

Understanding Reinforcement Learning

Anirban Bhattacharyya
Muntasir Sheikh
Sayak Chatterjee

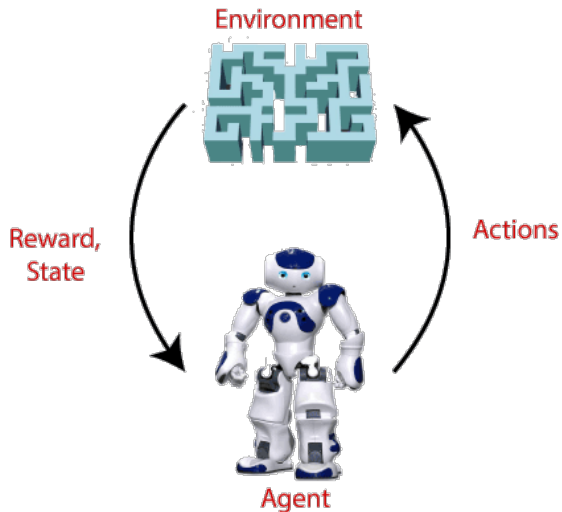
Indian Statistical Institute, Kolkata

- Reinforcement Learning (RL) is a feedback-based Machine learning technique. Here an agent learns to behave in an environment by performing the actions and seeing the results of actions. For each good action, the agent gets positive feedback or reward, and for each bad action, the agent gets negative feedback or penalty.
- In Reinforcement Learning, the agent learns automatically using feedback without any labeled data.
- Since there is no labeled data, so the agent is bound to learn by its experience only.
- RL solves a specific type of problem where decision making is sequential, and the goal is long-term, such as game-playing, robotics, etc.

Introduction (Contd.)

- The agent learns with the process of hit and trial. Based on experience, it learns to perform the tasks in a better way. Hence, we can say that **“Reinforcement learning is a type of machine learning method where an agent (computer program) interacts with the environment and learns to act within that.”** How a Robotic dog learns the movement of his arms is an example of Reinforcement learning.
- It is a core part of Artificial intelligence. Here we do not need to pre-program the agent, as it learns from its own experience without any human intervention.
- The agent continues doing these three things (take action, change state/remain in the same state, and get feedback), and by doing these actions, he learns and explores the environment.
- The agent learns that what actions lead to positive feedback or rewards and what actions lead to negative feedback penalty. As a positive reward, the agent gets a positive point, and as a penalty, it gets a negative point.

Introduction (Contd.)



Important Terminologies

- **Agent:** An entity that can perceive/explore the environment and act upon it.
- **Environment:** A situation in which an agent is present or surrounded by. In RL, we assume the stochastic environment, which means it is random in nature.
- **Action:** Actions are the moves taken by an agent within the environment.
- **State:** State is a situation returned by the environment after each action taken by the agent.
- **Reward:** A feedback returned to the agent from the environment to evaluate the action of the agent.
- **Policy:** Policy is a strategy applied by the agent for the next action based on the current state.
- **Value:** It is expected long-term returned with the discount factor and opposite to the short-term reward.

k -Armed Bandit

Suppose you are faced repeatedly with a choice among k different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps.

In our k -armed bandit problem, each of the k actions has an expected or mean reward when that action is selected; let us call this the value of that action. We denote the action selected on time step t as A_t , and the corresponding reward as R_t . The value then of an arbitrary action a , denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) = \mathbb{E}[R_t | A_t = a]$$

We assume that you do not know the action values with certainty, although you may have estimates. We denote the estimated value of action a at time step t as $Q_t(a)$.

Action Based Methods

One natural way to estimate $q_*(a)$ is just taking the following sample mean:

$$Q_t(a) = \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbf{I}\{A_i = a\}}{\sum_{i=1}^{t-1} \mathbf{I}\{A_i = a\}}.$$

The simplest action selection rule is to select one of the actions with the highest estimated value.

$$A_t = \operatorname{argmax}_a Q_t(a).$$

This is known as the greedy action selection method. But greedy action selection always exploits current knowledge to maximize immediate reward; it spends no time at all sampling apparently inferior actions to see if they might really be better.

Let's Be Less Greedy

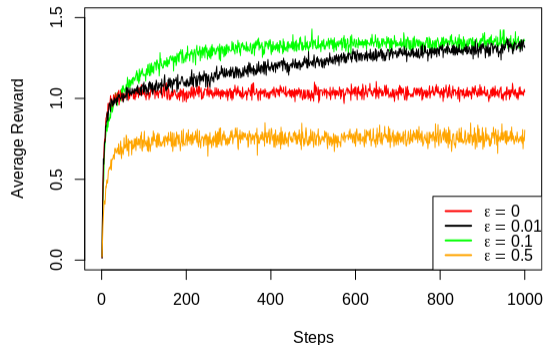
A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ε , instead select randomly from among all the actions with equal probability, independently of the action-value estimates.

$$\begin{cases} \text{Select } A_t \text{ randomly from } k \text{ possible actions} & \text{with probability } \varepsilon \\ A_t = \operatorname{argmax}_a Q_t(a) & \text{with probability } 1 - \varepsilon. \end{cases}$$

We call methods using this near-greedy action selection rule ε -greedy methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the $Q_t(a)$ converge to their respective $q_*(a)$. This of course implies that the probability of selecting the optimal action converges to greater than $1 - \varepsilon$, that is, to near certainty for small ε .

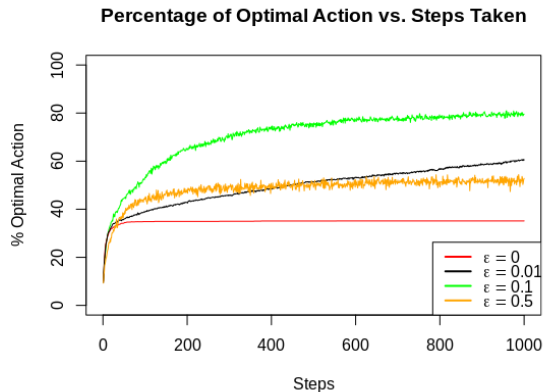
Greedy < ϵ -Greedy

Average Reward vs. Steps Taken



- Here we considered a 10-armed problem. The rewards for each action were Gaussian with a fixed mean and $sd=1$.
- The red curve is for the greedy algorithm. The ϵ -greedy algorithms for $\epsilon = 0.1, 0.01$ are performing significantly better than the greedy algorithm.
- The ϵ -greedy algorithm for $\epsilon = 0.01$ is becoming better than $\epsilon = 0.1$.

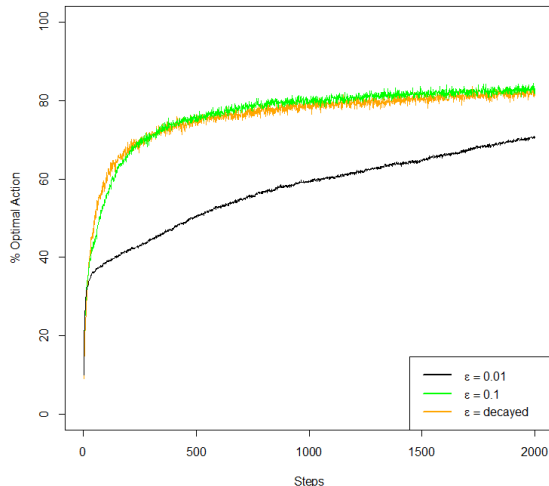
Greedy < ϵ -Greedy (Contd.)



- Since the greedy algorithm sticks to only one particular action, it fails to select the optimal action in most cases.
- Again $\epsilon = 0.01$ seems to be a better choice than $\epsilon = 0.1, 0.5$ as it will become better eventually.

Decayed Epsilon Greedy

Percentage of Optimal Action vs. Steps Taken



- We can avoid setting ϵ value by keeping ϵ dependent on time. For example take $\epsilon_t = \min \left(1, \frac{1}{\log(t+.00001)} \right)$. As time increases ϵ_t decreases making the procedure exploring less at later steps.
- Again $\epsilon = 0.01$ seems to be a better method than decayed ϵ for large number of steps but for less number of steps the later chooses optimal decision more frequently.

Incremental Implementation

To simplify notation we concentrate on a single action. Let R_i denote the reward received after the i -th selection of this action, and let Q_n denote the estimate of its action value after it has been selected $n - 1$ times.

$$Q_n = \frac{R_1 + R_2 + \dots + R_{n-1}}{n - 1} \implies Q_{n+1} = Q_n + \frac{1}{n} \cdot (R_n - Q_n).$$

This gives a recursive way of implementation.

$$\text{New Estimate} \leftarrow \text{Old Estimate} + \text{Step Size} \times (\text{Target} - \text{Old Estimate}).$$

But observe that, keeping the n -th step size to be $\frac{1}{n}$ assigns equal weight to all the previous rewards. This may not be a sensible assumption for most of the real-life problems.

Non-Stationary Problems

As noted earlier, we often encounter reinforcement learning problems that are effectively non-stationary where it makes sense to give more weight to recent rewards than to long-past rewards. For that, one may use constant step size parameter $\alpha \in (0, 1]$.

$$Q_{n+1} = Q_n + \alpha(R_n - Q_n).$$

Observe that

$$Q_{n+1} = \alpha R_n + \alpha(1 - \alpha)R_{n-1} + \alpha(1 - \alpha)^2 R_{n-2} + \dots + \alpha(1 - \alpha)^{n-1} R_1 + (1 - \alpha)^n Q_1.$$

So indeed, the weights on rewards decrease as we go back in time. One may also consider step size parameter varying from step to step. Suppose $\alpha_n(a)$ is the step size for n -th selection of action a . Then $\alpha_n(a) = \frac{1}{n}$ will give the sample average method discussed earlier. For convergence of iterations, a sufficient condition is

$$\sum_n \alpha_n(a) = \infty \quad \& \quad \sum_n \alpha_n^2(a) < \infty.$$

$\alpha_n(a) = \frac{1}{n}$ satisfies both conditions but the constant step size parameter doesn't.

Setup of Markov Decision Process

- \mathcal{S} : State Space, \mathcal{A} : Action Space, $\mathcal{R} \subset \mathbb{R}$: Set of Rewards.
- S_t : State at time t , A_t : Action taken at time t ,
 R_{t+1} : Reward obtained after taking action A_t at time t while staying in state S_t .
- The trajectory is given by:

$$S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_t, A_t, R_{t+1}, \dots$$

- In general, we seek to maximize the expected return, denoted by G_t . In the simplest case the return is

$$G_t = R_{t+1} + R_{t+2} + \dots + R_T$$

where T is the terminal step where the process ends.

- To take care of the situations when T becomes infinite, we bring in the idea of discounting with discount rate γ :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = R_{t+1} + \gamma G_{t+1}.$$

Policies and Value Functions

- For policy π at time t ,

$$\pi(a|s) = \mathbb{P}(A_t = a | S_t = s).$$

- Value function of state s for policy π is

$$\begin{aligned} v_{\pi}(s) &= \mathbb{E}_{\pi}(G_t | S_t = s) = \mathbb{E}_{\pi}(R_{t+1} + \gamma G_{t+1} | S_t = s) \\ &= \sum_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} [r + \gamma \mathbb{E}_{\pi}(G_{t+1} | S_{t+1} = s')] \mathbb{P}(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a) \pi(a|s) \\ \implies v_{\pi}(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} [r + \gamma v_{\pi}(s')] \mathbb{P}(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a). \end{aligned}$$

This is known as Bellman equation for v_{π} .

- Value of taking action a in state s under a policy π is

$$q_{\pi}(s, a) = \mathbb{E}_{\pi}(G_t | S_t = s, A_t = a).$$

Optimal Policies and Optimal Value Functions

- We say

$$\pi \geq \pi' \iff v_{\pi}(s) \geq v_{\pi'}(s) \forall s \in \mathcal{S}.$$

There is always at least one policy that is better than or equal to all other policies called the optimal policy.

- For optimal policies π_* , their value function would be:

$$v_{\pi_*}(s) = v_*(s) = \max_{\pi} v_{\pi}(s)$$

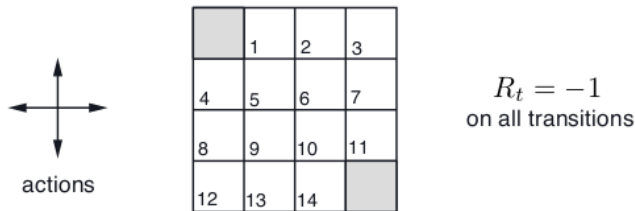
Since the transition probabilities of the MDP are unknown in practice, we use the following form of the Bellman equation for iterations.

$$v(s) \leftarrow \max_{a \in \mathcal{A}} (R(s, a) + \gamma v(s'))$$

Here $R(s, a)$ denotes the reward obtained for taking action a in state s and s' is the subsequent state. So we initially start with all $v(s) = 0$ and go on iterating until convergence to get an optimal policy.

A Toy Example

Consider the 4×4 gridworld shown below:

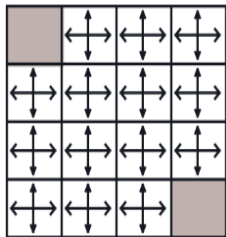


The non-terminal states are $\mathcal{S}_0 = \{1, 2, \dots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{up, down, right, left\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid.

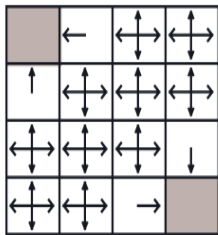
Starting with a random policy, we will try to improve our policy using Bellman equation. We will compute the iterations for $\gamma = 1$ and $\gamma = 0.9$.

$$\gamma = 1$$

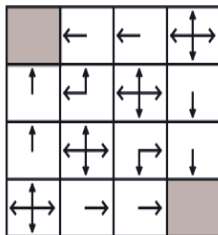
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0



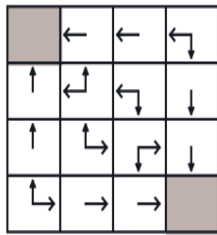
0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0



0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-1
-2	-2	-1	0

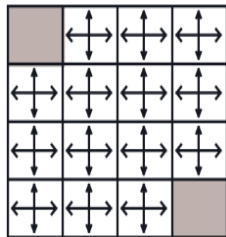


0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0

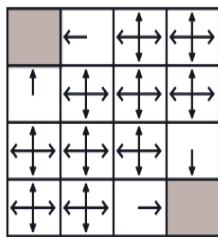


$$\gamma = 0.9$$

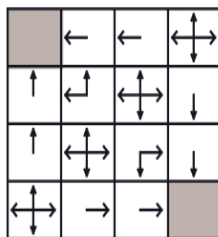
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0



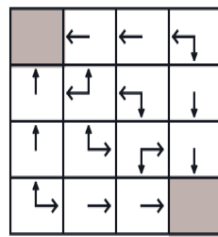
0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0



0	-1	-1.9	-1.9
-1	-1.9	-1.9	-1.9
-1.9	-1.9	-1.9	-1
-1.9	-1.9	-1	0



0	-1	-1.9	-2.7
-1	-1.9	-2.7	-1.9
-1.9	-2.7	-1.9	-1
-2.7	-1.9	-1	0



Introducing TD Learning

- If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning.
- TD methods can learn directly from raw experience without a model of the environment's dynamics.
- TD methods update estimates based in part on other learned estimates, without waiting for a final outcome.
- A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)).$$

Starting from a state, it waits until return to that state giving cumulative reward G_t .

- TD methods need to wait only until the next time step. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)).$$

SARSA

For tackling control problems (finding optimal policy) using TD learning, we mainly have two alternative approaches. The first one is **SARSA** an "On-policy TD Control" method.

- Initialize $Q(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ except $Q(\text{terminal}, *) = 0$.
- For each episode, initialize S . Then choose A from S using policy derived from Q .
- Execute the following in loop:
 1. Take action A and observe R and S' ;
 2. Choose A' from S' using policy derived from Q ;
 3. $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$;
 4. $S \leftarrow S', A \leftarrow A'$;until S is a terminal state.

So, the update step is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] .$$

Q-Learning

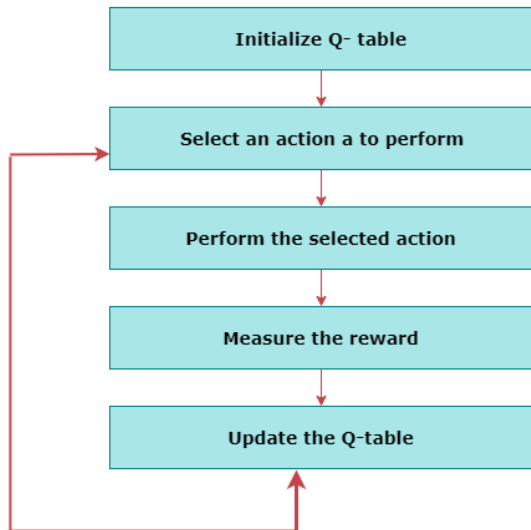
The second one is **Q-learning**: an "Off-policy TD Control" method. It was introduced by Watkins (1989).

- Initialize $Q(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ except $Q(\text{terminal}, *) = 0$.
- For each episode, initialize S .
- Execute the following in loop:
 1. Choose A from S using policy derived from Q ;
 2. Take action A and observe R and S' ;
 3. $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$;
 4. $S \leftarrow S'$;until S is a terminal state.

So, the updation step is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right].$$

Q-learning Flowchart



Next we will see an interesting example demonstrating the Q-learning method.

Convergence of Q-learning Iterations

The following theorem gives a sufficient condition for the convergence of the Q-learning iterations.

Theorem

Given a finite MDP($\mathcal{S}, \mathcal{A}, P, R$), the Q-learning iterations given by

$$Q_{t+1}(S_t, A_t) = Q_t(S_t, A_t) + \alpha_t(S_t, A_t) \left[R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q_t(S_{t+1}, a) - Q_t(S_t, A_t) \right]$$

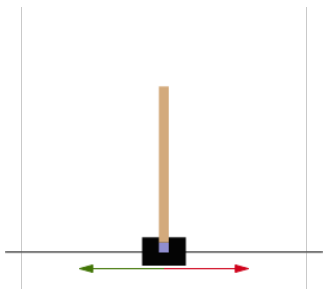
converges with probability 1 to the optimal Q function if

$$\sum_t \alpha_t(s, a) = \infty \quad \& \quad \sum_t \alpha_t^2(s, a) < \infty$$

for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}$.

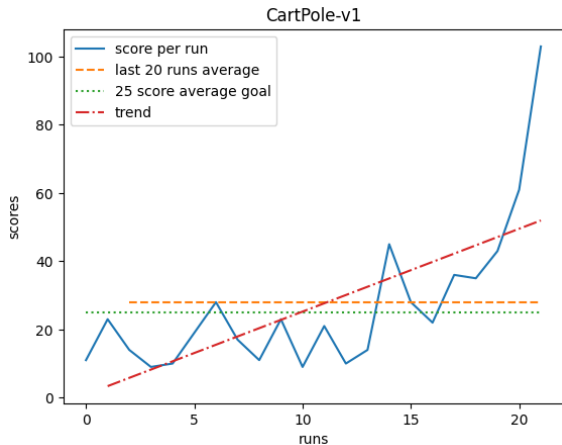
Cartpole: An Application of Q-Learning

- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track.
- The system is controlled by applying a force of $+1$ or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over.
- A reward of $+1$ is provided for every timestep that the pole remains upright.
- The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.



- Violet square indicates a pivot point
- Red and green arrows show possible horizontal forces that can be applied to a pivot point

Cartpole: An Application of Q-Learning (Contd.)



- We fixed discount rate $\gamma = 0.90$ and learning rate $\alpha = 0.001$.
- Here the “score per run” is the cumulative reward until termination.
- From the figure we can see that the score per run gradually increases as we increase the runs. That is intuitive because more number of runs implies better learning of the agent.

Cartpole: An Application of Q-Learning (Contd.)

Before Training

After Training




Concluding Remarks

- ϵ -greedy algorithms are generally better than the greedy strategy. Ideally taking ϵ as small as possible would tend our policy towards the true optimal policy. But decreasing ϵ also increases the time required to get satisfactory policy. So there is a trade-off.
- Markov Decision Process is just a framework to capture the interaction between the agent and the environment. But in this approach, the transition probabilities between states and the distribution of rewards are exactly known. But RL approach does not have any model assumption making it more flexible.
- The concepts of value and value function are key to most of the reinforcement learning methods. The value functions are important for efficient search in the space of policies.

Limitations

- For implementing Q-learning in cartpole problem, we used constant step parameter. But constant step parameter doesn't satisfy the sufficient condition for convergence of the iterations. We were lucky enough to get our iterations converged.
- The reinforcement learning methods we discussed here are structured around estimating value functions but it is not strictly necessary to do this to solve reinforcement learning problems. For example, solution methods such as genetic algorithms, genetic programming, simulated annealing, and other optimization methods never estimate value functions.
- Reinforcement learning needs a lot of data and a lot of computation. It is data-hungry. That is why it works really well in video games because one can play the game again and again and again, so getting lots of data seems feasible.
- To solve many problems of reinforcement learning, we can use a combination of reinforcement learning with other techniques rather than leaving it altogether. One popular combination is Reinforcement learning with Deep Learning.

References

-  Richard S. Sutton, Andrew G. Barto.
Reinforcement Learning - An Introduction, 2nd Edition.
-  Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh.
On the convergence of stochastic iterative dynamic programming algorithms.
Neural Computation, 6(6):1185–1201, 1994.
-  Cart-pole Environment from Open AI
<https://gym.openai.com/envs/CartPole-v1/>

**“Predicting the future isn’t magic,
it’s artificial intelligence.”**

- Dave Waters