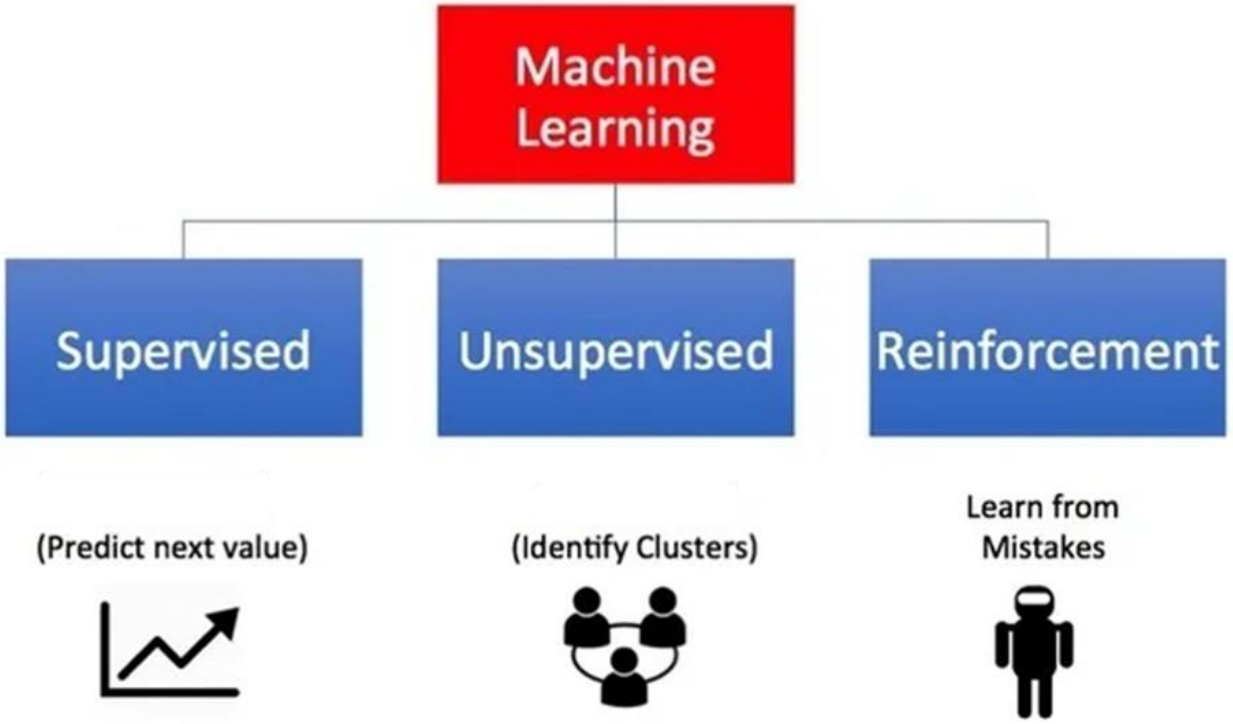
Reinforcement Learning

Types of Machine Learning

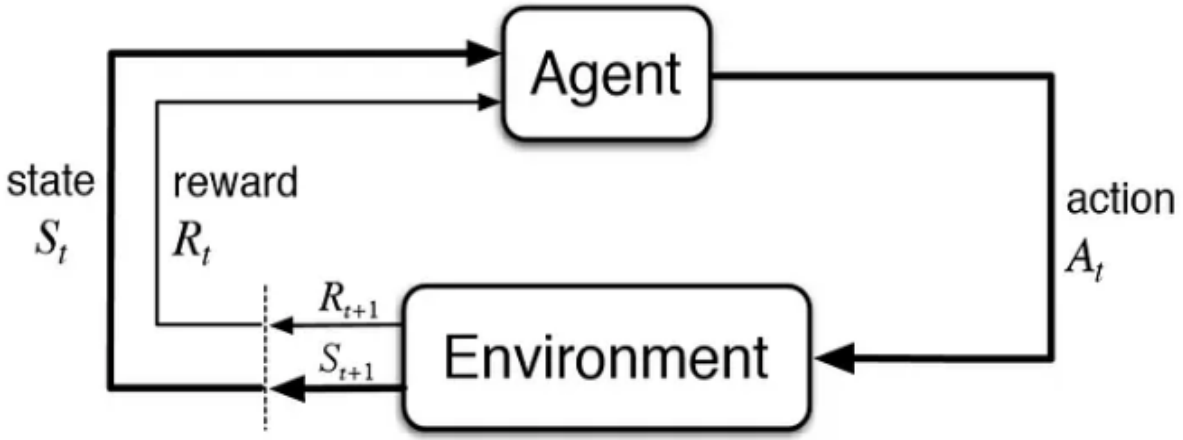


Introduction

- Reinforcement Learning (RL) overcomes the problem of data acquisition.
- In RL, the computer is simply given a goal to achieve.
- The computer then learns how to achieve that goal by trial-and-error by interacting with its environment.
- RL is a feedback-based Machine learning technique in which an agent learns based on this sensory input choosing an action to perform in the environment.
- There is no labeled data, so the agent is bound to learn by its experience only.
- By seeing the results of actions, the agent gets positive feedback for each good action, and for each bad action, the agent gets negative feedback or penalty.

Formulation of a Basic RL Problem

- The action changes the environment in some manner and this change is communicated to the agent through a scalar *reinforcement sig*nal.



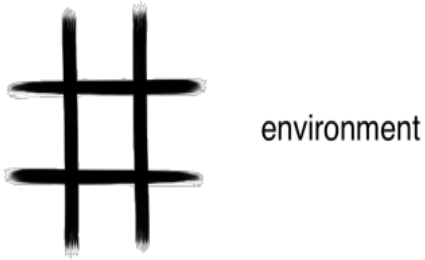
In reinforcement learning the goal is to find a suitable action model that would maximize the **total cumulative reward** of the agent.

Different Key Terms

- **Agent**: This is the algorithm/model that is going to perform the actions and learn over time.
- **Environment**: The surroundings that the agent interacts with.
- **Action**: This is what the agent performs. These are essentially the interactions of the agent in an environment.
- **Reward**: This is the outcome of an action and every action has a reward. A reward could be positive or negative (penalty).
- **State**: The current place of the agent in the environment. The actions that the agent performs can change its state.

The Environment

- Every RL system learns a mapping from situations to actions by trial-and-error interactions with a dynamic environment.
- This environment must at least be partially observable by the reinforcement learning system, and the observations may come in the form of sensor readings.
- The actions may be low level or high level.
- If the RL system can observe perfectly all the information in the environment that might influence the choice of action to perform, then the RL system chooses actions based on true “states” of the environment.
- This ideal case is the best possible basis for reinforcement learning and, in fact, is a necessary condition for much of the associated theory.



- **Policy:** A *policy is a rule determines which action should be performed in each state*; a policy is a mapping from states to actions.

- The policies could be deterministic (maps state to action) or non-deterministic (probability distribution of actions for a state).

- For deterministic policy: $a = \pi(s)$
  For stochastic policy: $\pi(a \mid s) = P[A_t = a \mid S_t = s]$

- Because the policy is essentially the agent's brain, it's not uncommon to substitute the word "policy" for "agent", i.e. saying "The policy is trying to maximize reward."

The Reinforcement Function

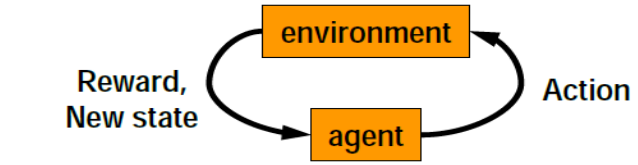
- The “goal” of the RL system is defined using the concept of a *reinforcement function.*
- There exists a mapping from state/action pairs to reinforcements; after performing an action in a given state the RL agent will receive some reinforcement (reward) in the form of a scalar value.
- The RL agent learns to perform actions that will maximize the sum of the reinforcements received when starting from some initial state and proceeding to a terminal state.
- It is the job of the RL system designer to define a reinforcement function that properly defines the goals of the RL agent.

Optimal Reinforcement Function

- Not always the learning agent attempts to maximize the reinforcement function.
- The learning agent could just as easily learn to minimize the reinforcement function due to limited resources and the agent must learn to conserve these resources while achieving a goal.
- An alternative reinforcement function would be used in the context of a game environment, when there are two or more players with opposing goals.
- In a game scenario, the RL system can learn to generate optimal behavior for the players involved by finding the maximin, minimax, or saddlepoint of the reinforcement function.

Formulating Reinforcement Learning

- learn from interaction with environment to achieve a goal



Your action influences the state of the world which determines its reward

- World described by a set of states and actions
- At every time step t, we are in a state s_t, and we:
  - Take an action a_t (possibly null action)
  - Receive some reward r_{t+1}
  - Move into a new state s_{t+1}
- An RL agent may include one or more of these components:
  - Policy π: agent's behaviour function
  - Value function: how good is each state and/or action
  - Model: agent's representation of the environment

- A policy π is a mapping from each state $s \in$ S, and action $a \in A(s)$, to the probability $\pi(s, a)$ while taking action a when in state s.

- The *value* of a state under a policy π, denoted $v^{\pi}(s)$, is the expected return when starting in s and following π thereafter.

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\}$$

Value Function

- The issue of how the agent learns to choose “good” actions, or how we might measure the utility of an action.
- The *value of a state is defined as the sum of the* reinforcements received when starting in that state and following some fixed policy to a terminal state.
- The optimal policy would therefore be the mapping from states to actions that maximizes the sum of the reinforcements when starting in an arbitrary state and performing actions until a terminal state is reached.
- Under this definition the value of a state is dependent upon the policy.
- The *value function is a mapping* from state to state values and can be approximated using any type of function approximator (e.g., multilayered perceptron, memory based system, radial basis functions, look-up table, etc.).
- The value function can guide the agent in selecting the optimum action.

Value Function

- Value function is the expected future reward
- Used to evaluate the goodness/badness of states
- Our aim will be to maximize the value function (the total reward we receive over time): find the policy with the highest expected reward
- By following a policy π, the value function is defined as:

$$V^{\pi}(s_t) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots]$$

- γ is called a discount rate, and it is always $0 \leq \gamma \leq 1$
- If γ close to 1, rewards further in the future count more, and we say that the agent is "farsighted"
- γ is less than 1 because there is usually a time limit to the sequence of actions needed to solve a task (we prefer rewards sooner rather than later)

- Initially, the approximation of the optimal value function is poor.
- *$V^*(x_t)$ is the optimal value function where x_t is the state vector; $V(x_t)$ is the approximation of the value* function; γ is a discount factor in the range [0,1] that causes immediate reinforcement to have more importance.
- *$V(x_t)$ will be initialized to random values and will contain no information about the optimal* value function $V^*(x_t)$.
- *This means that the approximation of the optimal value function in a given state is* equal to the true value of that state $V^*(x_t)$ *plus some error in the approximation, as expressed*

$$V(\mathbf{x}_t) = e(\mathbf{x}_t) + V^*(\mathbf{x}_t)$$

where *$e(x_t)$ is the error in the approximation of the value of the state occupied at time t. Likewise*

$$V(\mathbf{x}_{t+1}) = e(\mathbf{x}_{t+1}) + V^*(\mathbf{x}_{t+1})$$

- The value of state x_t for the optimal policy is the sum of the reinforcements when starting from state x_t and performing optimal actions until a terminal state is reached.

- By this definition, a simple relationship exists between the values of successive states, x_t and x_{t+1} and defined by the Bellman equation as

$$V^*(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma V^*(\mathbf{x}_{t+1})$$

where the discount factor γ is used to exponentially decrease the weight of reinforcements received in the future.

- The approximation *$V(x_t)$ also has the same relationship,*

$$V(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma V(\mathbf{x}_{t+1})$$

$$e(\mathbf{x}_t) + V^*(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma\left(e(\mathbf{x}_{t+1}) + V^*(\mathbf{x}_{t+1})\right)$$

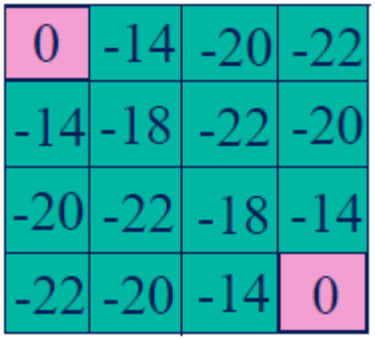
$$e(\mathbf{x}_t) + V^*(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma e(\mathbf{x}_{t+1}) + \gamma V^*(\mathbf{x}_{t+1})$$

- *$V^*(x_t)$ is subtracted from both sides to reveal the relationship in the* errors of successive states. This relationship is expressed $e(\mathbf{x}_t) = \gamma e(\mathbf{x}_{t+1})$

Approximating the Value Function

- Reinforcement learning is a difficult problem because the learning system may perform an action and not be told whether that action was good or bad.
- It will have to make many decisions and then acting on such decisions and the system learn from this experience.
- In the future, any time it chooses an action that leads to this particular situation, it will immediately learn that particular action is bad or good.
- The primary objective of learning is to find the correct mapping. Once this is completed, the optimal policy can easily be extracted.

Numbers represent value $V^{\pi}(s)$ of each state s

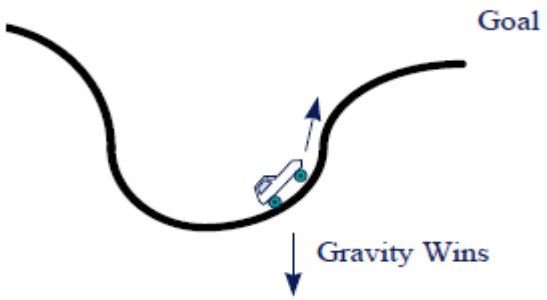


- The state space can be visualized using a 4x4 grid.
- Each square represents a state.
- The reinforcement function (Reward) is -1 (i.e., the agent receives a reinforcement of -1 on each transition).
- There are 4 actions possible in each state: north, south, east, west.
- The goal states are the upper left corner and the lower right corner.
- For each state the random policy randomly chooses one of the four possible actions.
- The numbers in the states represent the expected values of the states.
- For example, when starting in the lower left corner and following a random policy, on average there will be 22 transitions to other states before the terminal state is reached.

Reward and Avoidance Problems

- In the Pure Delayed Reward class of functions the reinforcements are all zero except at the terminal state.
- The sign of the scalar reinforcement at the terminal state indicates whether the terminal state is a goal state (a reward) or a state that should be avoided (a penalty).
- Reinforcement functions in minimum time to goal cause an agent to perform actions that generate the shortest path or trajectory to a goal state.
- The goal of the driver (RL agent)

is to successfully drive up the incline on the right to reach a goal state at the top of the hill.



"Car on the hill" problem

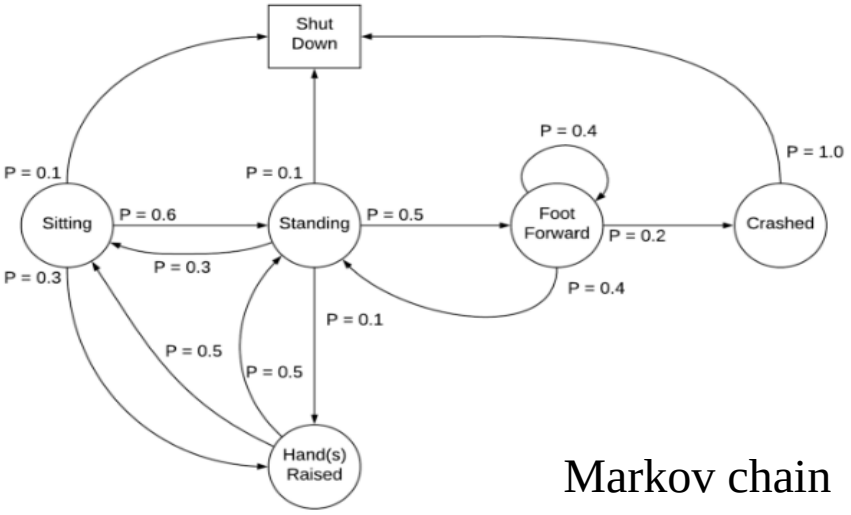
- The state of the environment is the car's position and velocity.
- Three actions are available to the agent in each state: forward thrust, backward thrust, or no thrust at all.
- The dynamics of the system are such that the car does not have enough thrust to simply drive up the hill.
- Rather, the driver must learn to use momentum to his advantage to gain enough velocity to successfully climb the hill.
- The reinforcement function is -1 for ALL state transitions except the transition to the goal state, in which case a zero reinforcement is returned.
- Because the agent wishes to maximize reinforcement, it learns to choose actions that minimize the time it takes to reach the goal state, and in so doing learns the optimal strategy for driving the car up the hill.

Markov Decision Process

- The Markov decision process (MDP) is a mathematical framework used for modeling decision-making problems where the outcomes are partly random and partly controllable.
- It's a framework that can address most reinforcement learning (RL) problems.
- According to the Markov property, the current state of the robot depends only on its immediate previous state (or the previous time step).
- Formally, for a state S_t to be Markov, the probability of the next state $S_{(t+1)}$ being s' should only be dependent on the current state S_t

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

- A Markov process is defined by (S, P) where S are the states, and *P* is the state-transition probability.
- It consists of a sequence of random states S_1, S_2, ... where all the states obey the Markov property.
- The state transition probability or $P_{ss'}$ is the probability of jumping to a state *s'* from the current state *s*.

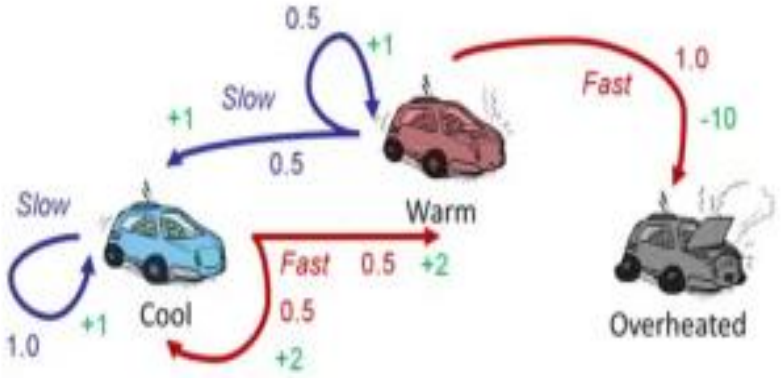


Markov chain

- The model describes the environment by a distribution over rewards and state transitions:

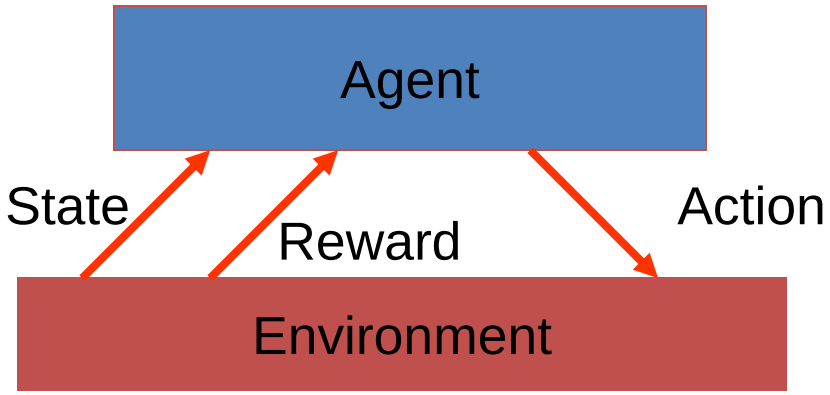
$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

- We assume the Markov property: the future depends on the past only through the current state



Review of MDP model

- MDP model <S,T,A,R>



$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r1]{a1} s2 \xrightarrow[r2]{a2} s3$

$$V(s) = \max [R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s')]$$

- S– set of states
- A– set of actions
- T(s,a,s') = P(s'|s,a)– the probability of transition from *s* to *s'* given action *a*
- R(s,a)– the expected reward for taking action *a* in state *s*

$$R(s,a) = \sum_{s'} P(s'|s,a) r(s,a,s')$$

$$R(s,a) = \sum_{s'} T(s,a,s') r(s,a,s')$$

- Consider the game tic-tac-toe:
  - reward: win/lose/tie the game $(+1/-1/0)$ [only at final move in given game]
  - state: positions of X's and O's on the board
  - policy: mapping from states to actions
    - based on rules of game: choice of one open position
  - value function: prediction of reward in future, based on current state
- In tic-tac-toe, since state space is tractable, can use a table to represent value function
  - Each board position (taking into account symmetry) has some probability

State	Probability of a win (Computer plays "o")
	0.5
	0.5
	1.0
	0.0
	0.5
etc	

  - Simple learning process:
    - start with all values = 0.5
    - policy: choose move with highest probability of winning given current legal moves from current state
    - update entries in table based on outcome of each game
    - After many games value function will represent true probability of winning from each state
  - Can try alternative policy: sometimes select moves randomly (exploration)

Challenges

- The outcome of your actions may be uncertain
- You may not be able to perfectly sense the state of the world
- The reward may be stochastic
- You may have no clue (model) about how the world responds to your actions
- You may have no clue (model) of how rewards are being paid off
- The world may change while you try to learn it

Approaches to implement RL

- **Value-based:**
  The value-based approach is about to find the optimal value function, which is the maximum at a state under any policy. Therefore, the agent expects the long-term return at any state(s) under policy π.

- **Policy-based:**
  Policy-based approach is to find the optimal policy for the maximum future rewards without using the value function. In this approach, the agent tries to apply such a policy that the action performed in each step helps to maximize the future reward.
  The policy-based approach has mainly two types of policy:

  - **Deterministic:** The same action is produced by the policy (π) at any state.
  - **Stochastic:** In this policy, probability determines the produced action.

Model based vs.Model free approaches

Model-based: In the model-based approach, a virtual model is created for the environment, and the agent explores that environment to learn it. There is no particular solution or algorithm for this approach because the model representation is different for each environment.

Adaptive dynamic learning(ADP) approach

– *Model free approach RL*:

derive the optimal policy without learning the model.

- Which one is better?

Utility Value

- The agent knows what state it is in
- The agent has a number of actions it can perform in each state.
- Initially, it doesn't know the value of any of the states
- If the outcome of performing an action at a state is deterministic, then the agent can update the utility value U() of states:

  U(oldstate) = reward + U(newstate)

- The agent learns the utility values of states as it works its way through the state space

Discount factor

- The agent may occasionally choose to explore suboptimal moves in the hopes of finding better outcomes
  - Only by visiting all the states frequently enough can we guarantee learning the true values of all the states.
- A discount factor is often introduced to prevent utility values from diverging and to promote the use of shorter (more efficient) sequences of actions to attain rewards
- The update equation using a discount factor γ is:

  U(oldstate) = reward + γ * U(newstate)
- Normally, γ is set between 0 and 1

The Task

- To learn an optimal *policy* that maps states of the world to actions of the agent.
  - For example: *If this patch of room is dirty, I clean it. If my battery is empty, I recharge it.*

$$\pi : S \rightarrow A$$

- What is it that the agent tries to optimize?

$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

Answer: The total future discounted reward:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad 0 \leq \gamma < 1$$

Note that immediate reward is worth more than future reward.

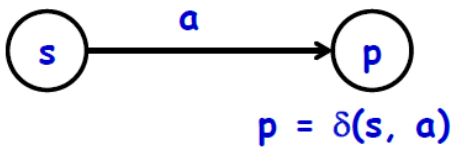
$V^{\pi}(s_t)$ is the optimal value function where s_t is the state vector; γ is a discount factor in the range [0,1] that causes immediate reinforcement to have more importance (weighted more heavily) than future reinforcement.

Value Function

- Suppose we have access to the optimal value function that computes the total future discounted reward **V*(s)**
- What would be the optimal policy **π*(s)** ?

Answer: we choose the action that maximizes:

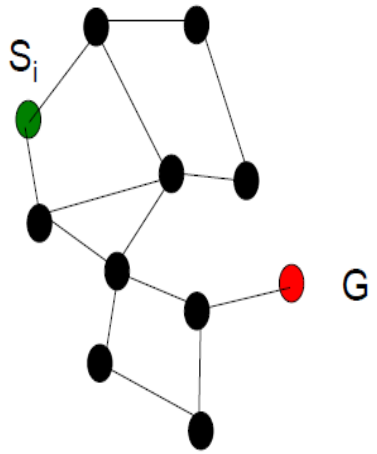
$$\pi^*(s) = \arg\max_a \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$$



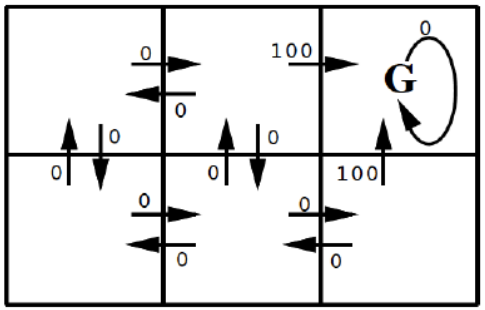
- We assume that we know what the reward will be if we perform action "**a**" in state "**s**":
  r(s,a)
- We also assume we know what the next state of the world will be if we perform action "**a**" in state "**s**":
  $s_{t+1} = \delta(s_t, a)$

Example 1

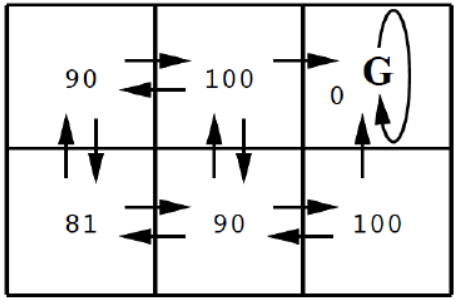
- Consider some complicated graph, and we would like to find the shortest path from a node S_i to a goal node G.
- Traversing an edge will incur costs – this is the edge weight.
- The value function encodes the total remaining distance to the goal node from any node s, that is: V(s) = "*1 / distance*" to goal from s.
- If you know V(s), the problem is trivial. You simply choose the node that has highest V(s).



Example 2



r(s,a) : immediate reward values

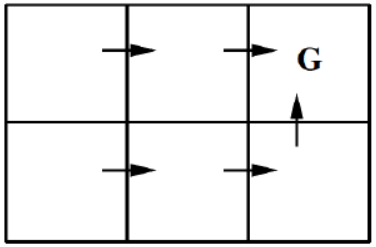


V*(s) : values

$\gamma = 0.9$

Therefore for the bottom left square:

$0 + (0.9 \times 0) + (0.9^2 \times 100) = 81$



One optimal policy

$$\pi^*(s) = \arg\max_a \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$$

- A deterministic Markov decision process is one in which the state transitions are deterministic (an action performed in state x_t always transitions to the same successor state x_{t+1}).
- Alternatively, in a nondeterministic Markov decision process, a probability distribution function defines a set of potential successor states for a given action in a given state.
- If the MDP is non-deterministic, then value iteration requires that we find the action that returns the maximum expected value.
- For example, to find the expected value of the successor state associated with a given action, one must perform that action an infinite number of times, taking the integral over the values of all possible successor states for that action.
- Theoretically, value iteration is possible in the context of non-deterministic MDPs, however, in practice it is computationally impossible to calculate the necessary integrals without added knowledge or some degree of modification.
- **Q-learning solves the problem of having to take the max over a set of integrals.**

- Rather than finding a mapping from states to state values (as in value iteration), Q-learning finds a mapping from state/action pairs to values (called Q-values).
- Instead of having an associated value function, Q learning makes use of the Q-function.
- In each state, there is a Q-value associated with each action.
- The definition of a Q-value is the sum of the (possibly discounted) reinforcements received when performing the associated action and then following the given policy thereafter.
- Likewise, the definition of an optimal Q value is the sum of the reinforcements received when performing the associated action and then following the optimal policy thereafter.

Off policy RL algorithm

- Q-learning is an **Off policy RL algorithm**, which is used for the temporal difference Learning.
- The temporal difference learning methods are the way of comparing temporally successive predictions.
- Q-learning is a kind of Reinforcement learning algorithm that enables machines to discover the best behaviors for performing in a given environment via a trial-and-error method.
- The main objective of Q-learning is to learn the policy which can inform the agent that what actions should be taken for maximizing the reward under what circumstances.
- The quality value, also known as the Q-value, is an estimate of the expected reward for doing a certain action in a specific condition and is the “Q” in Q-learning.

Q-Values

- Finding the best course of action that accelerates the long-term benefit is the aim of Q-learning.
- Starting with a database of Q-values for each state-action combination, the Q-learning algorithm operates.
- These parameters are initially set at random or to zero. The agent then investigates the surroundings, acting and earning rewards.
- A mathematical formula that considers the present Q-value, the reward received, and the anticipated value of the following state-action combination is used to update the Q-values based on these rewards.
- The Q-values, which reflect the ideal actions to perform in each state as the agent continues to investigate the environment, converge to their optimal values.

- The value function $V^{\pi}(s)$ assigns each state the expected reward

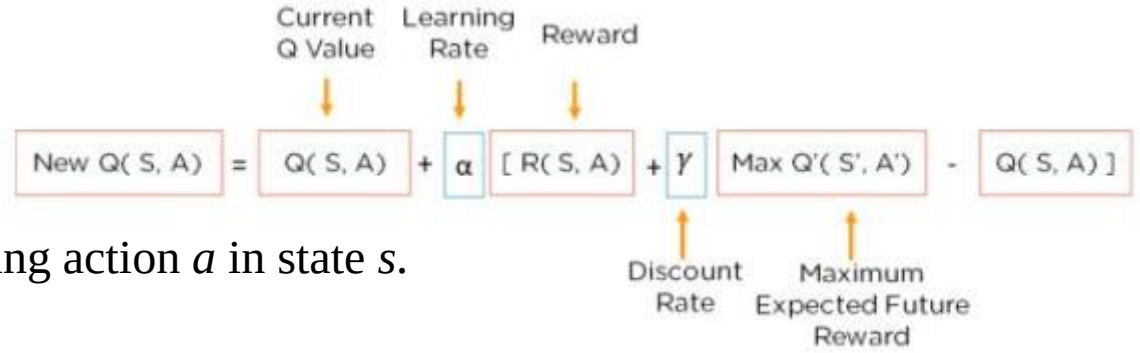
$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a_t, a_{t+i}, s_{t+i}} \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s \right]$$

- Usually not informative enough to make decisions.
- The Q-value $Q^{\pi}(s, a)$ is the expected reward of taking action a in state s and then continuing according to π.

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{a_{t+i}, s_{t+i}} \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s, a_t = a \right]$$

The Q Function

The Bellman Equation is used to determine the value of a particular state and deduce how good it is to be in/take that state.



Bellman Equation

Q(s,a) = expected reward for taking action *a* in state *s*.

R – Actual reward received for that action
while *s'* refers to the next state.

The highest expected reward for all possible
actions *a'* in state *s'* is represented *by max(Q(s', a'))*.

It uses the current state, and the reward associated with that state, along with the maximum expected reward and a discount rate, which determines its importance to the current state, to find the next state of our agent.

The learning rate determines how fast or slow, the model will be learning.

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

In the beginning, the agent has no idea about the environment.

He is more likely to explore new things than to exploit his knowledge because…he has no knowledge.

Through time steps, the agent will get more and more information about how the environment works and then, the agent is more likely to exploit knowledge than exploring new things.

If we skip this important step, the Q-Value function will converge to a local minimum which in most of the time, is far from the optimal Q-value function.

To handle this, we will have a threshold which will decay every episode using exponential decay formula.

Exploration Exploitation Dilemma

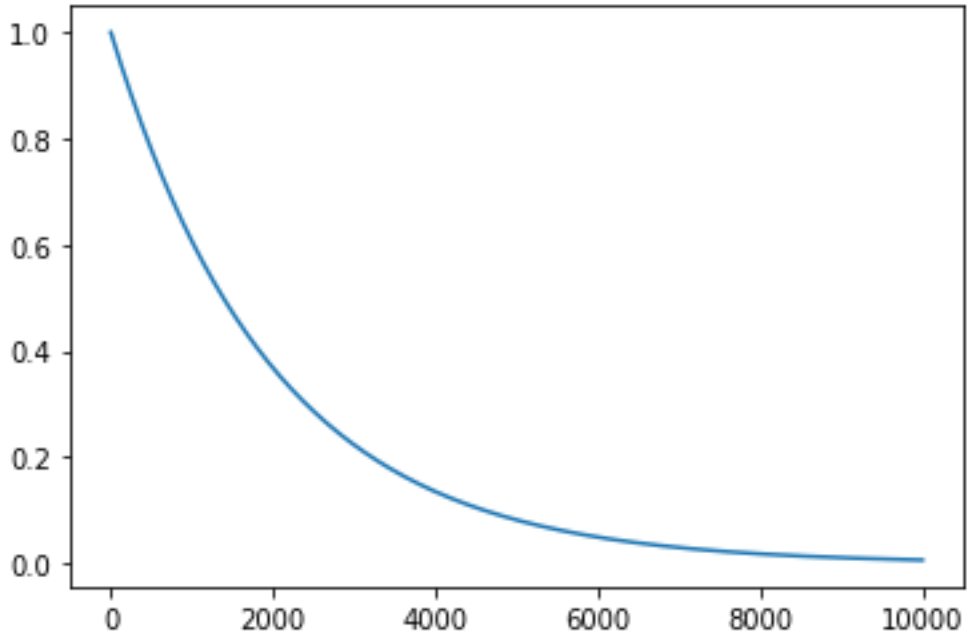
By doing that, at every time step ***t***, we will sample a variable uniformly over [0,1]. If the variable is smaller than the threshold, the agent will explore the environment. Otherwise, agent will exploit his knowledge.

$$N(t) = N_0 e^{-\lambda t},$$

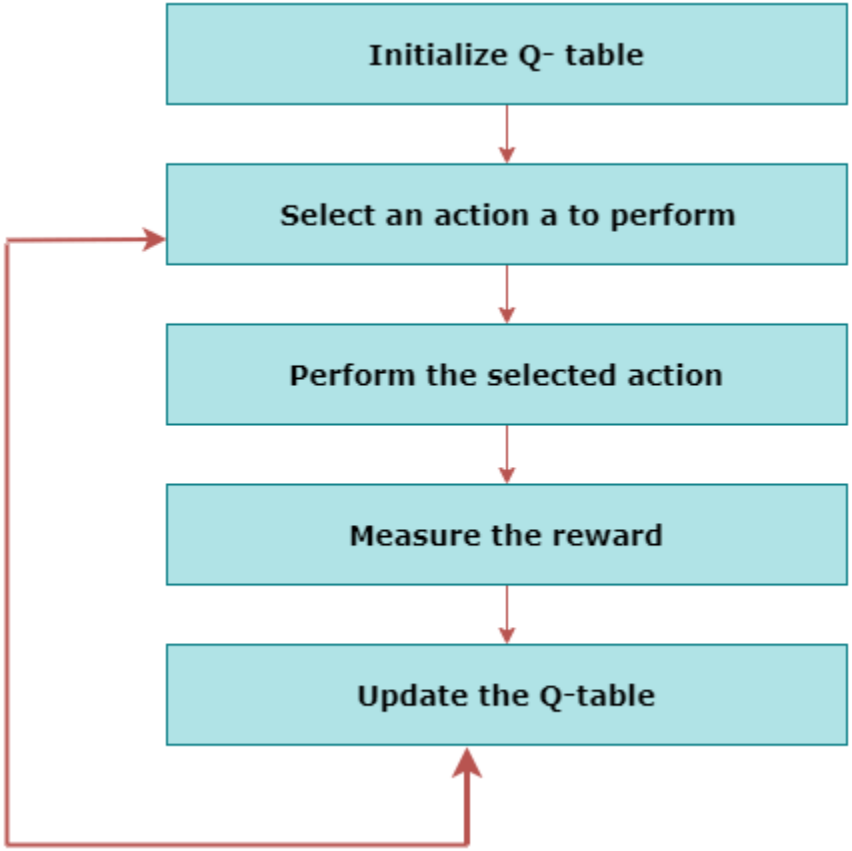
where N_0 is the initial value and $\boldsymbol{\lambda}$**,** a constant called *decay constant.*

- Exponential decay graph with

$N_0 = 1$ and $\lambda = 0.0005$



In the context of Q-learning, the value of a state is defined to be the maximum Q-value in the given state.

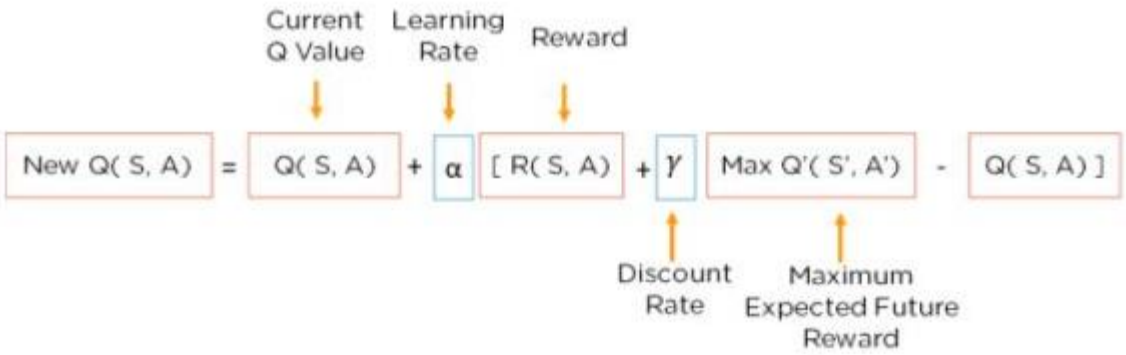


Q-table

- A Q-table or matrix is created while performing the Q-learning.
- The table follows the state and action pair, i.e., [s, a], and initializes the values to zero.
- After each action, the table is updated, and the q-values are stored within the table.
- The RL agent uses this Q-table as a reference table to select the best action based on the q-values.

The Q Function

The Bellman Equation is used to determine the value of a particular state and deduce how good it is to be in/take that state.



Bellman Equation

It uses the current state, and the reward associated with that state, along with the maximum expected reward and a discount rate, which determines its importance to the current state, to find the next state of our agent.

The learning rate determines how fast or slow, the model will be learning.

- Step 1: Create an initial Q-Table with all values initialized to 0.

Action	Fetching	Sitting	Running
Start	0	0	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	0	0
End	0	0	0

Step 2: Choose an action (sit) and perform it. Update values in the table

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	0	0
End	0	0	0

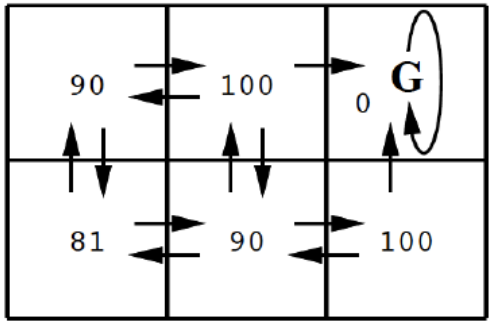
- Step 3: Get the value of the reward and calculate the value Q-Value using Bellman Equation.
- For the action performed, we need to calculate the value of the actual reward and the Q(S, A) v.alue

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	34	0
End	0	0	0

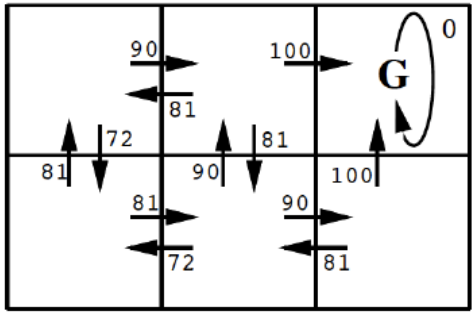
Step 4: Continue the same until the table is filled or an episode ends

Action	Fetching	Sitting	Running
Start	5	7	10
Idle	2	5	3
Wrong Action	2	6	1
Correct Action	54	34	17
End	3	1	4

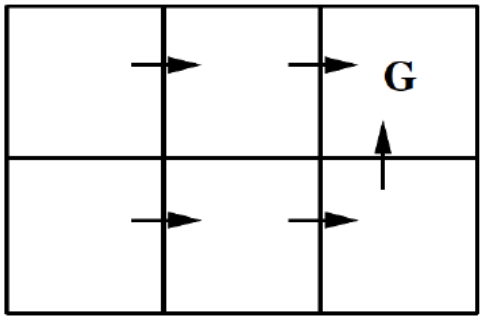
Link between value and policy. Optimal value requires optimal policy



V*(s) : values



Q(s,a) : values



One optimal policy

Note that:

$$\pi^*(s) = \arg\max_a Q(s,a)$$

$$V^*(s) = \max_a Q(s,a)$$

Q Learning

$$\pi^*(s) = \arg\max_a Q(s,a)$$

$$V^*(s) = \max_a Q(s,a)$$

Now:

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$
$$= r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$

This still depends on $\mathbf{r(s,a)}$ and $\delta\mathbf{(s,a)}$

Imagine the robot is exploring its environment, trying new actions as it goes.

- At every step it receives some reward "**r**", and it observes the environment change into a new state **s′** for action **a**.
- How can we use these observations, $\mathbf{(s,a,s',r)}$ to learn a model?

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

where: $\mathbf{s'} = \delta\mathbf{(s,a)}$

Q Learning

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$ where: $s' = \delta(s,a)$

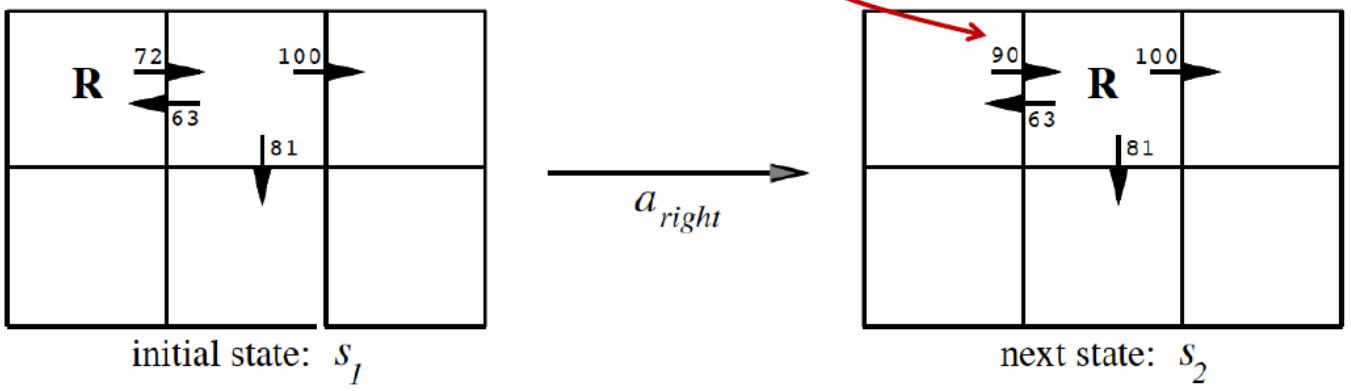
- This equation continually estimates Q at state **s** consistent with an estimate of Q at state **s′**, one step in the future.
- Note that **s′** is closer to the goal, and hence more "reliable", but still an estimate itself.
- Updating estimates based on other heuristics is called bootstrapping.
- We do an update after each state-action pair, that is, we are learning online!
- We are learning useful things about explored state-action pairs. These are typically most useful because they are likely to be encountered again.
- Under suitable conditions, these updates can actually be proved to converge to the real answer.

One Step of Q-Learning

$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$

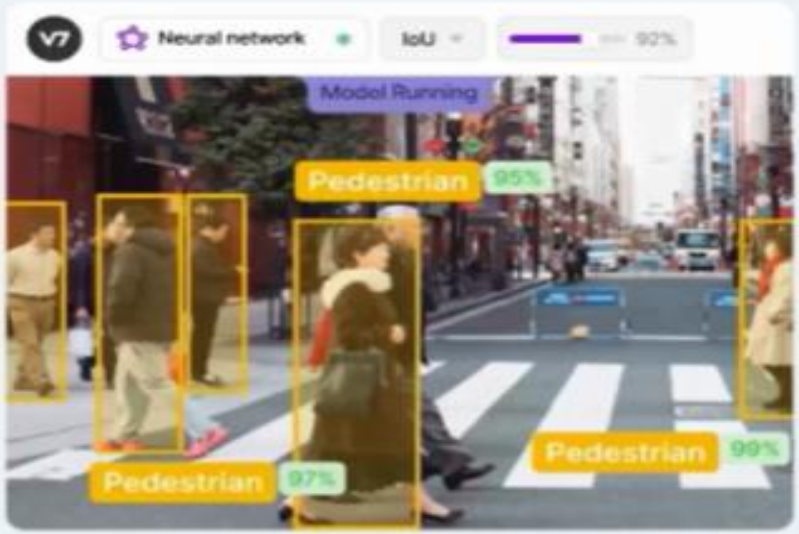
$$\leftarrow 0 + 0.9 \max\{63, 81, 100\}$$

$$\leftarrow 90$$

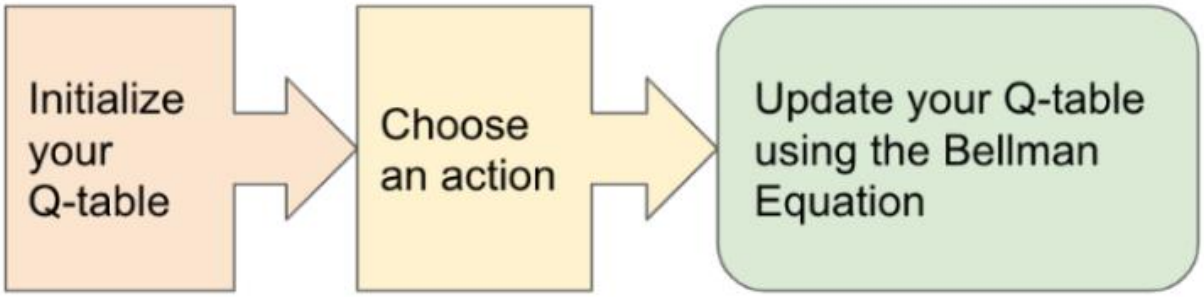


Q-learning propagates Q-estimates 1-step backwards

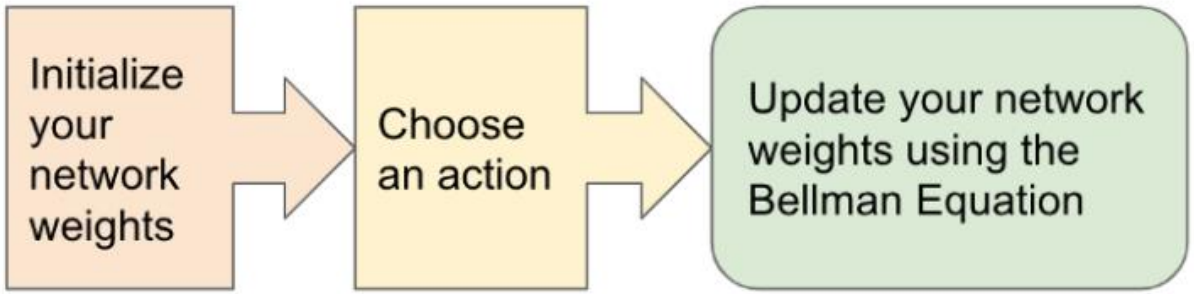
Label, train, and test computer vision models to **solve any task**



Deep-Q Network

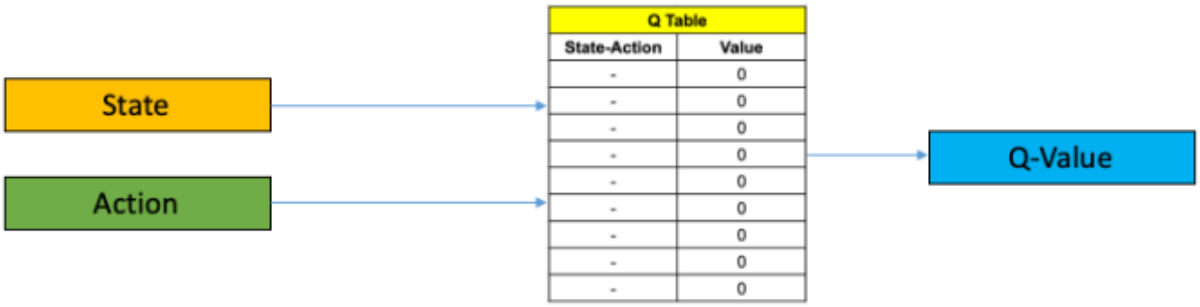


In the basic Q-Learning approach, we need to maintain a look-up table called q-map for each state-action pair and the corresponding value associated with it.

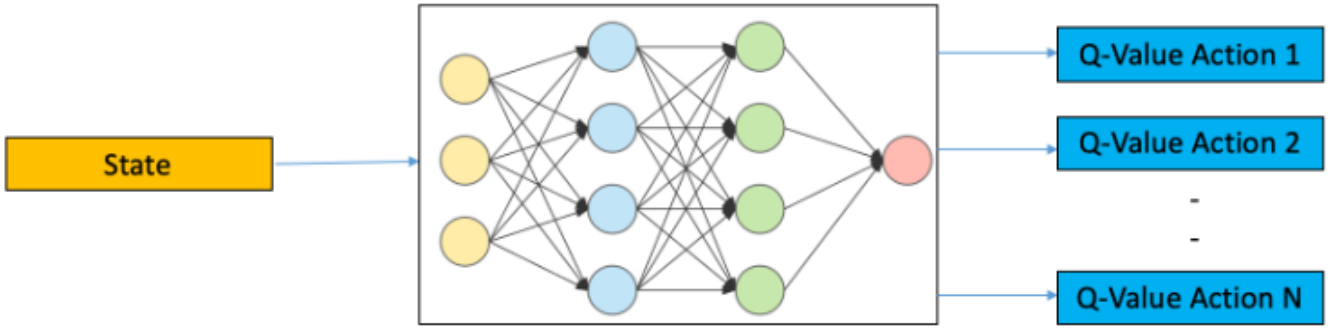


Deep Q-network employs Neural Network architecture to predict the Q-value for a given state.

- Reinforcement Learning involves managing state-action pairs and keeping a track of value (reward) attached to an action to determine the optimum policy.
- This method of maintaining a state-action-value table is not possible in real-life scenarios when there are a larger number of possibilities.
- Instead of utilizing a table, we can make use of Neural Networks to predict values for actions in a given state.



Q Learning



Deep Q Learning

- In deep RL, we deal with **parameterized policies**: policies whose outputs are computable functions that depend on a set of parameters (eg the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm.
- We often denote the parameters of such a policy by or , and then write this as a subscript on the policy symbol to highlight the connection:

$$a_t = \mu_\theta(s_t)$$
$$a_t \sim \pi_\theta(\cdot|s_t).$$

- It's difficult to come up with a perfect heuristic.
- Improving the heuristic generally entails playing the game many times, to determine specific cases where the agent could have made better choices.
- And, it can prove challenging to interpret what exactly is going wrong, and ultimately to fix old mistakes without accidentally introducing new ones.
- It would be much easier if we had a more systematic way of improving the agent with game play experience.