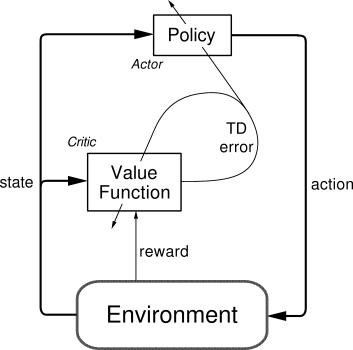
**ACTOR-CRITIC MODEL**

Actor-critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the *actor*, because it is used to select actions, and the estimated value function is known as the *critic*, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error. This scalar signal is the sole output of the critic and drives all learning in both actor and critic.



Actor-critic methods are the natural extension of the idea of reinforcement comparison methods (Section 2.8) to TD learning and to the full reinforcement learning problem. Typically, the critic is a state-value function. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. That evaluation is the TD error:

|  |
| --- |
|  |

where  is the current value function implemented by the critic. This TD error can be used to evaluate the action just selected, the action  taken in state . If the TD error is positive, it suggests that the tendency to select  should be strengthened for the future, whereas if the TD error is negative, it suggests the tendency should be weakened.

**AC Model Overview**

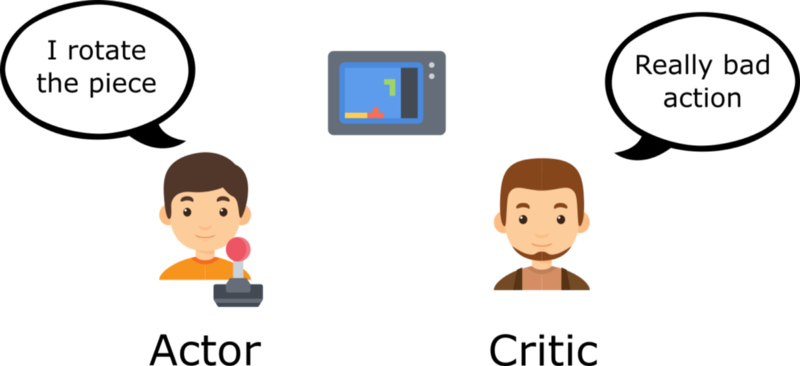
Therefore, we have to develop an ActorCritic class that has some overlap with the DQN we previously implemented, but is more complex in its training. Because we’ll need some more advanced features, we’ll have to make use of the underlying library Keras rests upon: Tensorflow. Note: You can definitely implement this in Theano as well, but I haven’t worked with it in the past and so have not included its code. Feel free to submit expansions of this code to Theano if you choose to do so to me!

The model implementation will consist of four main parts, which directly parallel how we implemented the DQN agent:

* Model parameters/setup
* Training code
* Prediction code

**How Actor Critic works**

Imagine you play a video game with a friend that provides you some feedback. You’re the Actor and your friend is the Critic.

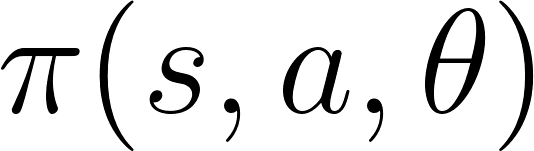


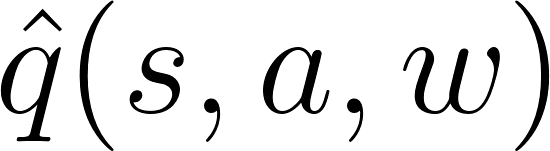
At the beginning, you don’t know how to play, so you try some action randomly. The Critic observes your action and provides feedback.

Learning from this feedback, **you’ll update your policy and be better at playing that game.**

On the other hand, your friend (Critic) will also update their own way to provide feedback so it can be better next time.

As we can see, the idea of Actor Critic is to have two neural networks. We estimate both:

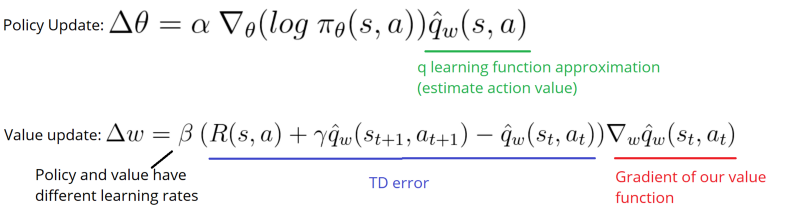


**ACTOR** : A policy function, controls how our agent acts.

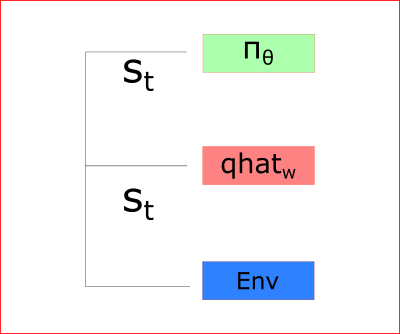
**CRITIC**:A value function, measures how good these actions are.

Both run in parallel.

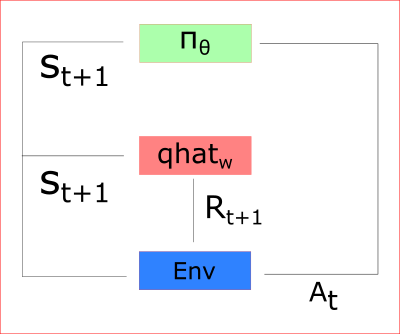
Because we have two models (Actor and Critic) that must be trained, it means that we have two set of weights (? for our action and w for our Critic) t**hat must be optimized separately:**



**The Actor Critic Process**



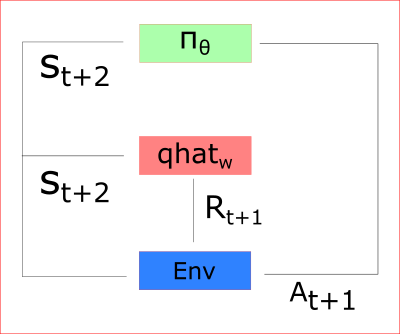
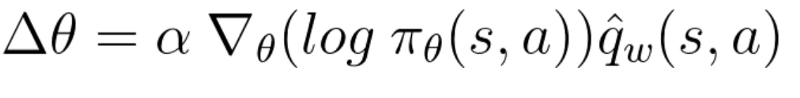
At each time-step t, we take the current state (St) from the environment and pass it as an input through our Actor and our Critic.



Our Policy takes the state, outputs an action (At), and receives a new state (St+1) and a reward (Rt+1).

Thanks to that:

* the Critic computes the value of taking that action at that state
* the Actor updates its policy parameters (weights) using this q value



Thanks to its updated parameters, the Actor produces the next action to take at At+1 **given** the new state St+1. The Critic then updates its value parameters:



**Q-LEARNING**

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It’s considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn’t needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

The ‘q’ in q-learning stands for quality. Quality in this case represents how useful a given action is in gaining some future reward.

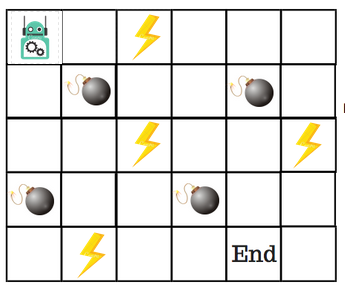
**Example : -**

Let’s say that a robot has to cross a maze and reach the end point. There are mines, and the robot can only move one tile at a time. If the robot steps onto a mine, the robot is dead. The robot has to reach the end point in the shortest time possible.

The scoring/reward system is as below:

1. The robot loses 1 point at each step. This is done so that the robot takes the shortest path and reaches the goal as fast as possible.
2. If the robot steps on a mine, the point loss is 100 and the game ends.
3. If the robot gets power ⚡️, it gains 1 point.
4. If the robot reaches the end goal, the robot gets 100 points.

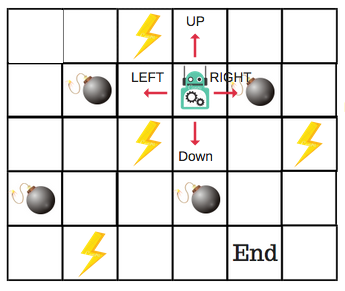
Now, the obvious question is: How do we train a robot to reach the end goal with the shortest path without stepping on a mine?



So, how do we solve this?

**Introducing the Q-Table**

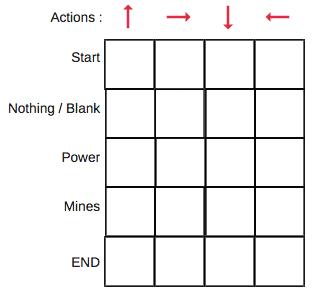
Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state.



There will be four numbers of actions at each non-edge tile. When a robot is at a state it can either move up or down or right or left.

So, let’s model this environment in our Q-Table.

In the Q-Table, the columns are the actions and the rows are the states.



Each Q-table score will be the maximum expected future reward that the robot will get if it takes that action at that state. This is an iterative process, as we need to improve the Q-Table at each iteration.

But the questions are:

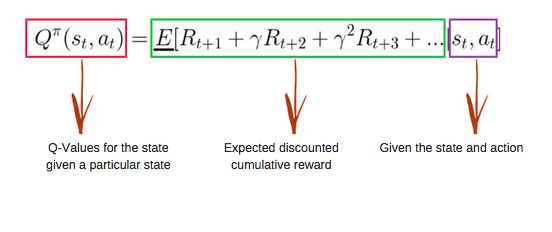
* How do we calculate the values of the Q-table?
* Are the values available or predefined?

To learn each value of the Q-table, we use the**Q-Learning algorithm.**

**Mathematics: the Q-Learning algorithm**

**Q-function**

The **Q-function** uses the Bellman equation and takes two inputs: state (**s**) and action (**a**).



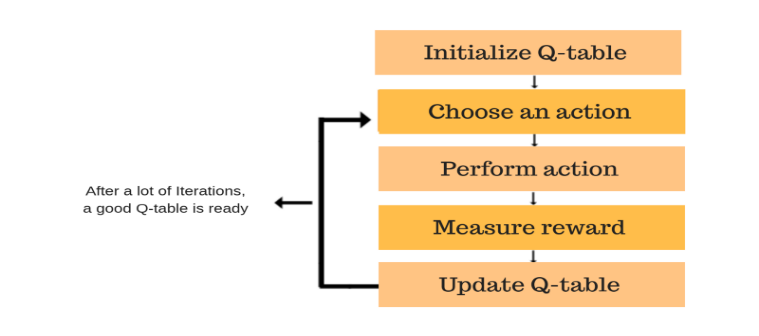
Using the above function, we get the values of **Q** for the cells in the table.

When we start, all the values in the Q-table are zeros.

There is an iterative process of updating the values. As we start to explore the environment**,** the Q-function gives us better and better approximations by continuously updating the Q-values in the table.

Now, let’s understand how the updating takes place.

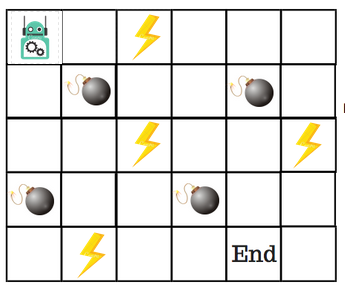
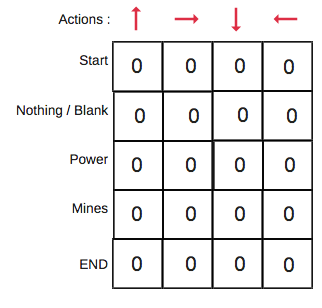
**Introducing the Q-learning algorithm process**



Each of the colored boxes is one step. Let’s understand each of these steps in detail.

**Step 1: Initialize the Q-Table**

We will first build a Q-table. There are n columns, where n= number of actions. There are m rows, where m= number of states. We will initialise the values at 0.



In our robot example, we have four actions (a=4) and five states (s=5). So we will build a table with four columns and five rows.

**Steps 2 and 3: choose and perform an action**

This combination of steps is done for an undefined amount of time. This means that this step runs until the time we stop the training, or the training loop stops as defined in the code.

We will choose an action (a) in the state (s) based on the Q-Table. But, as mentioned earlier, when the episode initially starts, every Q-value is 0.

So now the concept of exploration and exploitation trade-off comes into play. [This article has more details](https://medium.freecodecamp.org/a-brief-introduction-to-reinforcement-learning-7799af5840db).

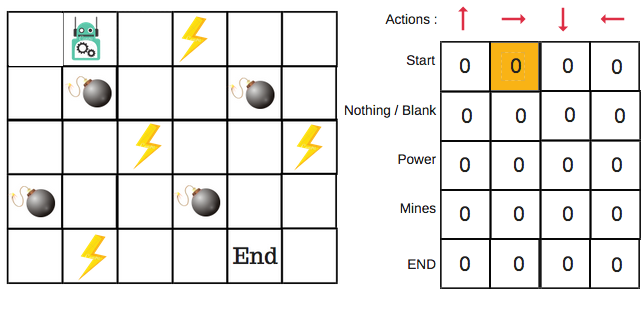
We’ll use something called the **epsilon greedy strategy**.

In the beginning, the epsilon rates will be higher. The robot will explore the environment and randomly choose actions. The logic behind this is that the robot does not know anything about the environment.

As the robot explores the environment, the epsilon rate decreases and the robot starts to exploit the environment.

During the process of exploration, the robot progressively becomes more confident in estimating the Q-values.

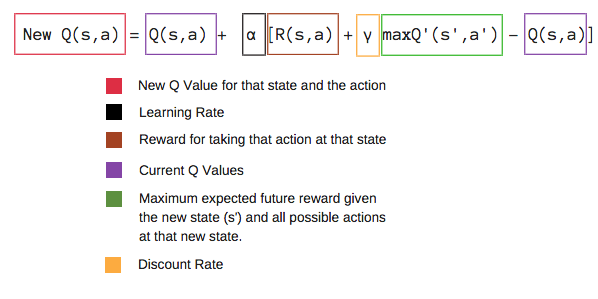
**For the robot example, there are four actions to choose from**: up, down, left, and right.We are starting the training now — our robot knows nothing about the environment. So the robot chooses a random action, say right.



We can now update the Q-values for being at the start and moving right using the Bellman equation.

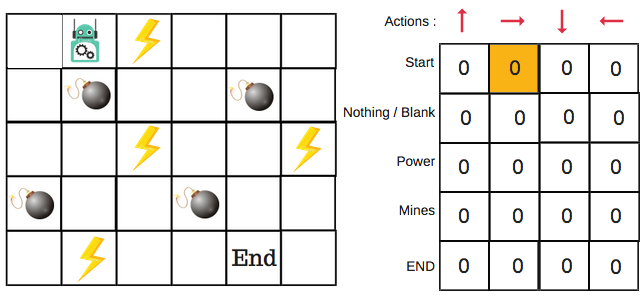
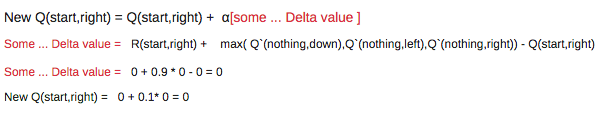
**Steps 4 and 5: evaluate**

Now we have taken an action and observed an outcome and reward.We need to update the function Q(s,a).



In the case of the robot game, to reiterate the scoring/reward structure is:

* **power** = +1
* **mine** = -100
* **end** = +100

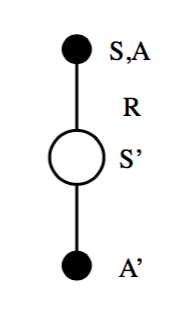


We will repeat this again and again until the learning is stopped. In this way the Q-Table will be updated.

**SARSA**

**State–action–reward–state–action** (**SARSA**) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for learning a [Markov decision process](https://en.wikipedia.org/wiki/Markov_decision_process) policy, used in the [reinforcement learning](https://en.wikipedia.org/wiki/Reinforcement_learning) area of [machine learning](https://en.wikipedia.org/wiki/Machine_learning).

This name simply reflects the fact that the main function for updating the Q-value depends on the current state of the agent "**S**1", the action the agent chooses "**A**1", the reward "**R**" the agent gets for choosing this action, the state "**S**2" that the agent enters after taking that action, and finally the next action "**A**2" the agent chooses in its new state. The acronym for the quintuple (st, at, rt, st+1, at+1) is SARSA.[[2]](https://en.wikipedia.org/wiki/State%E2%80%93action%E2%80%93reward%E2%80%93state%E2%80%93action#cite_note-2) Some authors use a slightly different convention and write the quintuple (st, at, rt+1, st+1, at+1), depending to which time step the reward is formally assigned. The rest of the article uses the former convention.



**Algorithm{\displaystyle Q(s\_{t},a\_{t})\leftarrow Q(s\_{t},a\_{t})+\alpha [r\_{t}+\gamma Q(s\_{t+1},a\_{t+1})-Q(s\_{t},a\_{t})]}**

A SARSA agent interacts with the environment and updates the policy based on actions taken, hence this is known as an *on-policy learning algorithm*. The Q value for a state-action is updated by an error, adjusted by the [learning rate](https://en.wikipedia.org/wiki/Learning_rate) alpha. Q values represent the possible reward received in the next time step for taking action *a* in state *s*, plus the discounted future reward received from the next state-action observation.

Watkin's [Q-learning](https://en.wikipedia.org/wiki/Q-learning) updates an estimate of the optimal state-action value function {\displaystyle Q^{\*}} based on the maximum reward of available actions. While SARSA learns the Q values associated with taking the policy it follows itself, Watkin's Q-learning learns the Q values associated with taking the optimal policy while following an [exploration/exploitation](https://en.wikipedia.org/wiki/Reinforcement_learning) policy.

Some optimizations of Watkin's Q-learning may be applied to SARSA.