**Advantage Actor Critic Algorithm**

In the field of Reinforcement Learning, the Advantage Actor Critic (A2C) algorithm combines two types of Reinforcement Learning algorithms (Policy Based and Value Based) together. **Policy Based** agents directly learn a policy (a probability distribution of actions) mapping input states to output actions. **Value Based** algorithms learn to select actions based on the predicted value of the input state or action.

# The Advantage Actor-Critic Algorithm Overview

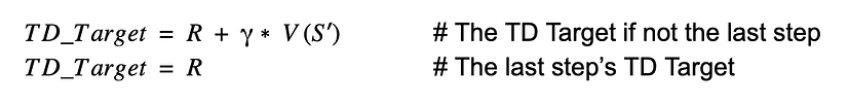
The actor critic algorithm consists of two networks (the actor and the critic) working together to solve a particular problem. At a high level, the **Advantage Function** calculates the agent’s **TD Error** or **Prediction Error**. The actor network chooses an action at each time step and the critic network evaluates the quality or the Q-value of a given input state. As the critic network learns which states are better or worse, the actor uses this information to teach the agent to seek out good states and avoid bad states.

# The Advantage Function

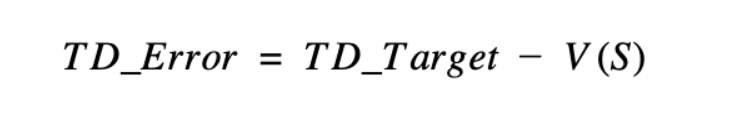
What’s the advantage function? Considering that “Advantage” is in the Advantage Actor Critic algorithm’s name, it must be pretty important. In order to understand what the Advantage Function is, we first need to understand how to calculate the TD Error, or the Temporal Difference Error.

In Temporal Difference Learning, agents learn by making predictions about future rewards and adjusting their actions based on prediction error. One of the reasons Temporal Difference Learning is quite interesting is that prediction error also seems to be one of the ways that the brain learns new things.

# Calculating the TD Error

****

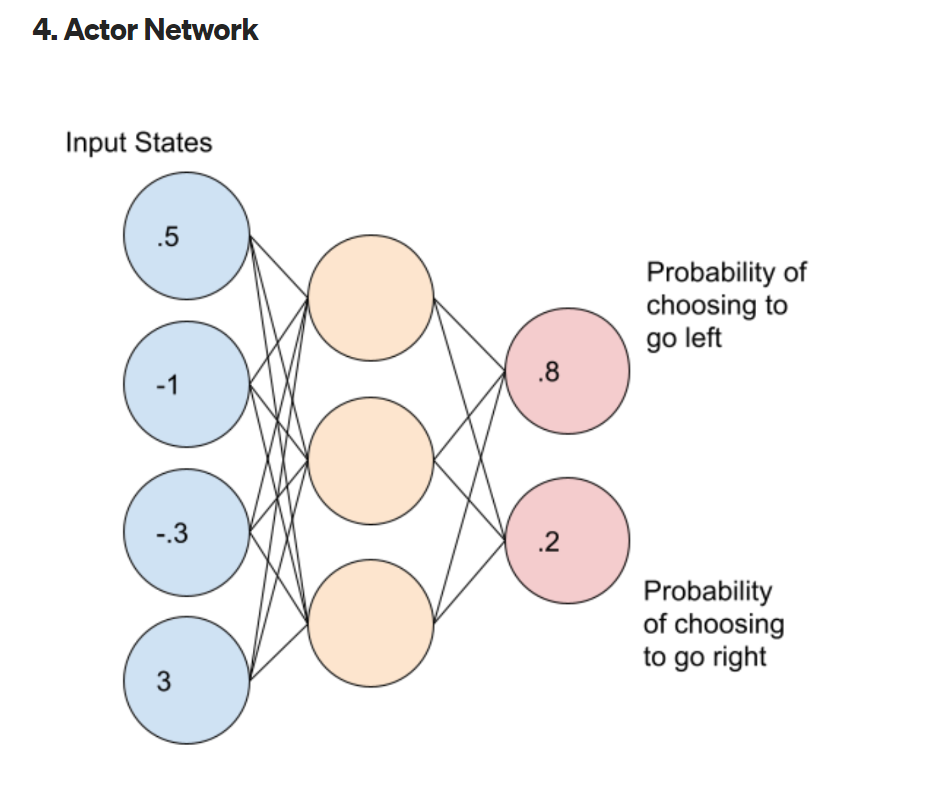
In order to calculate the Advantage Function (TD Error), we need to first calculate the **TD Target**. In the equation above, the TD Target is the predicted value of all future rewards from the current state S. The function V(s’) represents the Critic Network calculating the value of the next state S’.

****

In the Advantage Actor Critic algorithm, the **Advantage** is equal to the **TD Error** shown above.

Note that the advantage function may not always be the same as the TD Error function. For example, in many Policy Gradient algorithms, the advantage is commonly calculated to be the sum of future discounted rewards.

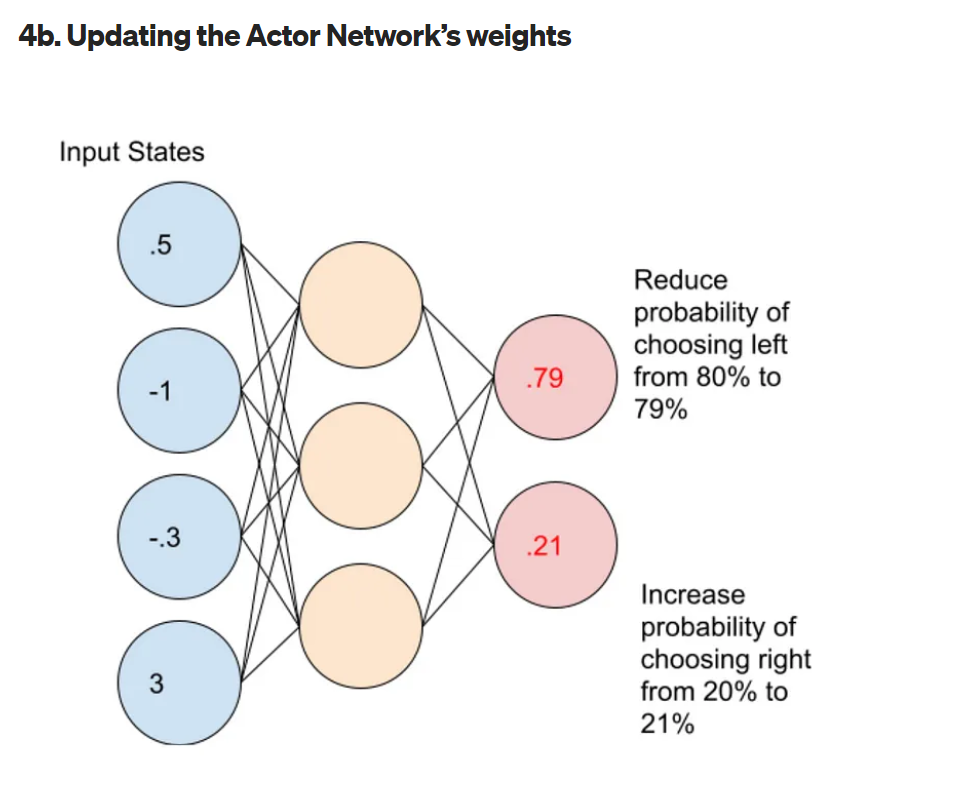
**The Advantage function tells us if a state is better or worse than expected. If an action is better than expected (the advantage is greater than 0), we want to encourage the actor to take more of that action. If an action is worse than expected (the advantage is less than 0), we want to encourage the actor to take the opposite of that action. If an action performs exactly as expected (the advantage equals 0), the actor doesn’t learn anything from that action.**



The actor network maps each **state** to a corresponding **action**. Just like with the Critic Network, we can update the Actor Network weights after every time step.

The actor network outputs a **probability distribution** corresponding to each action. We sample actions from this probability distribution according to each action’s probability. If the action to go left has a value of .8 and the action to go right has a value of .2, we will only choose the left action 80% of the time and the right action 20% of the time. Because the output is a probability distribution, note that the agent will not always choose the action with the highest probability.

In our implementation, the Actor Network is a simple network consisting of 3 densely connected layers with the LeakyReLU activation function. The network uses the Softmax activation function and the **Categorical Cross Entropy loss function because the network outputs a probability distribution of actions.**

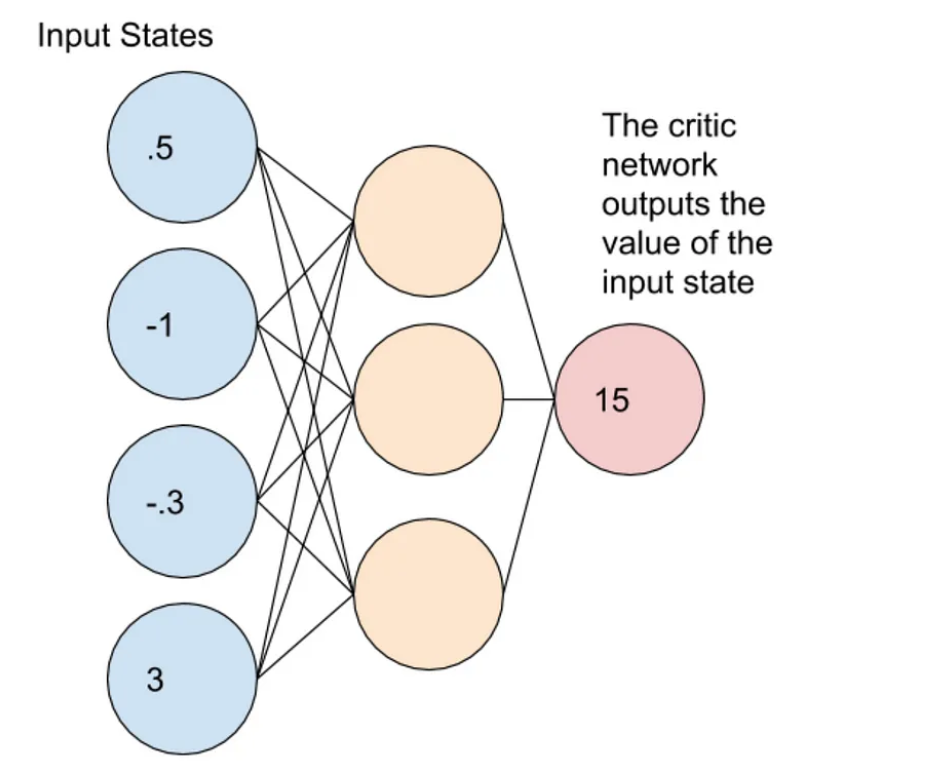


Once we’ve constructed and initialized our actor network, we need to update its weights. In the above example, the agent has decided choosing to go left was the wrong decision. In this case, the agent wants to reduce the probability of choosing left from 80% to 79%. Likewise, the agent needs to increase the probability of choosing right from 20% to 21%. After updating these probabilities, we can update our network weights by fitting the network to the new probabilities.

How does the algorithm decide which actions to encourage and which to discourage? The A2C algorithm makes this decision by calculating the advantage. The advantage decides how to **scale** the action that the agent just took. Importantly the advantage can also be **negative** which discourages the selected action. Likewise, a **positive** advantage would encourage and reinforce that action.

# Critic Network

The critic network maps each **state** to its corresponding **Q-value**. The Q-value represents the value of a state where **Q** represents the **Quality** of the state:

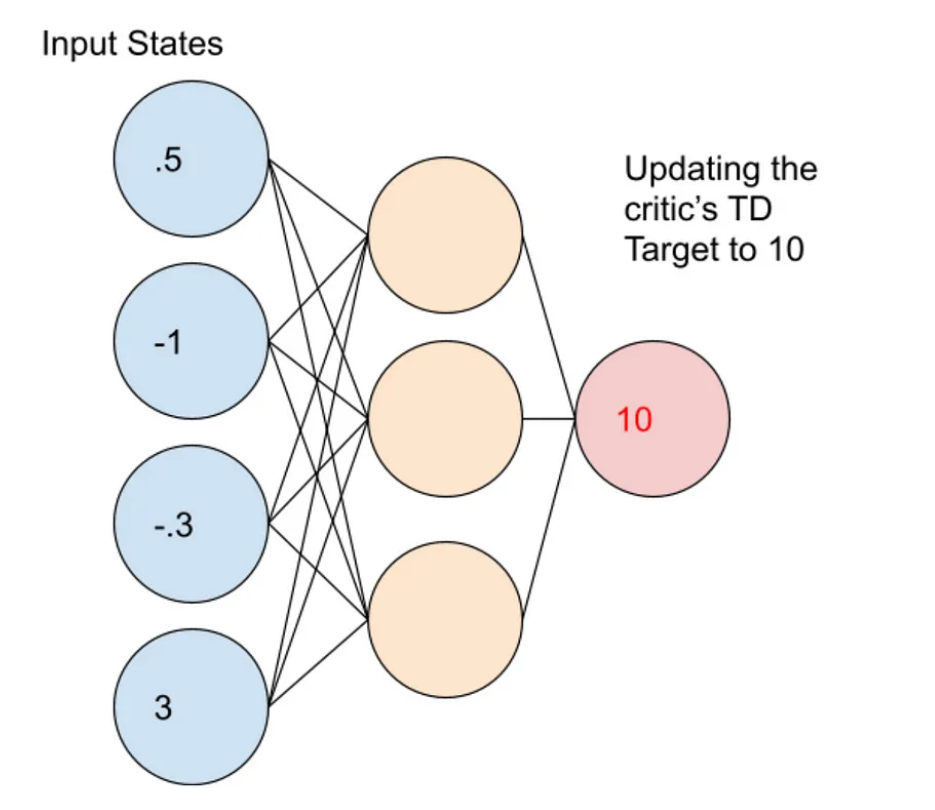


Unlike the Actor Network which outputs a probability distribution of actions, the Critic Network outputs the TD Target of the input state as a floating point number. In the figure above, the critic network evaluates the input state to have a Q-value of 15.

Because the output of the Critic Network is the TD Target, the network is optimized using the Mean Squared Error loss function.

# Updating Critic Network Weights

Now that we know how to calculate the TD Target and the TD Error, how do we update the Critic Network weights? Note that as the TD Error approaches 0, the Critic Network gets better and better at predicting the outcome from the current state. In this case, we want to drive the TD Error as close to 0 as possible. In order to update the critic network weights, we use the Mean Squared Error of the TD Error function.



In order to update the network, we fit our network weights so that they target the new TD Target value of 10. Note that the Advantage Actor Critic algorithm is different than the vanilla Policy Gradient (REINFORCE) algorithm. Instead of waiting for the end of an episode to finish as in the REINFORCE algorithm, we can update the critic network after every time step.

# Implementation Details

As the agent explores its environment, **the critic network is attempting to drive the advantage function to 0.** At the beginning of the learning process, the critic will likely make large errors causing the calculated TD error, advantage function, to be quite incorrect. Because the algorithm starts out with the critic having no knowledge of the environment, the actor similarly can’t learn much from the critic. As the critic starts to make more and more accurate predictions, the calculated TD error (Advantage) becomes more accurate, getting closer to 0. The actor is able to learn from the increasingly accurate TD error to decide if a move was good or bad.

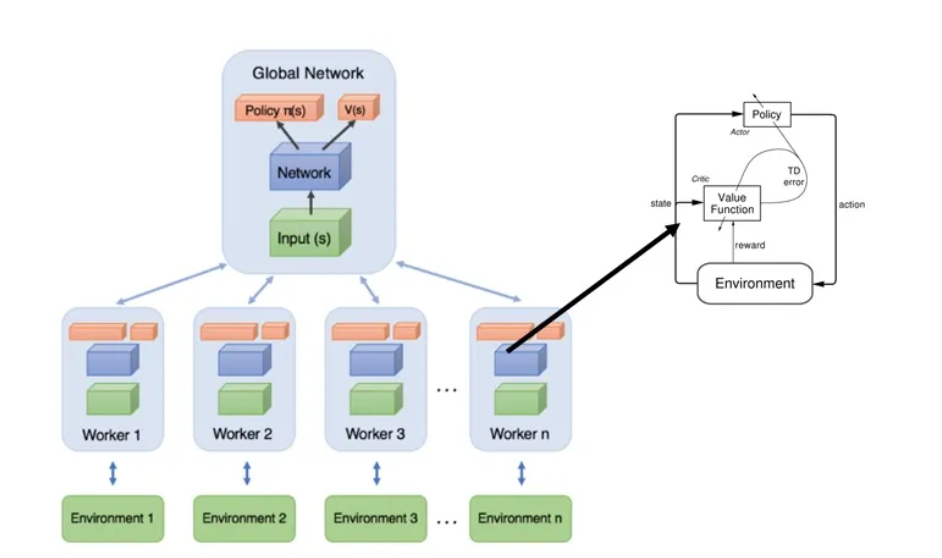
For example:

For the CartPole problem, the reward of 1 at each time step stops coming when the game ends. The unexpected ending of rewards causes the actor to figuratively think, “that move was worse than expected, let’s try something else next time”. By repeating this learning process over many game episodes, the critic and actor together learn to balance the pole for longer periods of time.

**The** **Advantage Actor Critic**has two main variants:

the **Asynchronous Advantage Actor Critic (A3C)** and the **Advantage Actor Critic (A2C).**

**A3C** was introduced in [Deepmind’s paper “Asynchronous Methods for Deep Reinforcement Learning” (Mnih et al, 2016)](https://arxiv.org/abs/1602.01783" \t "_blank). In essence, A3C implements parallel training where multiple workers in parallel environments independently update a global value function—hence “asynchronous.” One key benefit of having asynchronous actors is effective and efficient exploration of the state space.



You can figure out the biggest difference by looking at the name of this mysterious architecture: **Asynchronous** Advantage Actor-Critic. In DQN, a single agent (or so-called worker) interacts with a single environment, generating training data. The A3C launches several workers asynchronously (as much as your CPU can handle) and lets them all interact with their own instance of the environment. They also train their own copy of the network and share their results at the end of the simulation.

## The Advantage

So why exactly is this better than a traditional DQN? There are multiple reasons for that. First, by asynchronously launching more workers, you are essentially going to collect as much more training data, which makes the collection of the data faster.

Since every single worker instance also has their own environment, you are going to get more diverse data, which is known to make the network more robust and generates better results!

The agents (or workers) are trained in parallel and update periodically a global network, which holds shared parameters. The updates are not happening simultaneously and that’s where the asynchronous comes from. After each update, the agents resets their parameters to those of the global network and continue their independent exploration and training for n steps until they update themselves again.

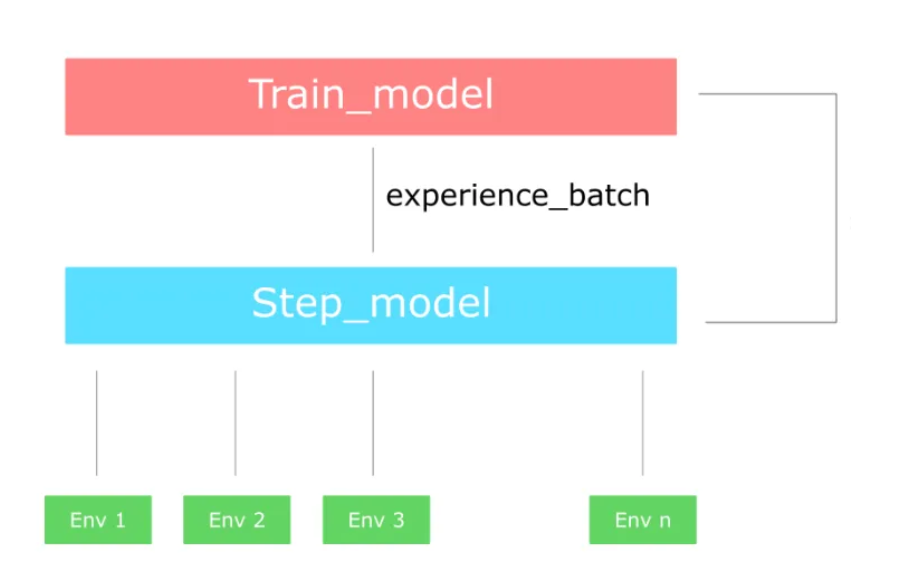
We see that the information flows not only from the agents to the global network but also between agents as each agent resets his weights by the global network, which has the information of all the other agents. Smart right?

In that case, we have an improved version of A2C with multiple agents instead of one. A2C will wait for all the agents to finish their segment and then update the global network weights and reset all the agents.

But. There is always a but. Some argue that there is no need to have many agents if they are synchronous, as they essentially are not different at all. And I agree. In fact, what we do, is to create **multiple versions of the environment** and just two networks.

The first network (usually referred to as step model) interacts with all the environments for n time steps in parallel and outputs a batch of experiences. With those experience, we train the second network (train model) and we update the step model with the new weights. And we repeat the process.

Each actor performs in a different environment:



they are not difficult models to implement as they rely on the same ideas as Policy Gradients and Deep Q Networks.

**The idea of combining policy and value-based method is since 2018, considered standard for solving reinforcement learning problems. Most modern algorithms rely on actor-critics and expand this basic idea into more sophisticated and complex techniques. Some examples are: Deep Deterministic Policy Gradients(DDPG),Proximal Policy Optimization (PPO), Trust Region Policy Optimization (TRPO).**