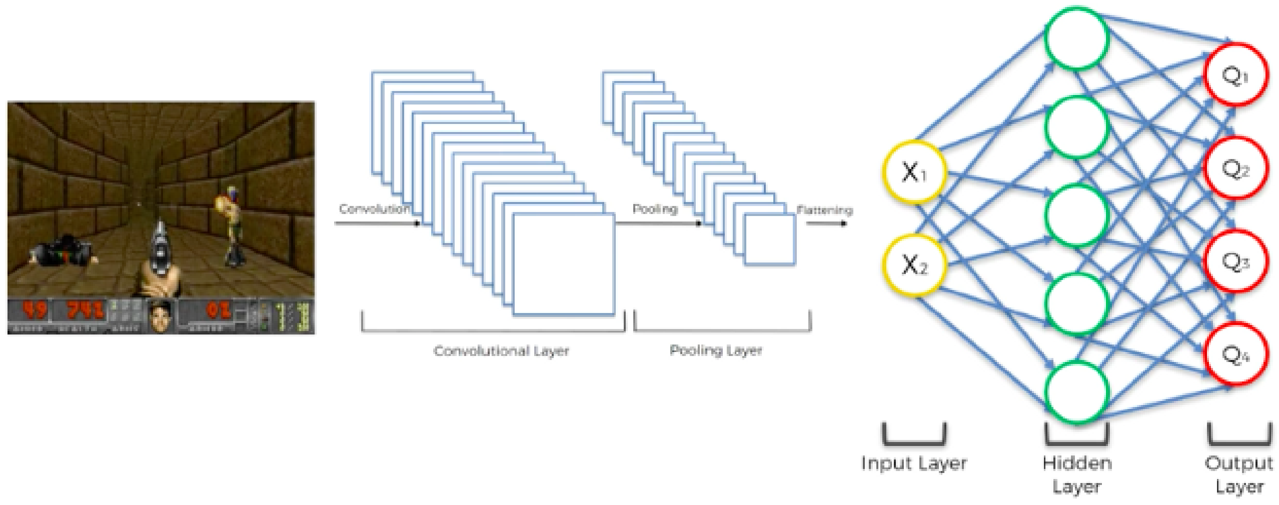
As humans what key factors do we use to retain information and understand our environment?

Our senses, duh!

We use our eyes to perceive the world around us, gain information, process it and then make decisions.

This means we need to figuratively give our agent "sight" for it to maximize long term reward in it's environment.

Now this is the part **Deep Convolution Q learning** comes into play.



Deep Convolution Q learning gives the agent vision.

Now the agent actually has vision of it's environment the same way a human would when playing the game.

The agent processes the images the environment supplies it, through a convolution and pooling layer. Through these layers, the images will be dissected in understanding which state the agent is in and where are the obstacles in the agents environment located.

The outputs of the convolution and pooling layer will then be flattened into numerical vectors to be inputted into the input layer of the artificial neural network.

The final (output layer) of the artificial neural network outputs a vector of different predicted Q(s, a) values for each action possible in the given state. We need to take the biggest Q-value of this vector to find our best action.

In our case, we want to update our neural nets weights to reduce the error.

The error (or TD error) is calculated by taking the difference between our **Q_target** (maximum possible value from the next state) and **Q_value** (our current prediction of the Q-value)

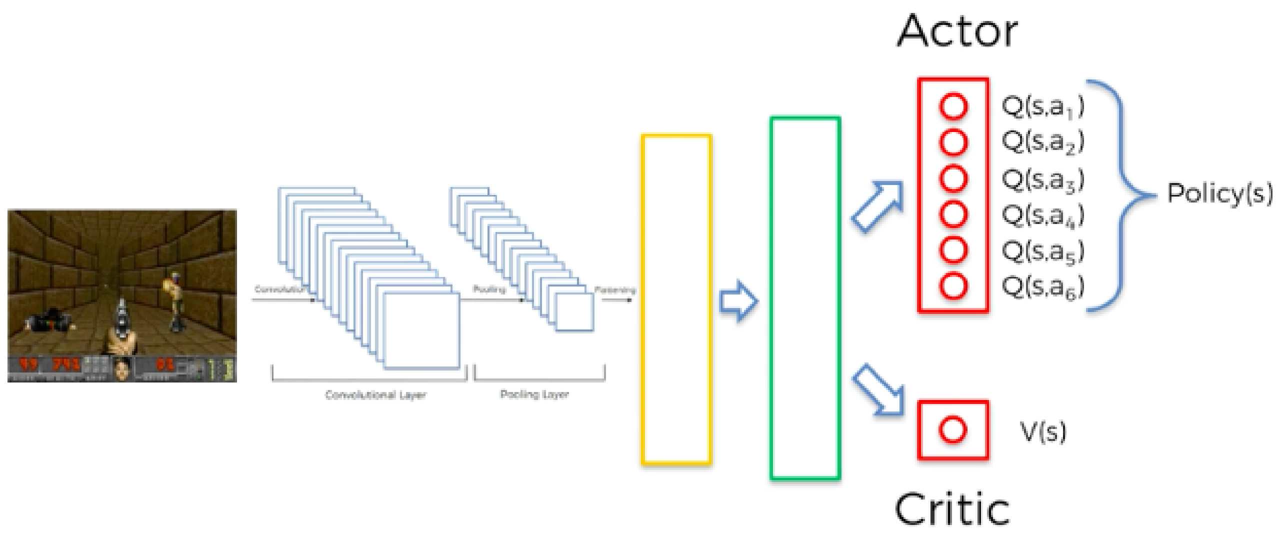
$$\underbrace{\Delta w}_{\text{Change in weights}} = \underbrace{\alpha}_{\text{learning rate}}[\underbrace{\underbrace{(R + \gamma\, max_a\, \hat{Q}(s', a, w))}_{\text{Maximum possible Qvalue for the next\_state (= Q\_target)}} - \underbrace{\hat{Q}(s, a, w)}_{\text{Current predicted Q-val}}}_{\text{TD Error}}]\ \underbrace{\nabla_w \hat{Q}(s, a, w)}_{\text{Gradient of our current predicted Q-value}}$$

Equation to find the error by subtracting the q-target by the predicted q-value.

The error or loss function is then back propagated through our neural network to update the weights so the neural network can output the prediction of actions that result in positive reward.

First A of A3C: Actor-Critic

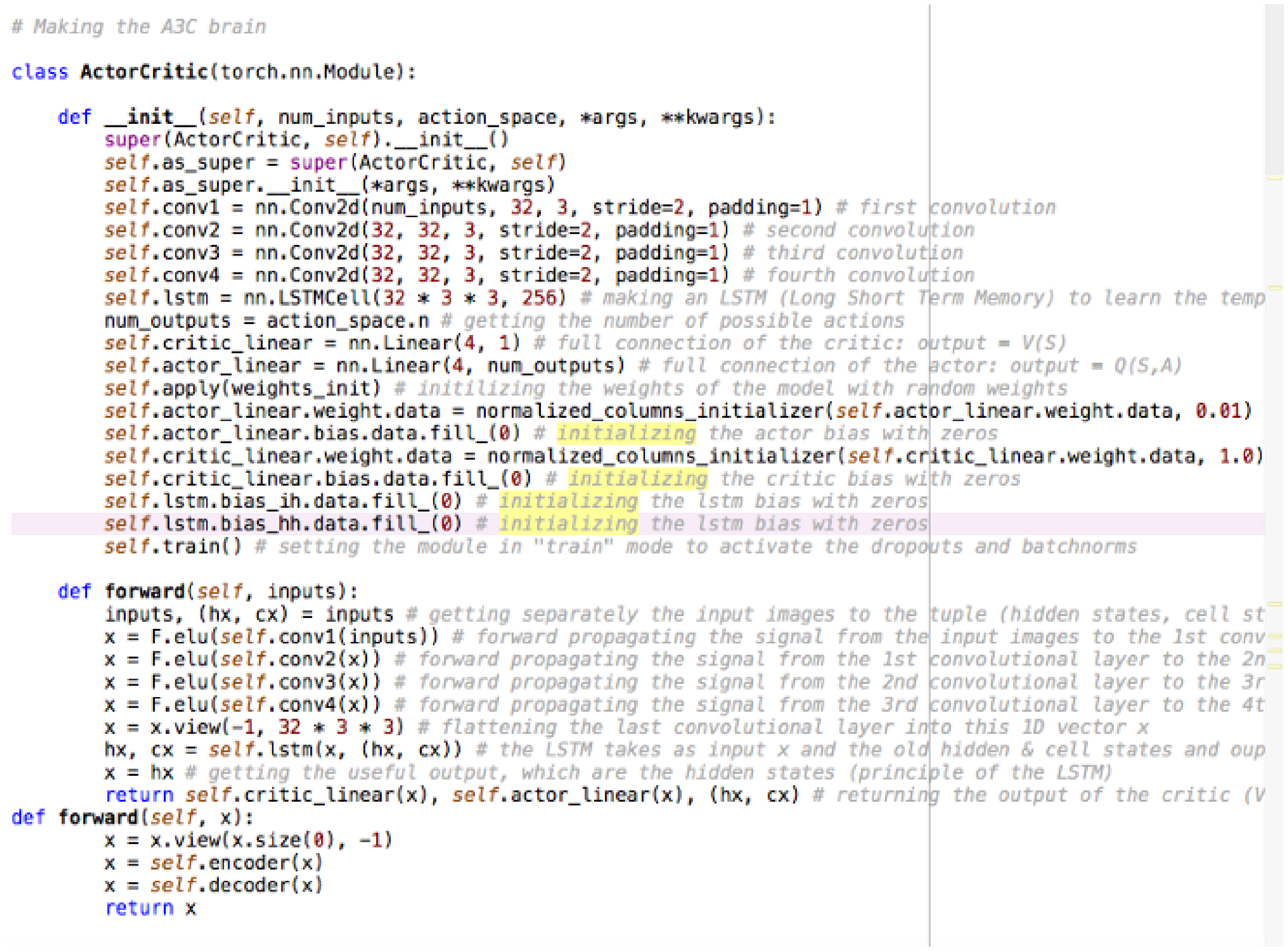
Now we finally get to the juicy stuff and get to implement the A3C model on top of our deep convolutional q learning algorithm.



The Actor-Critic is basically like the brain of the A3C model. At it's core it implements deep convolution Q learning, however the neural network now outputs two different items. The Actor and the Critic.

- The Critic measures how good the action taken is (value-based) V(s)
- The Actor outputs a set of action probabilities the agent can take (policy-based) Q(s,a)

Basically, the agent uses the value estimate (the critic) to update the policy (the actor) so that the actor can output better actions that result in a higher reward.



```
# Making the A3C brain

class ActorCritic(torch.nn.Module):

    def __init__(self, num_inputs, action_space, *args, **kwargs):
        super(ActorCritic, self).__init__()
        self.as_super = super(ActorCritic, self)
        self.as_super.__init__(*args, **kwargs)
        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 temp
        num_outputs = action_space.n # getting the number of possible actions
        self.critic_linear = nn.Linear(4, 1) # full connection of the critic: output = V(S)
        self.actor_linear = nn.Linear(4, 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)
        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)
        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 st
        x = F.elu(self.conv1(inputs)) # forward propagating the signal from the input images to the 1st conv
        x = F.elu(self.conv2(x)) # forward propagating the signal from the 1st convolutional layer to the 2n
        x = F.elu(self.conv3(x)) # forward propagating the signal from the 2nd convolutional layer to the 3r
        x = F.elu(self.conv4(x)) # forward propagating the signal from the 3rd convolutional layer to the 4t
        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 oup
        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
def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

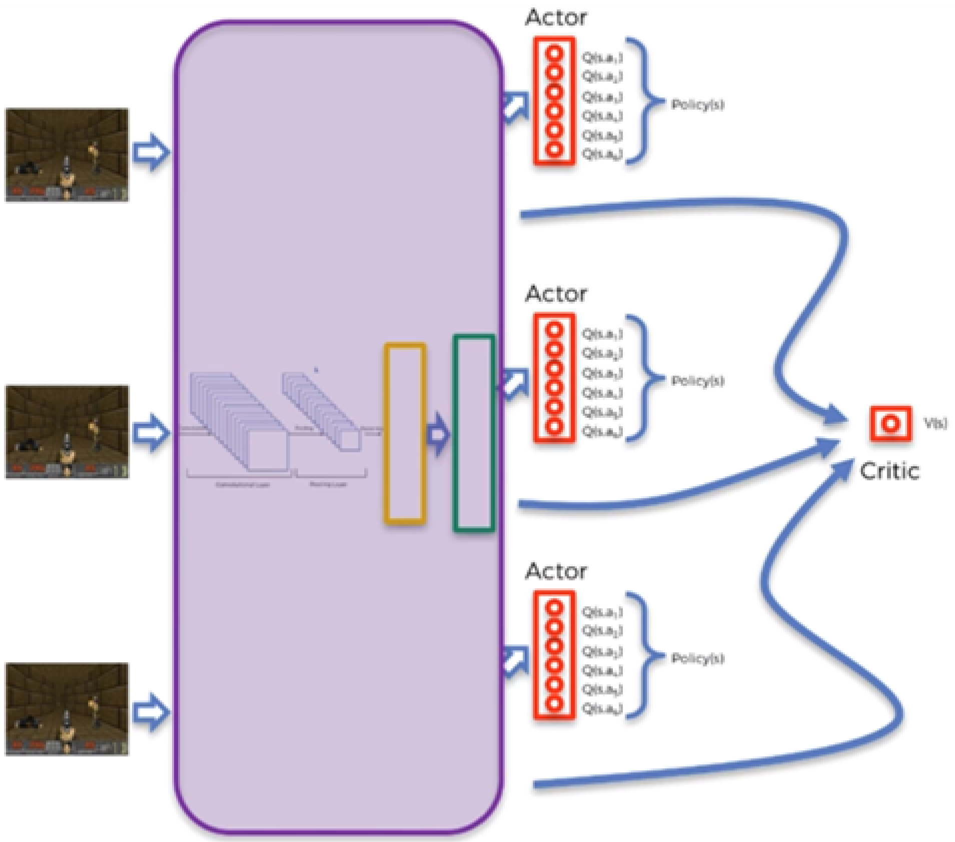
The code implementation of the A3C brain with deep convolution Q learning and the actor and critic.

2nd A of A3C: Asynchronous

initialized differently. This means that each of these agents start at a different point in their environment so they will go through the same environment in different ways to solve the same problem.

The purpose of this is that the agents can share their experience with one another and help each other out while exploring their environment in different ways.

This possible because all the **agents** are now contributing to the same **Critic**.



There is one **global** network now shared between all 3 agents as the critic will help the Q-values of each agent improve.

Final A of A3C: Advantage

The **Advantage** is how the **Critic** tells the **Actor** that it's predicted Q-values from the ANN are **good** or **bad**. It calculates the **policy loss.**

This is calculated through the **Advantage** equation.

Advantage Equation

Basically the advantage subtracts the known value of a state [V(s)] from the predicted selected action from the Actor [Q(s,a)].

The critic knows the value of state but it doesn't know how much better the Q-value that is being selected compared to the current value of state. This is where the advantage comes in. The higher the advantage, the more the agents will look at doing those actions.

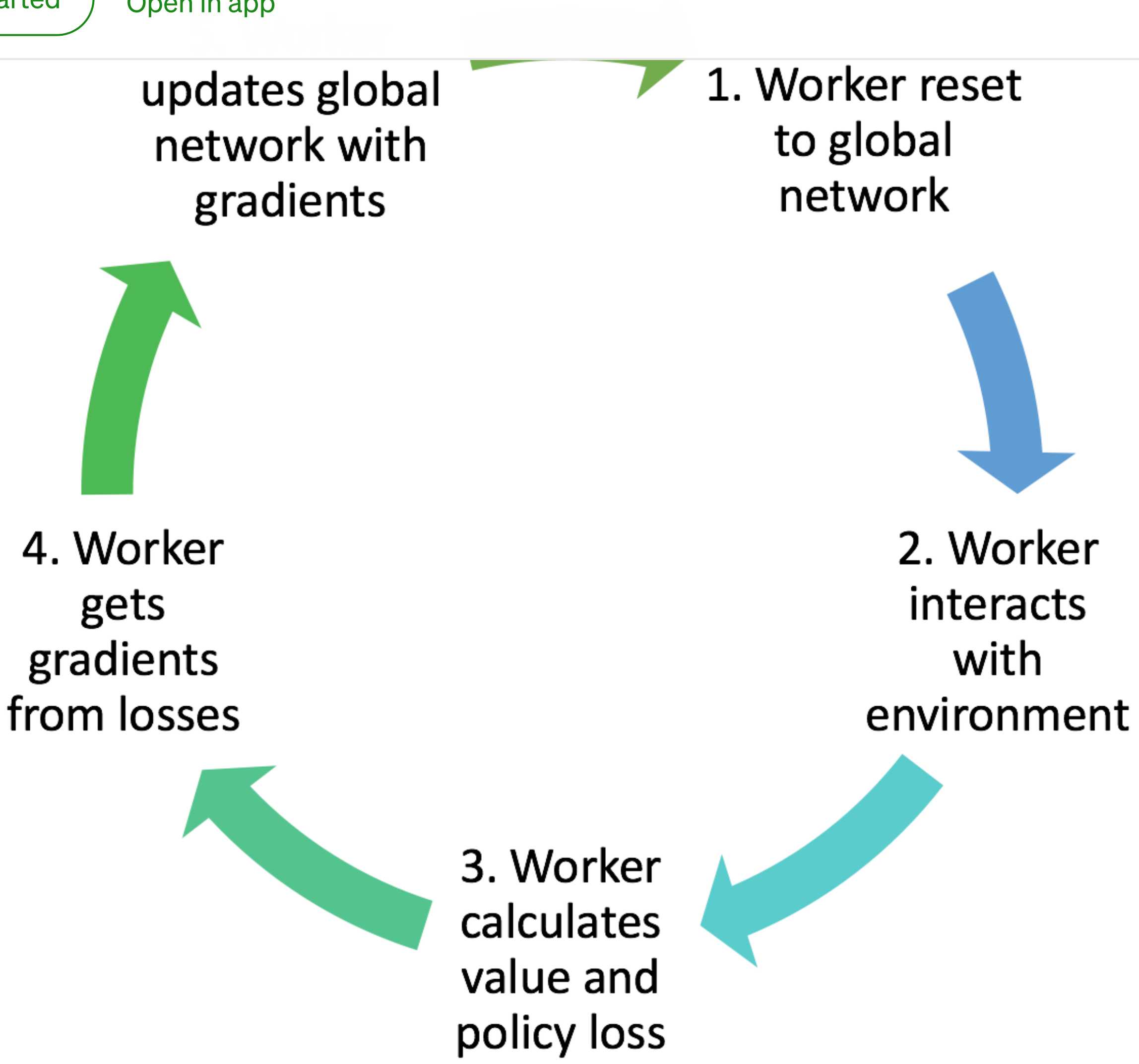
The policy loss helps to improve the agent's behaviour by making them do more of the positive actions rather than the negative impacting ones.

If the Actor selects a good action for the agent then the Q(s,a) > V(s)

If not then the policy loss is back propagated through the **Global Network**, also adjusting the weights in order to maximize Advantage and improve the selected action for the future.

A3C Coming all together

Below is a pretty good looking diagram of the training workflow of each agent with the global network.


updates global network with gradients
1. Worker reset to global network
2. Worker interacts with environment
3. Worker calculates value and policy loss
4. Worker gets gradients from losses

Training workflow of each worker agent in A3C.

A3C To Play BREAKOUT!!!

Using the A3C algorithm I was able to implement it to train my agent to play the Atari game Breakout!

Full code for Breakout!

Snippet of the code used to run the neural network and the actor and critic

This code ties in the Atari game environment, the neural network and trains the Agent and the Critic.

The final video recorded version of my trained agent playing the Atari game Breakout using the A3C algorithm, can be seen below.

As you can see in the first test, the agent has poor output and isn't moving in it's environment.

Now during the 4th trial you can see the agent playing the game fairly well.

And finally during the Last trial you can see the agent playing breakout extremely well!

You the Agent must now take an action to earn a positive reward!

You're Predicted Q(s,a) values are to:

> **1. Clap this post!**
>
> **2. Share with your network!!**
>
> **3. Connect with me on linkedin to stay updated with my journey in AI and exponential technologies!!!**

Machine Learning Reinforcement Learning Artificial Intelligence Exponential Technology

OpenAI