Artificial Intelligence
DSE 3252

Reinforcement Learning -2

ROHINI R RAO
DEPT OF DATA SCIENCE & COMPUTER APPLICATIONS

JANUARY 2023

Monte Carlo Methods

The term “Monte Carlo” is used for any estimation method whose operation involves a significant random component.

In Reinforcement Learning it is used for methods based on averaging complete returns

- Does not require complete knowledge of the environment
- Does not require prior knowledge of the environment’s dynamics

Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP)

Monte Carlo methods require only experience

- Sample sequences of states, actions, and rewards from actual or simulated interaction with an environment.
- Learning from actual experience, yet can still attain optimal behavior.

Monte Carlo Prediction

Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense

Episodic Tasks :

- To ensure that well-defined returns are available, we assume experience is divided into episodes
- all episodes eventually terminate no matter what actions are selected
- Only on the completion of an episode are value estimates and policies changed

MC Methods

- suppose we wish to estimate $v_{\Pi}(s)$, the value of a state s under policy Π,
- given a set of episodes obtained by following Πand passing through state s.
- Each occurrence of state s in an episode is called a visit to state s.
- States may be visited multiple times in the same episode
- The first time it is visited in an episode is called the first visit to state s.
- **Types of MC methods** are
  - First-visit MC method estimates $v_{\Pi}(s)$, as the average of the returns following first visits to s
  - Every-visit MC method averages the returns following all visits to s.

Monte Carlo Policy Evaluation

Return is the total discounted reward:

Trajectory in episode is

$$S1, A1, R2, \ldots.Sk \sim$$

The value function is the expected return

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s]$$

Estimate any expected value simply by adding up samples and dividing by the total number of samples

- i – Episode index
- s – Index of state

$$\bar{V}_\pi(s) = \frac{1}{N} \sum_{i=1}^{N} G_{i,s}$$

First Visit Monte Carlo

Average returns only for the first time s is visited in an episode

1. Initialize the policy, state-value function
2. Start by generating an episode according to the current policy
    1. Keep track of the states encountered through that episode
3. Select a state in 2.1
    1. Add to a list the return received after first occurrence of this state
    2. Average over all returns
    3. Set the value of the state as that computed average
4. Repeat step 3
5. Repeat 2-4 until satisfied

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Every visit Monte Carlo

1. Initialize the policy, state-value function
2. Start by generating an episode according to the current policy
    1. Keep track of the states encountered through that episode
3. Select a state in 2.1
    1. Add to a list the return received after every occurrence of this state.
    2. Average over all returns
    3. Set the value of the state as that computed average
4. Repeat step 3
5. Repeat 2-4 until satisfied

Computation of Return

$$A+3 \rightarrow A+2 \rightarrow B-4 \rightarrow A+4 \rightarrow B-3 \rightarrow \text{terminate}$$

$$B-2 \rightarrow A+3 \rightarrow B-3 \rightarrow \text{terminate}$$

First visit	Every visit
V(A) = 1/2(2 + 0) = 1 V(B) = 1/2(-3 + -2) = -5/2	V(A) = 1/4(2 + -1 + 1 + 0) = 1/2 V(B) = 1/4(-3 + -3 + -2 + -3) = -11/4

Incremental Mean & Policy Improvement

We update v(s) incrementally after episodes. For each state St, with return Gt:

$$N(S_t) \leftarrow N(S_t) + 1$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

In non-stationary problems, it can be useful to track a running mean, i.e., forget old episodes:

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

Policy improvement is done by making the policy greedy with respect to the current value function.

$$\pi(s) \doteq \arg\max_a q(s,a)$$

Monte Carlo Estimation of Action Values

Dynamic Programming (With a model)

- state values alone are sufficient to determine a policy;
- looks ahead one step and chooses whichever action leads to the best combination of reward and next state

Monte Carlo Methods (Without a model)

- however, state values alone are not sufficient
- One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy.

Thus, primary goal for Monte Carlo methods is to estimate q_*.

To achieve consider the policy evaluation problem for action values.

Policy Evaluation Phase

The policy evaluation problem for action values

- is to estimate $q_{\Pi}(s, a)$, the expected return when starting in state s, taking action a, following policy Π.

Issues include

- Many state–action pairs may never be visited.
- If Π is a deterministic policy, then in following Π one will observe returns only for one of the actions from each state.
- With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience.

For policy evaluation to work for action values, **continual exploratio**n is required.

Exploring Starts Assumption

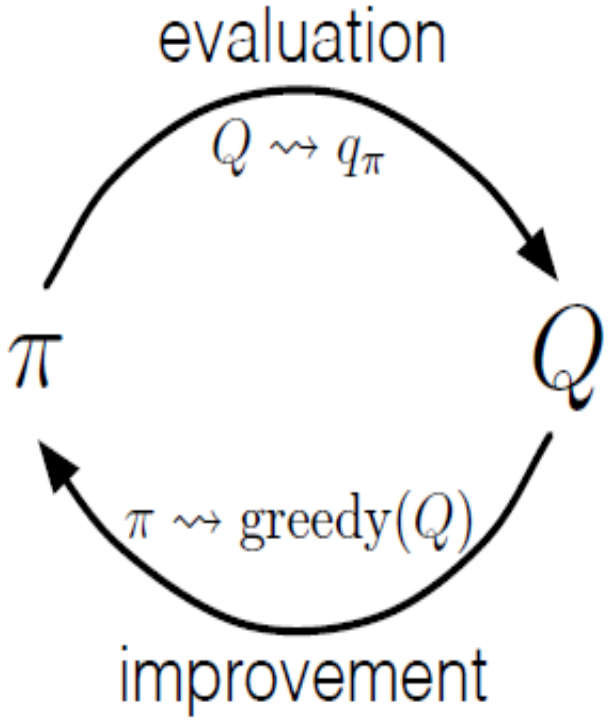
- Specify that the episodes start in a state–action pair, and that every pair has a nonzero probability of being selected as the start.
- This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes.

Monte Carlo Control

this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

where $\xrightarrow{\text{E}}$ denotes a complete policy evaluation and $\xrightarrow{\text{I}}$ denotes a complete policy



Generalised Policy Iteration (GPI)

Monte Carlo Control

- Policy Improvement is a greedy policy with respect to current value function, which is action-value function and therefore no model is required to construct the greedy policy
- For any action-value function q, the corresponding greedy policy is the one that for each state s, deterministically chooses an action with maximal action value:

$$\pi(s) \doteq \arg\max_a q(s,a).$$

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s,a)) \\ &= \max_a q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

On Policy vs Off Policy

On-policy method

- attempt to evaluate or improve the same Policy that is used to make decisions.
- For Example:
- *Trying different restaurants in an area to search for the best restaurant.*
- *The agent evaluates the decision as well as improve the decision*

Off-policy method

- uses a behavioral policy to explore the environment and to collect samples generating Agent's behavior
- and a second policy being learned called the target policy which is optimized.
- For Example:
- *Follow Google's recommendations to search for the best restaurant in the area (Behavioral policy)*
- *Agent decides restaurant (Target policy)*

On-Policy can be used for model-based and model-free reinforcement learning

Off-policy is used for model-free reinforcement learning algorithms

Monte Carlo Control with Epsilon-soft

On-policy methods

- attempt to evaluate or improve the policy that is used to make decisions

In on-policy control methods the policy is generally soft

- $\Pi(a|s) > 0$ for all $s \in S$ and all $a \in A(s)$
- but gradually shifted closer and closer to a deterministic optimal policy

ε- greedy policies

- most of the time they choose an action that has maximal estimated action value
- but with probability ε they instead select an action at random

Monte Carlo policy iteration

- alternate between evaluation and improvement on an episode-by-episode basis
- After each episode, the observed returns are used for policy evaluation
- and then the policy is improved at all the states visited in the episode

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

//Non-greedyaction with the minimal probability of selection

Policy Improvement

That any ε-greedy policy with respect to q_π is an improvement over any ε-soft policy π is assured by the policy improvement theorem. Let π' be the ε-greedy policy. The conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a) \qquad (5.2) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon} q_\pi(s,a) \end{aligned}$$

(the sum is a weighted average with nonnegative weights summing to 1, and as such it must be less than or equal to the largest number averaged)

$$\begin{aligned} &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a) \\ &= v_\pi(s). \end{aligned}$$

Temporal Difference Learning

TD Methods

- can learn directly from raw experience without a model of the environment's dynamics
- update estimates based in part on other learned estimates, without waiting for a final outcome (bootstrap)

A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha \Big[G_t - V(S_t) \Big]$$

The simplest TD method makes the update

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

immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called *TD(0)*, or *one-step TD*, because it is a special case of the TD(λ) and

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0, 1]$

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\textit{terminal}) = 0$

Loop for each episode:

  Initialize S

  Loop for each step of episode:

    $A \leftarrow$ action given by π for S

    Take action A, observe R, S'

    $V(S) \leftarrow V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$

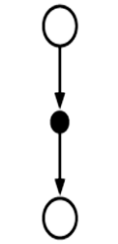
    $S \leftarrow S'$

  until S is terminal

TD combines DP and MC methods

TD combines the sampling of MC with Bootstrapping of DP

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] .
\end{aligned}
$$



TD(0)

Sample updates

- because they involve looking ahead to a sample successor state (or state–action pair)
- Differ from expected updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

TD Error

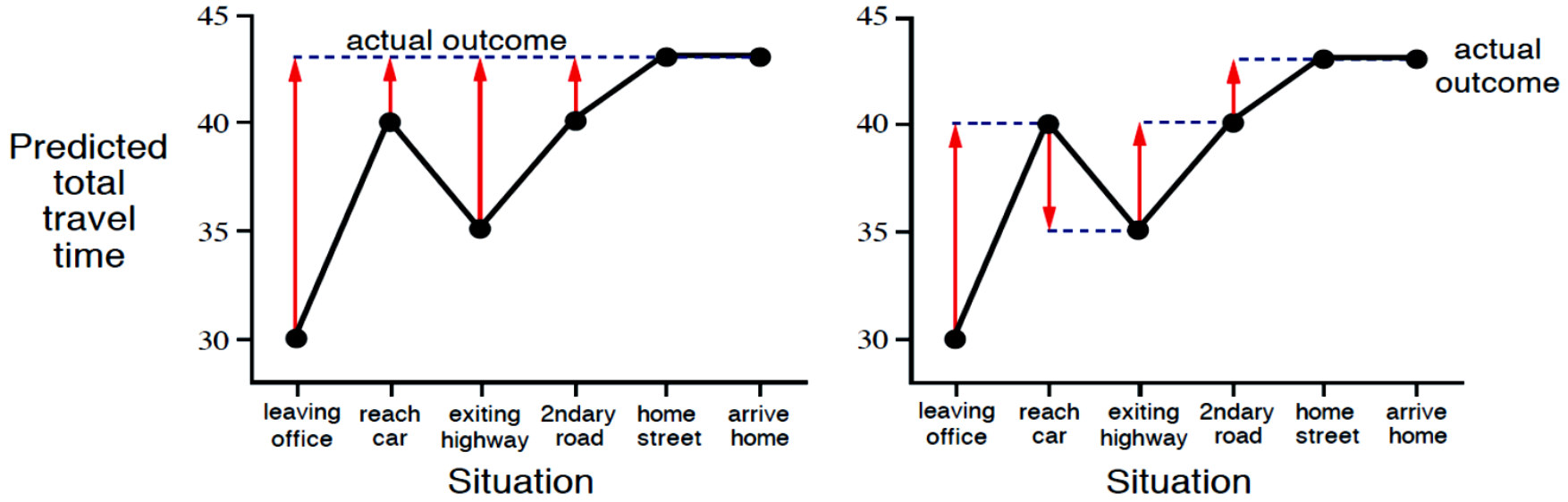
- at each time is the error in the estimate made at that time.
- depends on the next state and next reward, it is not actually available until one time step later

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

- The rewards is the elapsed time on each leg of the journey
- If not discounting then the return for each state is the actual time to go from that state.
- The value of each state is the predicted time to go

Changes recommended in the driving home example by Monte Carlo methods vs TD methods



Advantages of TD

TD methods better than:

DP methods do not require a model of the environment, of its reward and next-state probability distributions

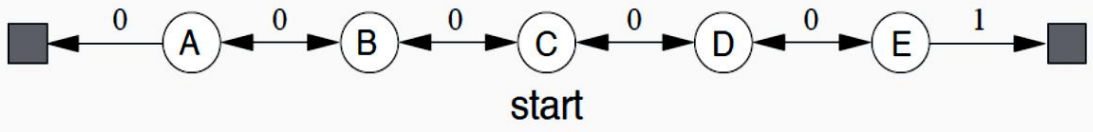
Monte Carlo methods

- as they can be implemented in an online, fully incremental fashion
- Need to wait only 1 time step for updation

both TD and Monte Carlo methods converge asymptotically to the correct predictions

Random Walk

In this example we empirically compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process:



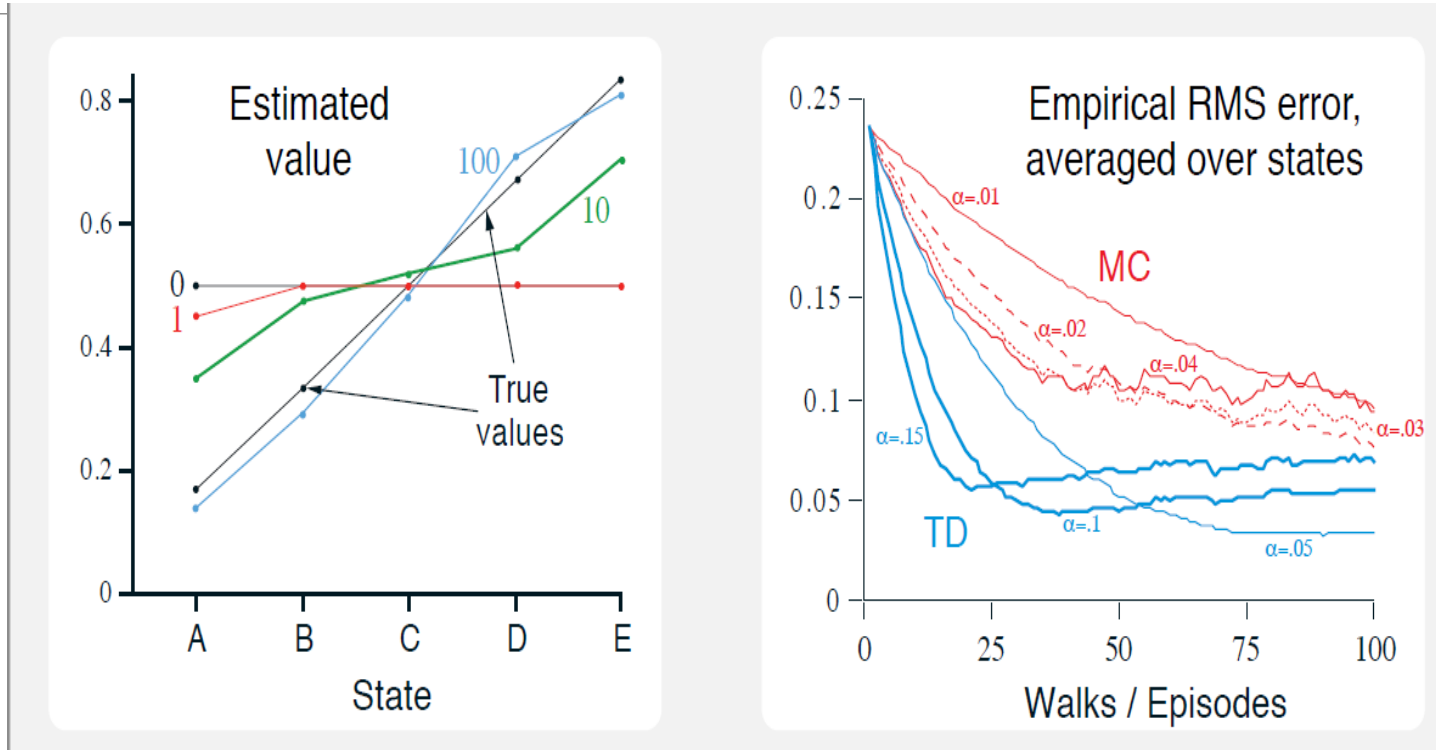
- all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability.
- Episodes terminate either on the extreme left or the extreme right.
- When an episode terminates on the right, a reward of +1 occurs
- all other rewards are zero.

For example

Episode : C, 0,B, 0, C, 0,D, 0, E, 1.

- If this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state.
- Thus, the true value of the center state is $v_{\pi}(C) = 0.5$.
- The true values of all the states, A through E, are 1/6, 2/6,3/6,4/6,5/6

Random Walk



The performance measure shown is the root mean-squared (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs

Optimality of TD(0)

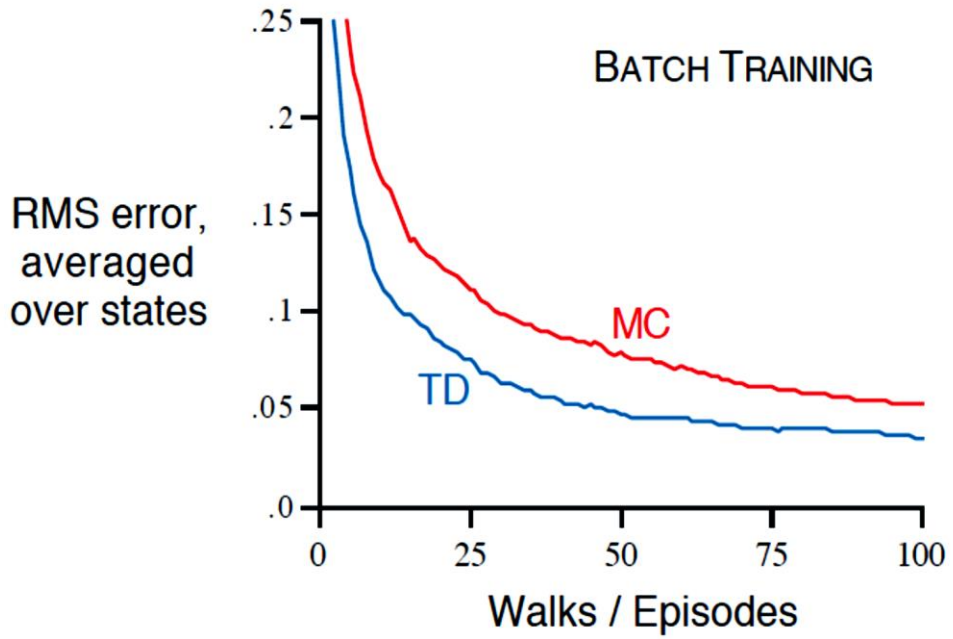
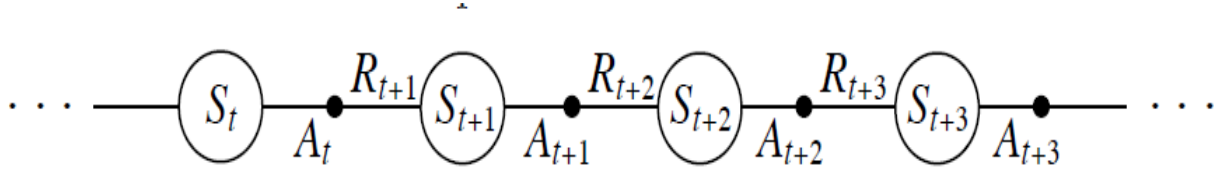


Figure 6.2: Performance of TD(0) and constant-α MC under batch training on the random walk task.

Batch updating

- because updates are made only after processing each complete batch of training data
- Suppose there is available only a finite amount of experience, say 100 time steps
- common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer
- Given an approximate value function, V , the increments are computed for every time step t at which a nonterminal state is visited
- but the value function is changed only once, by the sum of all the increments
- Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges

Sarsa: On-policy TD Control



- for an on-policy method we must estimate $q_{\Pi}(s, a)$ for the current behavior policy Π and for all states s and actions a.
- The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

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

This update is done after every transition from a nonterminal state St.
If S_{t+1} is terminal, then $Q(S_{t+1},A_{t+1})$ is defined as zero.
This rule uses every element of the quintuple of events, $(S_t,A_t,R_{t+1}, S_{t+1},A_{t+1})$, that make up a transition from one state–action pair to the next.
This quintuple gives rise to the name Sarsa for the algorithm

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

Initialize S

Choose A from S using policy derived from Q (e.g., ε-greedy)

Loop for each step of episode:

Take action A, observe R, S'

Choose A' from S' using policy derived from Q (e.g., ε-greedy)

$Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$

$S \leftarrow S'; A \leftarrow A';$

until S is terminal

On-policy control algorithm based on Sarsa prediction method

On Policy :

- continually estimate q_{Π} for the behavior policy Π
- and at the same time change Π toward greediness with respect to q_{Π}.

ε-greedy or ε-soft policies

- SARSA converges with probability 1 to an optimal policy
- Action-value function as long as all state–action
- pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy
- (ε = 1/t)

Sarsa

Expected SARSA

Instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy.

The Update Rule is

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned}$$

Given the next state, S_{t+1}, this algorithm moves deterministically in the same direction as SARSA moves in expectation

Expected SARSA is more complex computationally than SARSA but, in return, it eliminates the variance due to the random selection of A_{t+1}

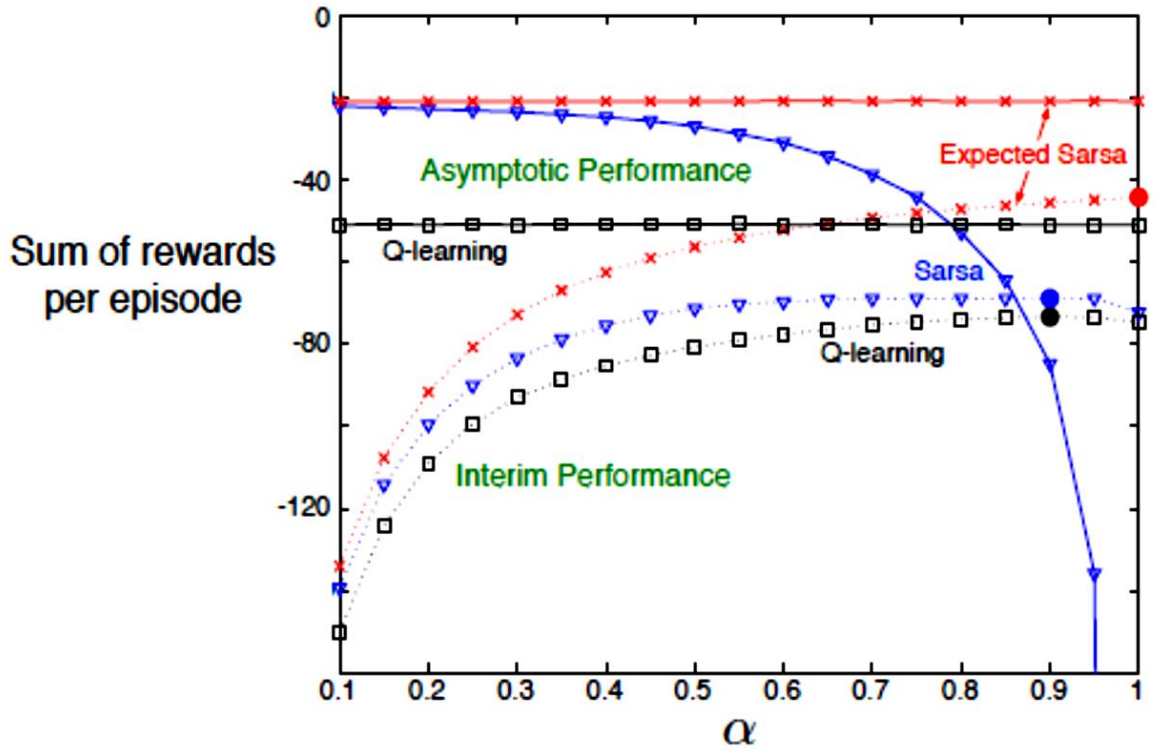


Figure 6.3: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).

Q-Learning: Off Policy TD Control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989)

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

- The learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed.
- This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs.
- The policy still has an effect in that it determines which state–action pairs are visited and updated.
- However, all that is required for correct convergence is that all pairs continue to be updated.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:

  Initialize S

  Loop for each step of episode:

    Choose A from S using policy derived from Q (e.g., ε-greedy)

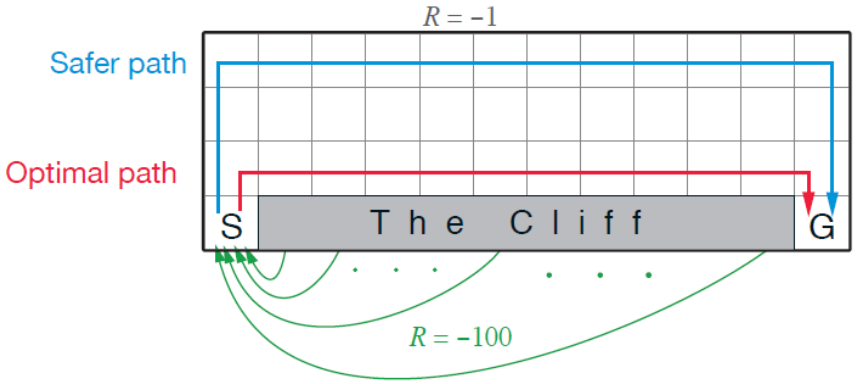
    Take action A, observe R, S'

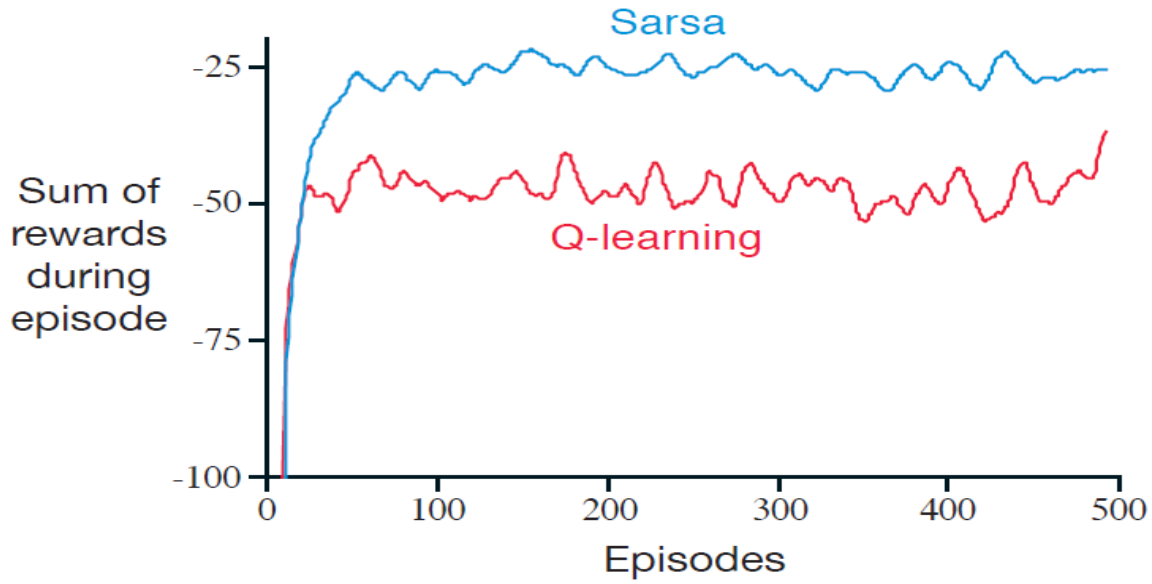
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$

    $S \leftarrow S'$

  until S is terminal

Cliff Walking

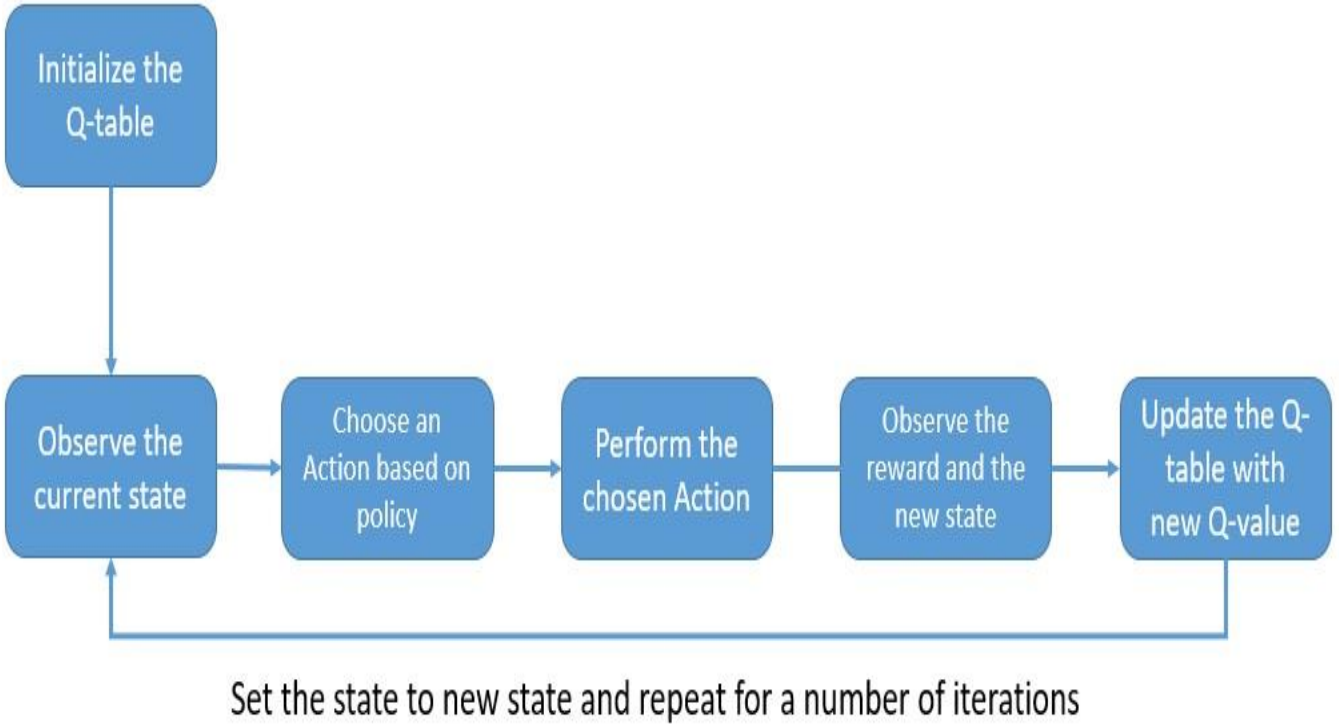




Comparison of QL and SARSA

- Both approach work in
  - finite environment
  - or discretized continuous environment
- QL directly learns the optimal policy while SARSA learns a “near” optimal policy.
- QL is a more aggressive agent, while SARSA is more conservative.
  - Example- In Cliff Walking
  - QL will take the shortest path because it is optimal while SARSA will take the longer, safer route
- In practice
  - For fast-iterating environment, QL should be your choice
  - If mistakes are costly , then SARSA is the better option
  - If state space is too large, Use deep q network
    - For example: Ms. Pac-Man has 150 pellets which can be present or absent
    - The number of possible states is for pellets only 2^{150}
    - Add all possible combinations of positions for all ghosts- so very difficult to estimate q-value

Q-Learning Process

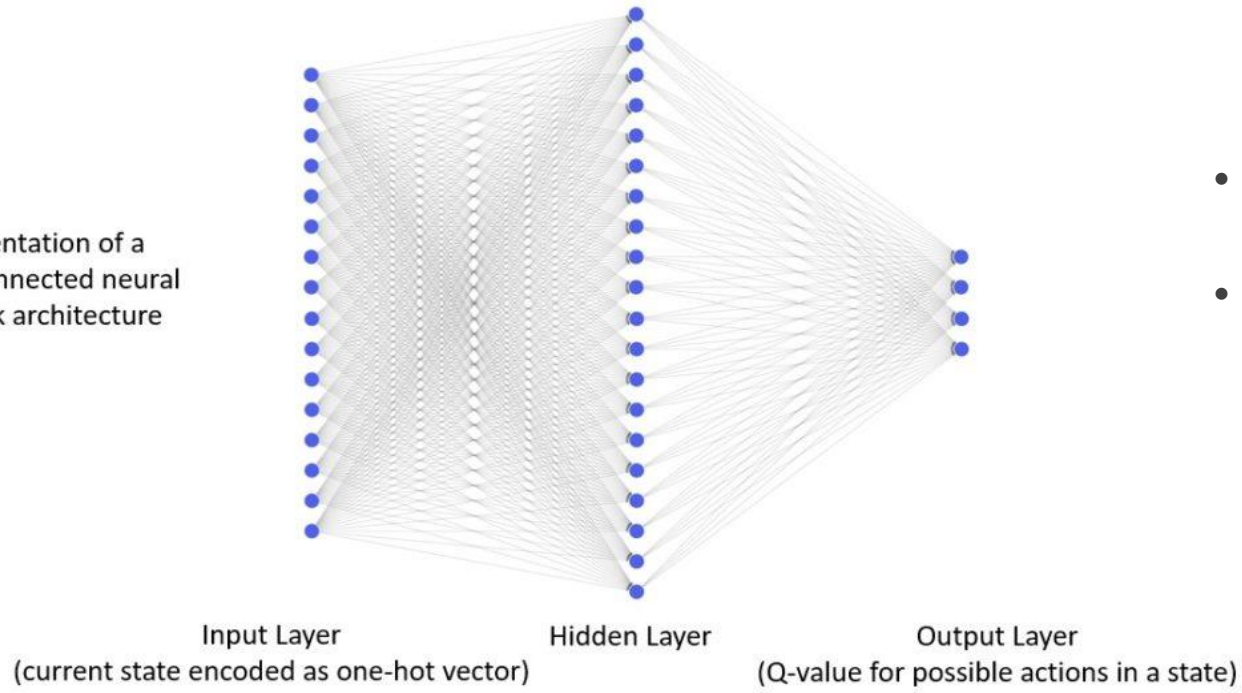


- The q-values are stored and updated in a **q-table**
- dimensions matching the number of actions and states in the environment

Reinforcement Learning with Neural Networks

- The number of actions and states in a real-life environment can be thousands, making it extremely inefficient to manage q-values in a table
- can use neural networks to predict q-values for actions in a given state instead of using a table.

Representation of a fully-connected neural network architecture



- ReLU as the activation function in the hidden layer
- Linear activation function in the output layer

Reinforcement Learning with Neural Networks

The Loss Function and Optimizer

The mean-squared-error loss function measures the square value of the difference between the prediction and the target:

$$\text{loss} = \{(\text{r} + \gamma * max_{a'} Q'(s', a')) - Q(s, a)\}^2$$

take the feedback backward through the network and update the weights

Simplest Solution is Backpropagation with the classical stochastic gradient descent

Reinforcement Learning with Neural Networks

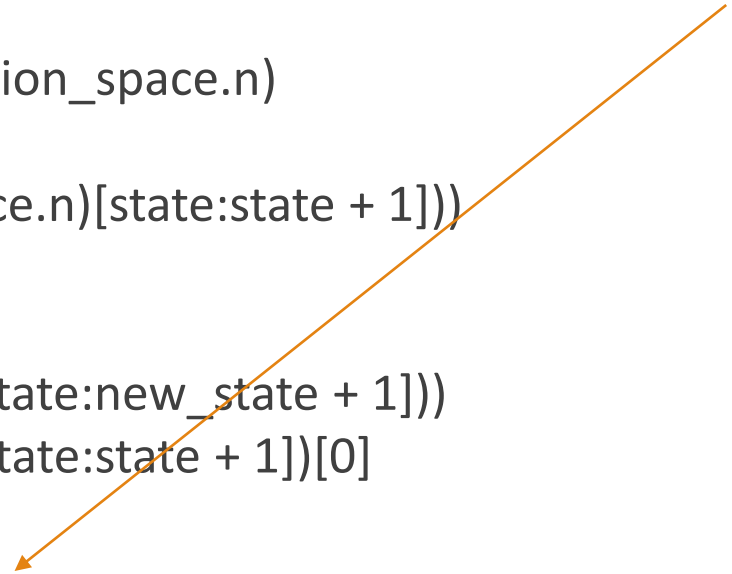
Code Blocks:

```
// Parameters
 discount_factor = 0.95
 eps = 0.5
 eps_decay_factor = 0.999
 num_episodes=500

 //Neural Network
    model = Sequential()
    model.add(InputLayer(batch_input_shape=(1, env.observation_space.n)))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(env.action_space.n, activation='linear'))
    model.compile(loss='mse', optimizer='adam', metrics=['mae’])
```

Reinforcement Learning with Neural Networks

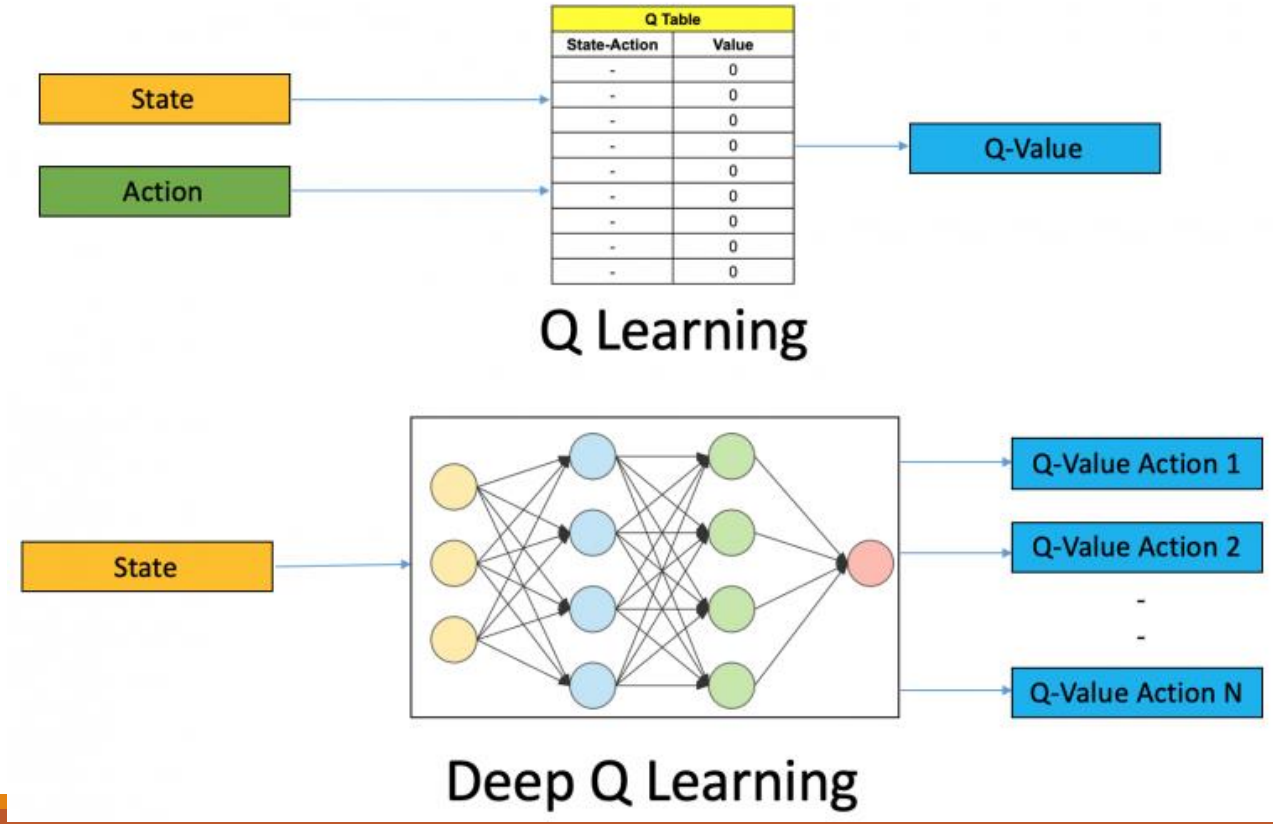
```
for i in range(num_episodes):
    state = env.reset()
    eps *= eps_decay_factor
    done = False
    while not done:
        if np.random.random() < eps:     action = np.random.randint(0, env.action_space.n)
        else:
            action = np.argmax(model.predict(np.identity(env.observation_space.n)[state:state + 1]))
        new_state, reward, done, _ = env.step(action)
        target = reward +  discount_factor *
            np.max(model.predict(np.identity(env.observation_space.n)[new_state:new_state + 1]))
        target_vector = model.predict(np.identity(env.observation_space.n)[state:state + 1])[0]
        target_vector[action] = target
        model.fit(np.identity(env.observation_space.n)[state:state + 1],
            target_vec.reshape(-1, env.action_space.n),  epochs=1, verbose=0)
        state = new_state
```

one-hot encoded current state, and the target value converted as a vector to train the model on a single step



Deep Q Learning

- approximate these Q-values with machine learning models such as a neural network
- idea behind DeepMind's algorithm that led to its acquisition by Google for 500 million dollars!



Steps in Deep Q Networks

1. All the past experience is stored by the user in memory
2. The next action is determined by the maximum output of the Q-network
3. The loss function here is mean squared error of the predicted Q-value and the target

   Q-value – Q*

   As per the Q-value update equation derived from the Bellman equation:

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

   - green represents the target
   - Since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge

Psuedocode for Deep Q-Learning

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, ... till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = R(s, a, s')$$

Sample new initial state s'

else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

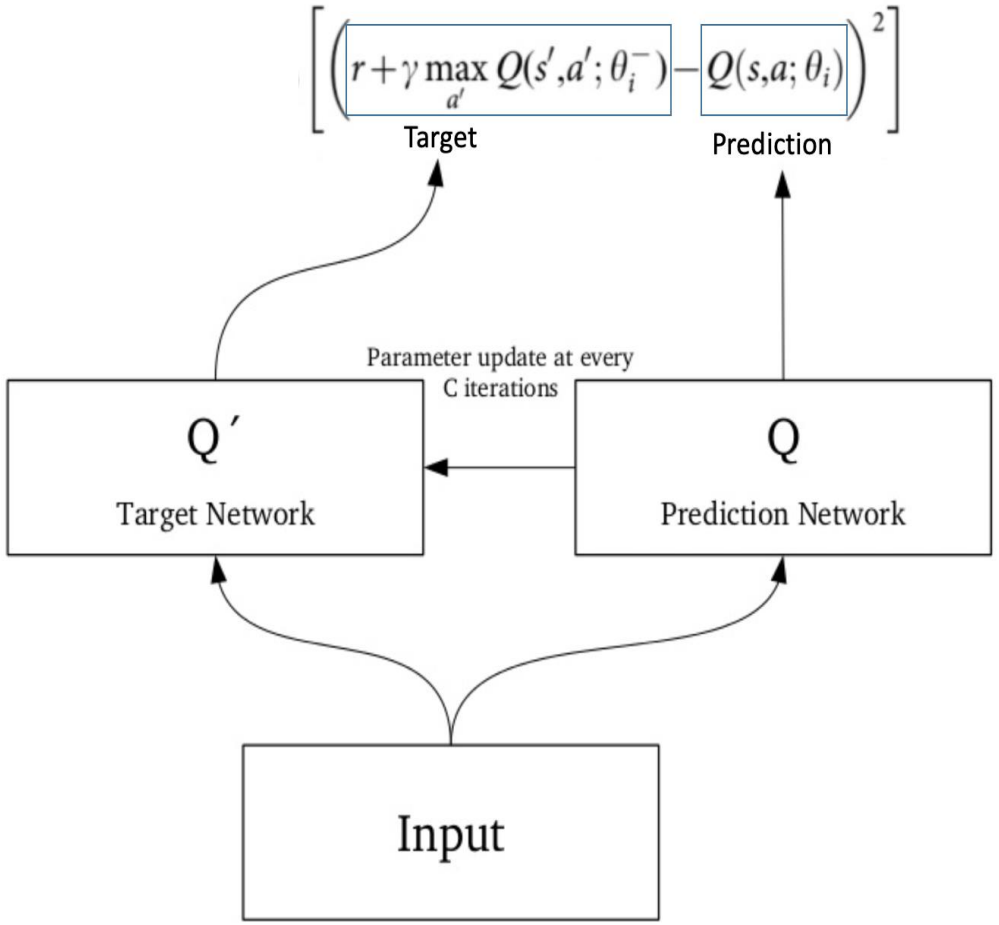
$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[(Q_\theta(s, a) - \text{target}(s'))^2 \right] \Big|_{\theta=\theta_k}$$

$$s \leftarrow s'$$

Chasing a nonstationary target!

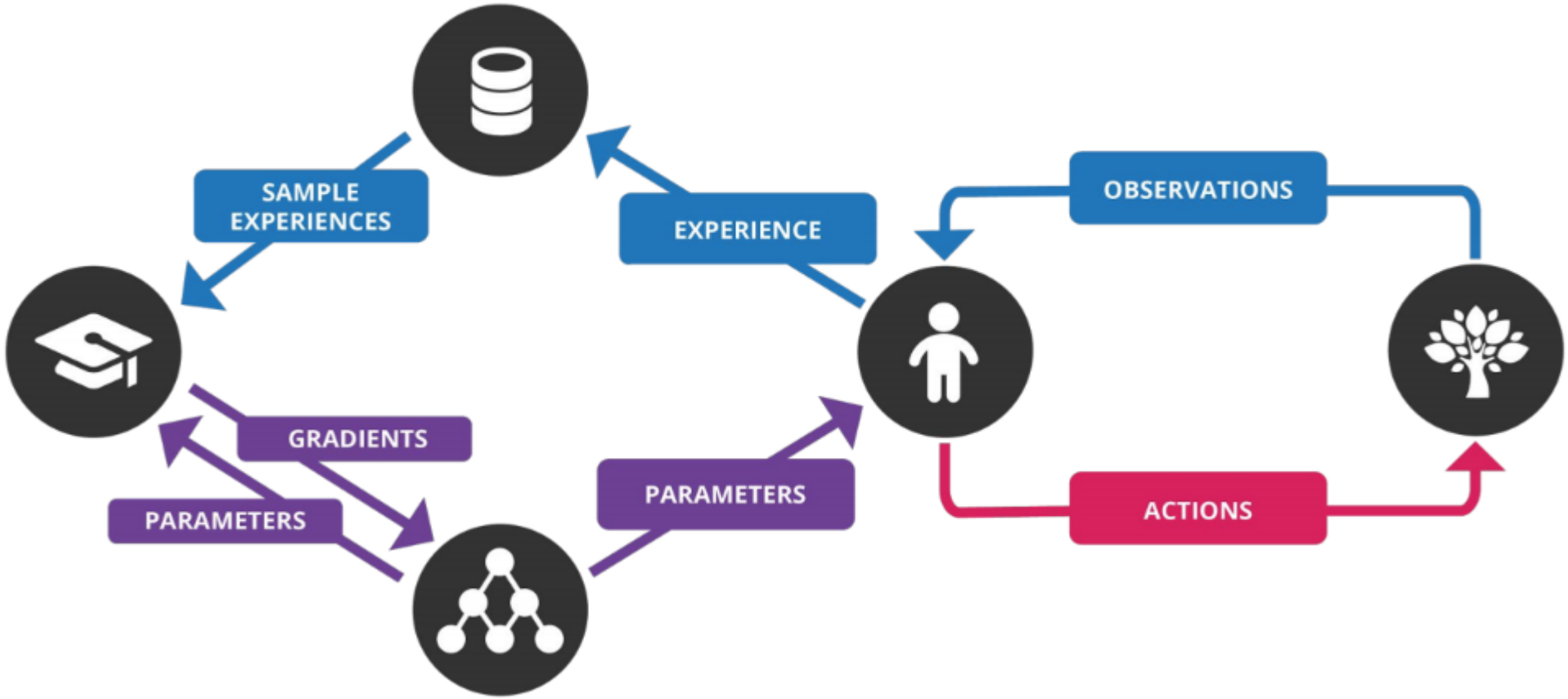
Updates are correlated within a trajectory!

Solution in deep Q-network (DQN)



- Separate network to estimate the target
- **Target network** has the same architecture as the function approximator but with frozen parameters
- For every **C iterations** (a hyperparameter), the parameters from the **prediction network** are copied to the target network.
- This leads to more stable training because it keeps the target function fixed (for a while):
- **Experience Replay**
  - Instead of running Q-learning on state/action pairs as they occur during simulation or the actual experience
  - the system stores the data discovered for [state, action, reward, next_state] – in a large table.

deep Q-network (DQN)



steps involved in deep Q-network (DQN) :

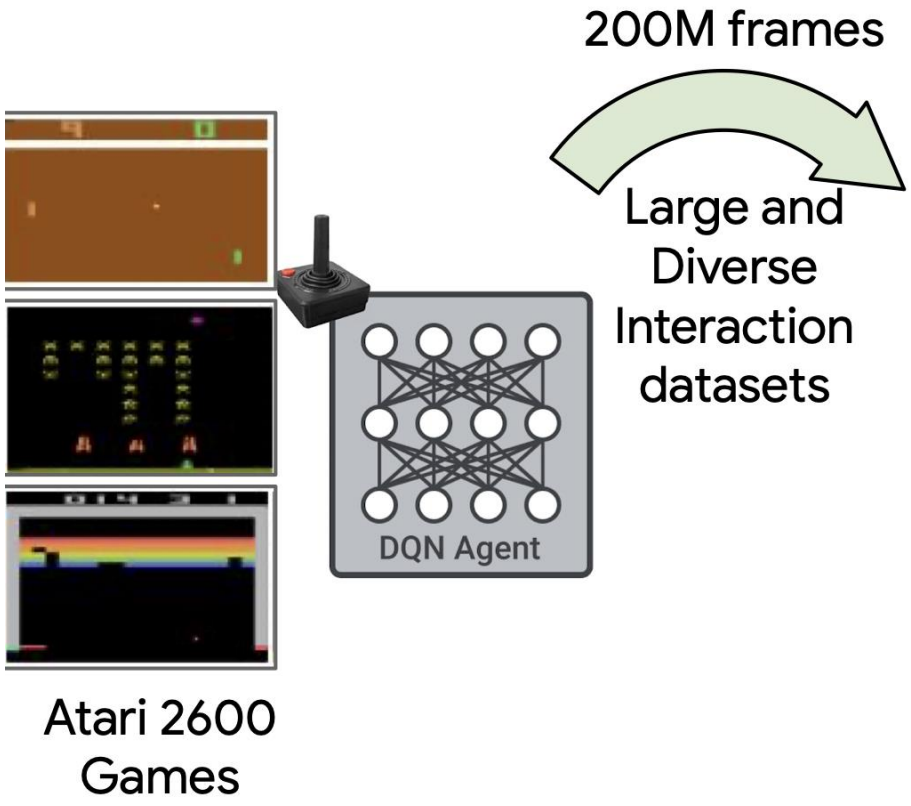
1. Preprocess and feed the game screen (state s) to DQN, which will return the Q-values of all possible actions in the state
2. Select an action using the ε-greedy policy.
3. Perform this action in a state s and move to a new state s’ to receive a reward.
4. This state s’ is the preprocessed image of the next game screen. We store this transition in our replay buffer as <s,a,r,s’>
5. Next, sample some random batches of transitions from the replay buffer and calculate the loss- (squared difference between target Q and predicted Q)
6. Perform gradient descent with respect to our actual network parameters in order to minimize this loss
7. After every C iterations, copy our actual network weights to the target network weights
8. Repeat these steps for M number of episodes

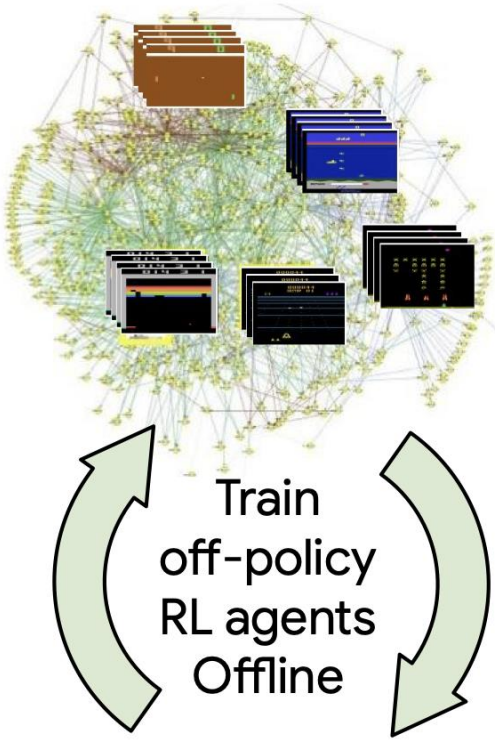
Cartpole & DQN

```
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
```

Cartpole & DQN

```
policy = EpsGreedyQPolicy()

memory = SequentialMemory(limit=50000, window_length=1)

dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory,
nb_steps_warmup=10,

target_model_update=1e-2, policy=policy)

dqn.compile(Adam(lr=1e-3), metrics=['mae'])


dqn.fit(env, nb_steps=5000, visualize=True, verbose=2)
```





Discretizing Continuous State Space

In case of Video- Frames are input & CNN is used in the DQN

Cartpole - 5-state discretization of the state space

- S = { hard_lean_left,
- soft_lean_left,
- soft_lean_right,
- hard_lean_right,
- fallen

}

Discretizing Continuous State Space

Rewards

fallen => -1

soft_lean_left => +1

soft_lean_right => +1

hard_lean_left => 0

hard_lean_right => 0

Actions the agent could invoke on the cart in order to balance the pole:

A = { push_left,

push_right

}

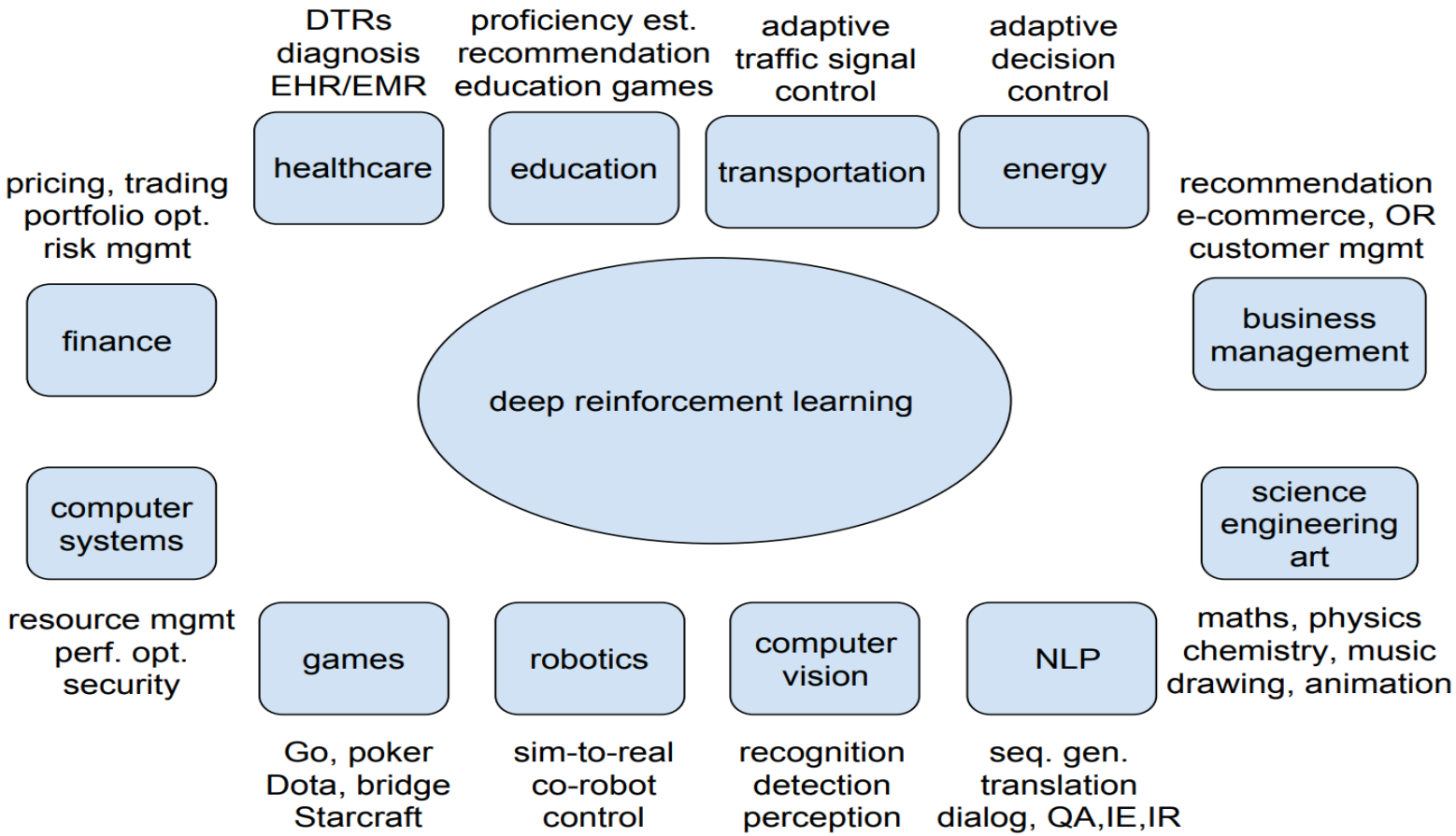
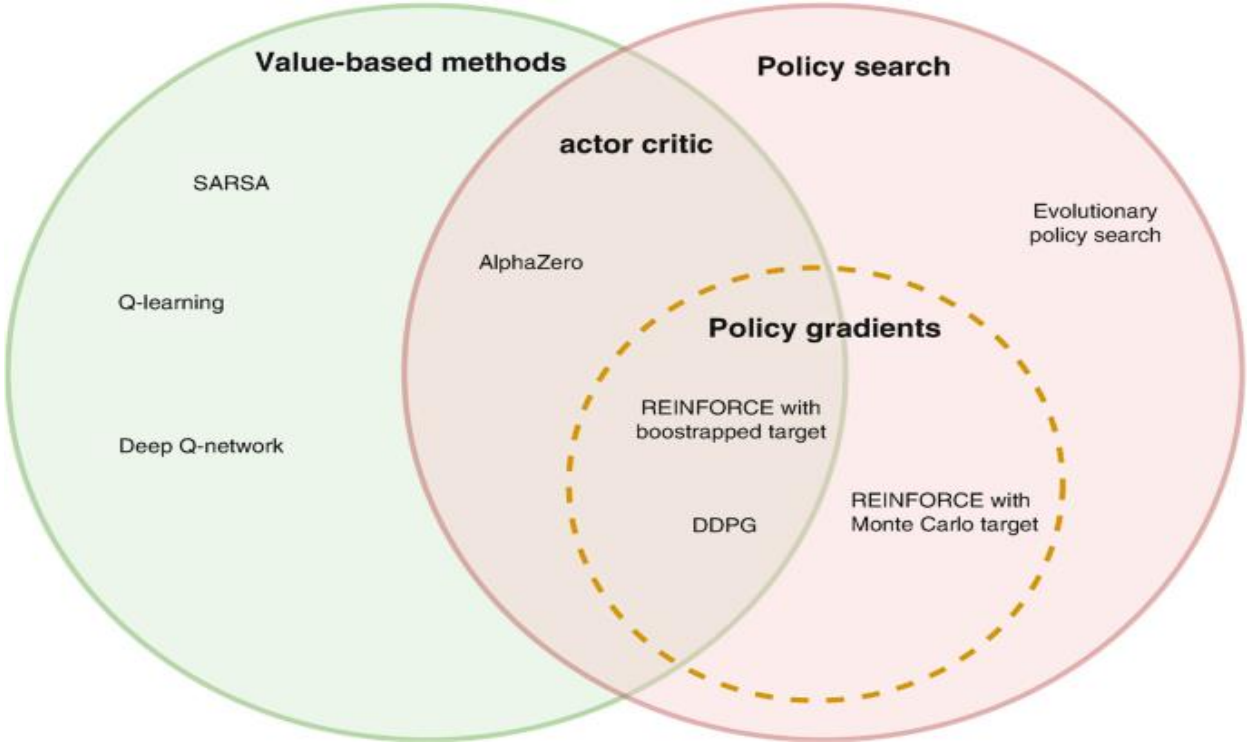


Figure 4: Deep Reinforcement Learning Applications

- Industry Ready Applications
  - Google's Cloud AutoML
  - Facebook's Horizon Platform

Policy vs Value based Reinforcement Learning

- **Policy learning** focuses on directly inferring a policy that maximizes the reward on a specific environment
- **Value learning** tries to quantify the value of every state-action pair
- **Example:** AI agent trying to learn a new chess opening
- Using policy reinforcement learning, the AI agent would try to infer a strategy to develop the pieces in a way that can achieve certain well-known position
- In Value-learning, the AI agent would assign a value to every position and select the moves that score higher.



Deterministic Vs Stochastic Policy

Policy is a set of rules that is referenced whenever we want to know what action to take

Deterministic policy

- offer a state-action mapping $\pi: s \mapsto a$
- Ideally the optimal mapping (that is, if all the Bellman equations are learned to perfection)

Stochastic policy

- conditional probability distribution over the actions in a given state, $\pi: P(a|s)$

Advantages of Stochastic Policy: Use in

1. Continuous Action Spaces
2. Stochastic Environments
3. Multi-agent environments
4. Partially Observable MDPs

For example : In game of rock-paper-scissors use stochastic policy

REINFORCE

Key idea

- **reinforcing good actions:** to push up the probabilities of actions that lead to higher return
- and push down the probabilities of actions that lead to a lower return
- until we get the optimal policy.

The policy gradient method will iteratively amend the policy network weights

- (with smooth updates)
- to make state-action pairs that resulted in positive return more likely
- and make state-action pairs that resulted in negative return less likely

Trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$

Return for a Trajectory $R(\tau) = (G_0, G_1, ..., G_H)$

Total Return or Future Return $G_k \leftarrow \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_k$

REINFORCE

Policy: A policy is defined as the probability distribution of actions given a state

$$\pi(A_t = a | S_t = s)$$

$$\forall A_t \in \mathcal{A}(s), S_t \in \mathcal{S}$$

- The objective of a Reinforcement Learning agent is to maximize the "expected" reward when following a policy π.
- Define a set of parameters θ to parametrize this policy
- Parameters can be the coefficients of a complex polynomial or the weights and biases of units in a neural network
- If we represent the total reward for a given trajectory τ as r(τ), we arrive at :
- ***Reinforcement Learning Objective****: Maximize the "expected" reward following a parametrized policy*

$$J(\theta) = \mathbb{E}_{\pi}\left[r(\tau)\right]$$

$$\sum_{\tau} \mathbb{P}(\tau;\theta) R(\tau)$$

REINFORCE

- Notation Θ is the policy's parameter vector
- Thus Π(a|s, Θ) = Pr{A_t=a | S_t=s, Θ_t=Θ} for the probability that action a is taken at time t given that the environment is in state s at time t with parameter Θ
- learning the policy parameter based on the gradient of some scalar performance measure J(Θ) with respect to the policy parameter
- These methods seek to maximize performance, so their updates approximate gradient ascent in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}, \qquad \text{where } \widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$$

is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument Θ_t

REINFORCE (Monte Carlo Policy Gradients)

Input: a differentiable policy parameterization $\pi(a|s,\theta)$
Algorithm parameter: step size $\alpha > 0$)
Initialize the policy parameter θ at random

(1) Use the policy π_θ to collect a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...a_H, r_{H+1}, s_{H+1})$

(2) Estimate the Return for trajectory τ: $R(\tau) = (G_0, G_1, ..., G_H)$
where G_k is the expected return for transition k:

$$G_k \leftarrow \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_k$$

(3) Use the trajectory τ to estimate the gradient $\nabla_\theta U(\theta)$

$$\nabla_\theta U(\theta) \leftarrow \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

(4) Update the weights θ of the policy

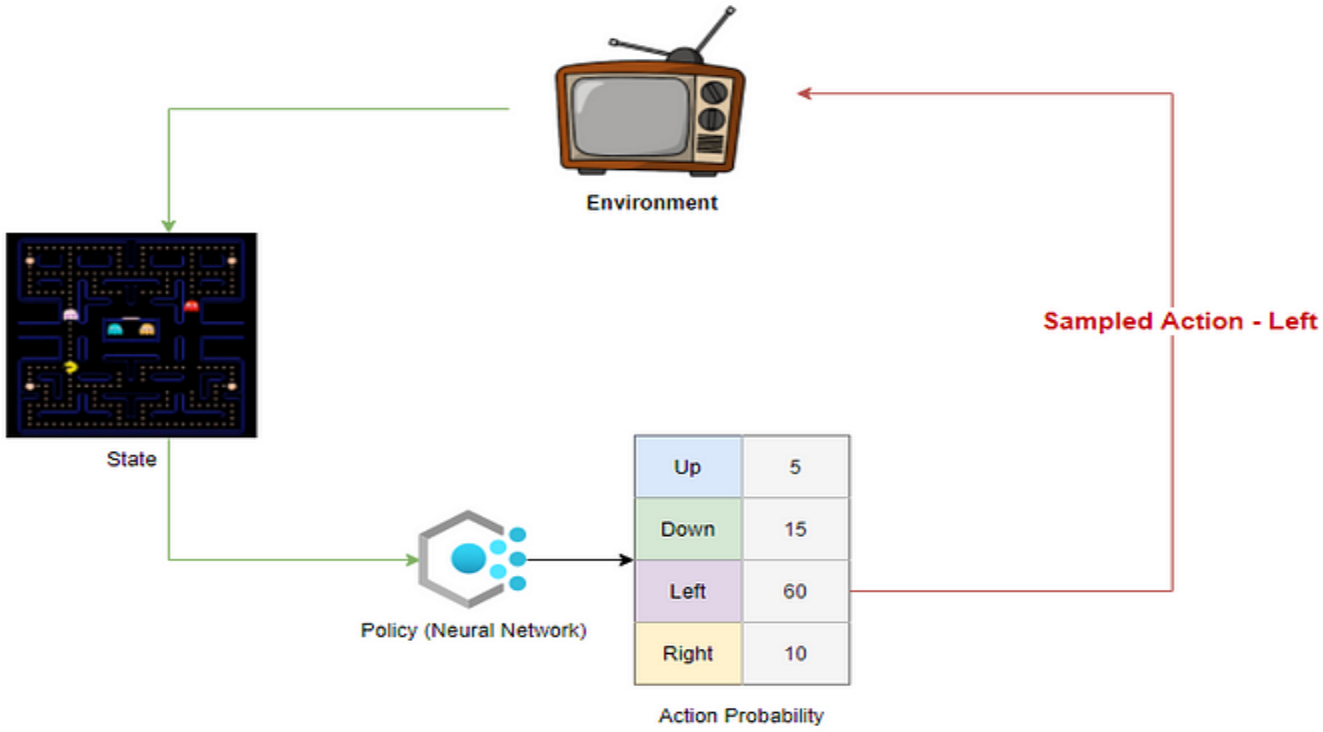
$$\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta)$$

(5) Loop over steps 1-5 until not converged

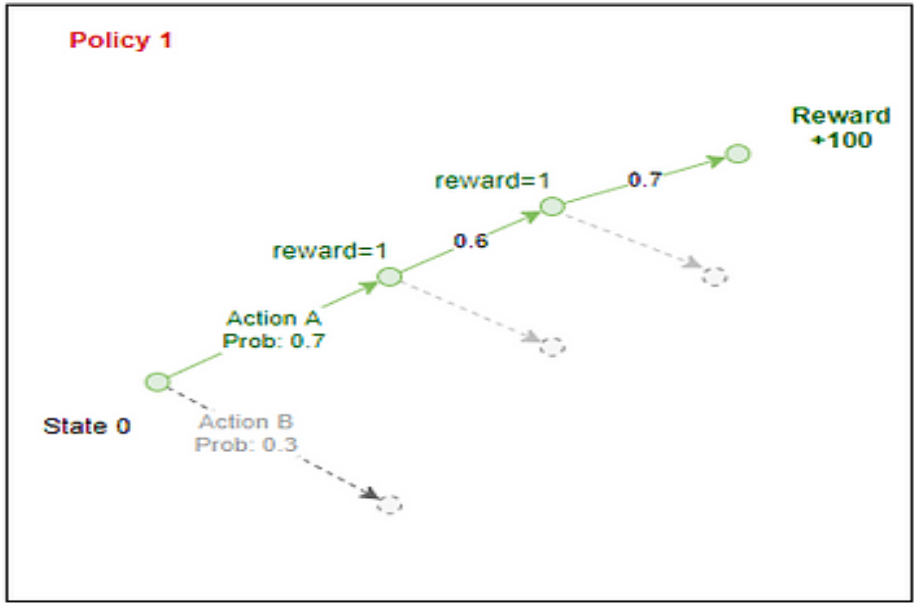
- Sometimes probabilities may be extremely tiny or very close to 1
- instead use a surrogate objective, log p (natural logarithm)
- since the log of probability space ranges from (-∞,0)
- and this makes the log probability easier to compute.

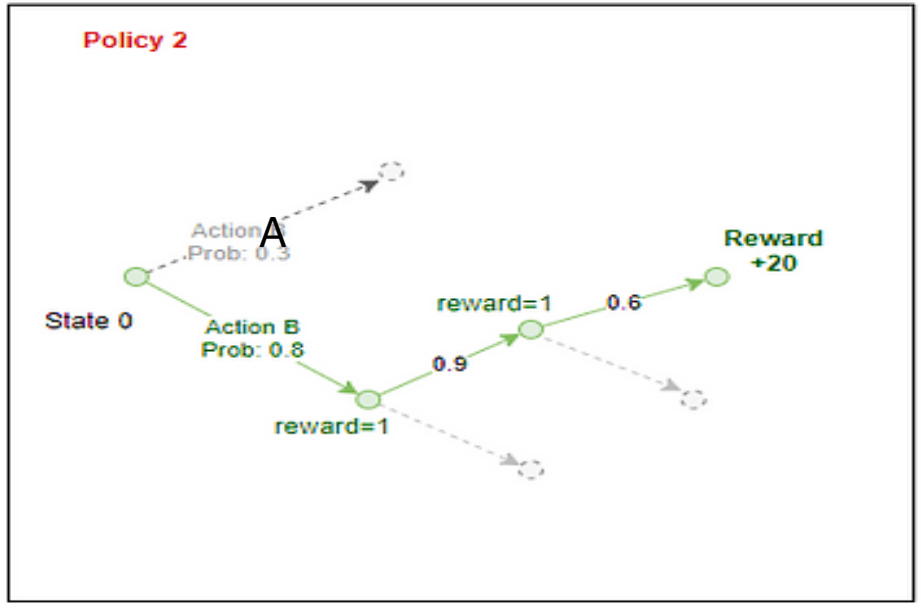
Example

The policy is usually a Neural Network that takes the state as input and generates a probability distribution across action space as output



Example: Policy 1 vs Policy 2 Trajectories



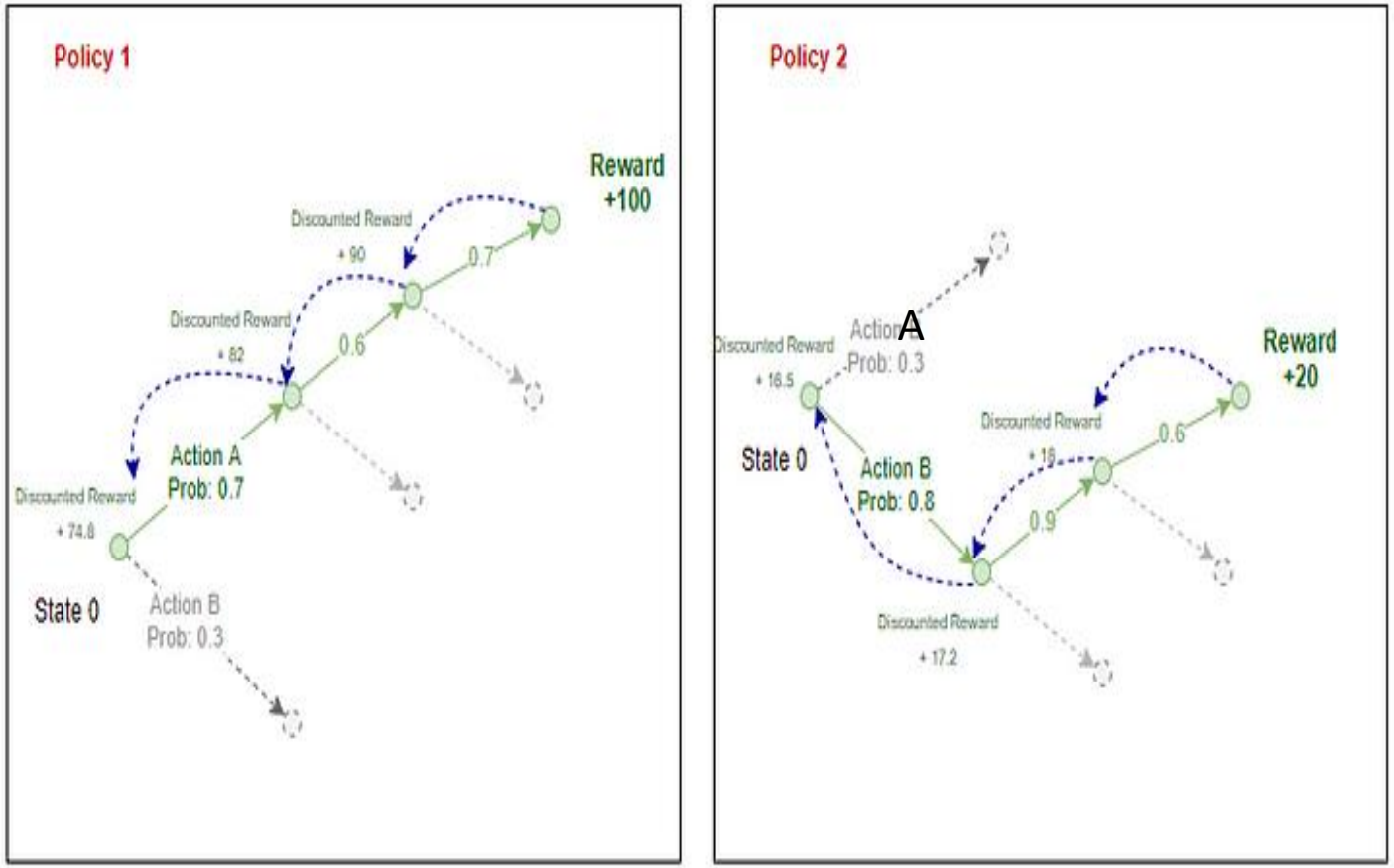


$$\text{Discounted Reward at } t,\ R(t) = r(t) + \gamma * R(t+1) = r(t) + \gamma * r(t+1) + \gamma^2 * r(t+2) + \ldots + \gamma^{(T-t)} * r(T)$$

γ = Discount Factor, usually 0.9
T = Terminal state's time step

$$\text{Expected Reward } G(\theta) = \sum_{i=0}^{n} \text{Probability of action}_k \text{ at state}_i * \text{discounted reward}_{(k,i)}$$

θ = Policy
n = Number of steps in the episode
k = index of action



The steps in the implementation of REINFORCE

1. *Initialize a Random Policy (a NN that takes the state as input and returns the probability of actions)*
2. *Use the policy to play N steps of the game — record action probabilities-from policy, reward-from environment, action — sampled by agent*
3. *Calculate the discounted reward for each step*
4. *Calculate expected reward G*
5. *Adjust weights of Policy (back-propagate error in NN) to increase G*
6. *Repeat from 2*

Disadvantages of REINFORCE

The Policy Gradient is :

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$$

- Has high variability in log probabilities and cumulative reward values, because each trajectories during training can deviate from each other at great degrees
- Can result in noisy gradients, and cause unstable learning and/or the policy distribution skewing to a non-optimal direction
- Can have cumulative reward of 0- both “goods” and “bad” actions with will not be learned if the cumulative reward is 0

$$\text{Introducing baseline } b(s):$$

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))]$$

- will make smaller gradients, and thus smaller and more stable updates.

Common Baseline Functions

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\, G_t\right] \quad \text{REINFORCE}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; Q^w(s,a)\right] \quad \text{Q Actor-Critic}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; A^w(s,a)\right] \quad \text{Advantage Actor-Critic}$$

$$= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; \delta\right] \quad \text{TD Actor-Critic}$$

Actor Critic Methods

Policy gradient in REINFORCE

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0,a_0,\ldots,s_t,a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] \mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[G_t]$$

$$\mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[G_t] = Q(s_t, a_t)$$

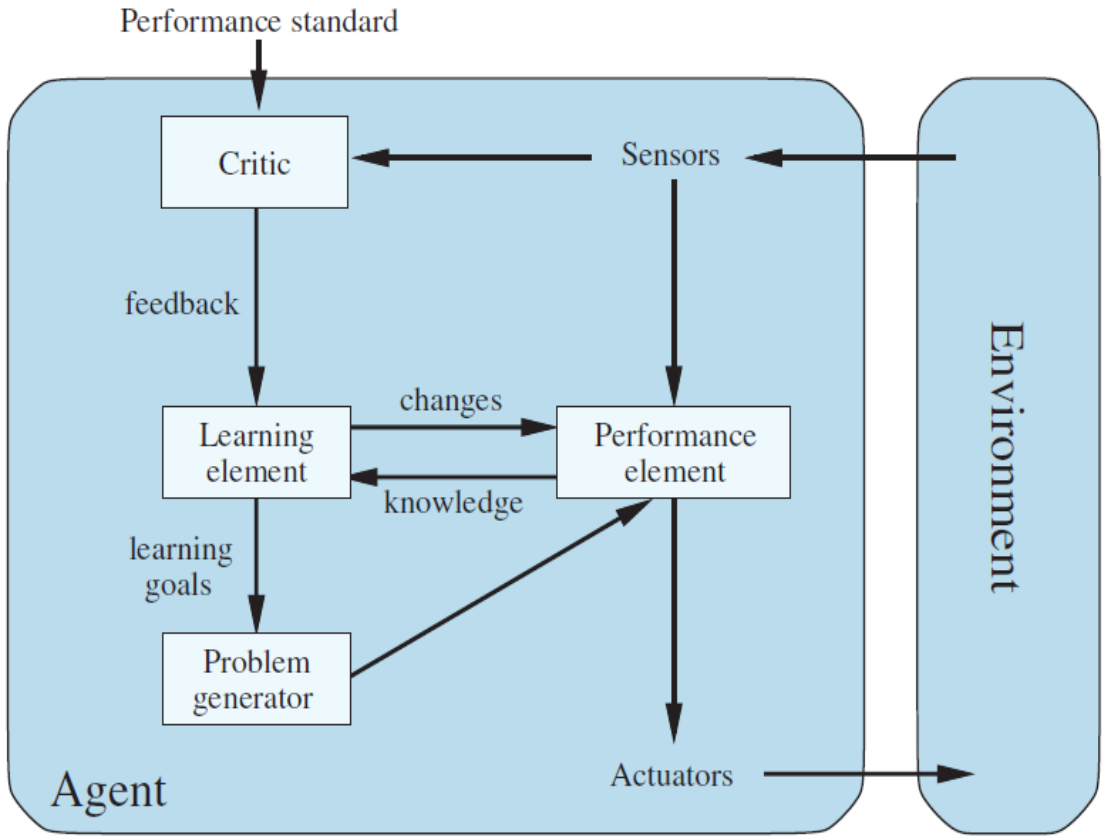
Actor Critic Methods

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0, a_0, \ldots, s_t, a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] Q_w(s_t, a_t)$$

$$= \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)]$$

Q value can be learned by parameterizing the Q function with a neural network

General Learning Agent

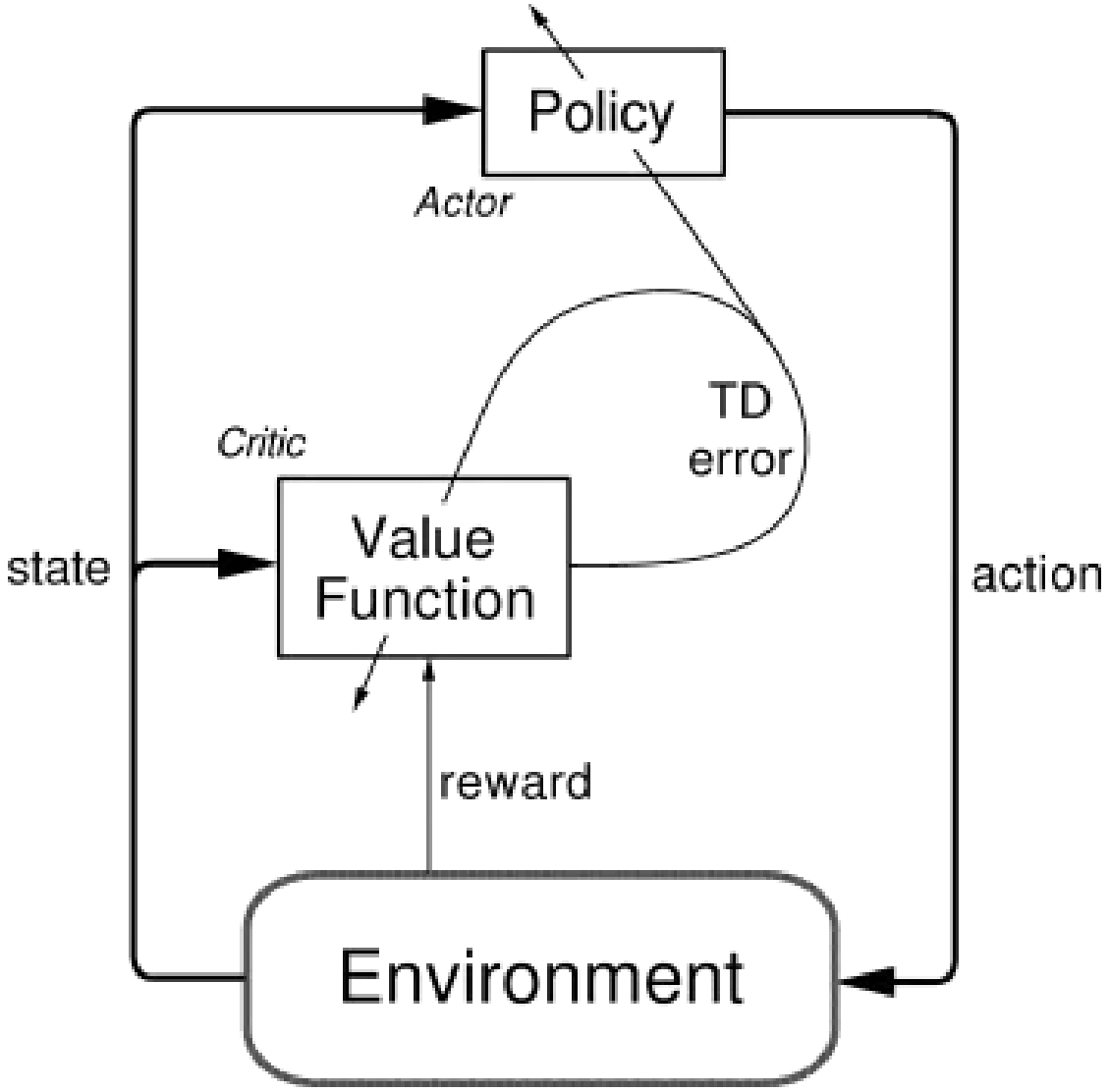


Learning element which is responsible for making improvements

Performance element, which is responsible for selecting external actions

Critic tells the learning element how well the agent is doing with respect to a fixed performance standard

Problem Generator which is responsible for suggesting actions that will lead to new and informative experiences.



Actor Critic Methods

- **The "Critic" estimates the value function**
  - This could be the action-value (the Q value) or state-value (the V value).
- **The "Actor" updates the policy distribution**
  - in the direction suggested by the Critic (such as with policy gradients).
- Both the Critic and Actor functions are parameterized with neural networks

Algorithm 1 Q Actor Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.

for $t = 1 \ldots T$: **do**

Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$

Then sample the next action $a' \sim \pi_\theta(a'|s')$

Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta \log \pi_\theta(a|s)$; Compute the correction (TD error) for action-value at time t:

$$\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$$

and use it to update the parameters of Q function:

$$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$$

Move to a $\leftarrow a'$ and s $\leftarrow s'$

end for

Advantage Actor Critic (A2C)

Advantage Value- Take a specific action compared to the average, general action at the given state

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

- Using Bellman Equation:

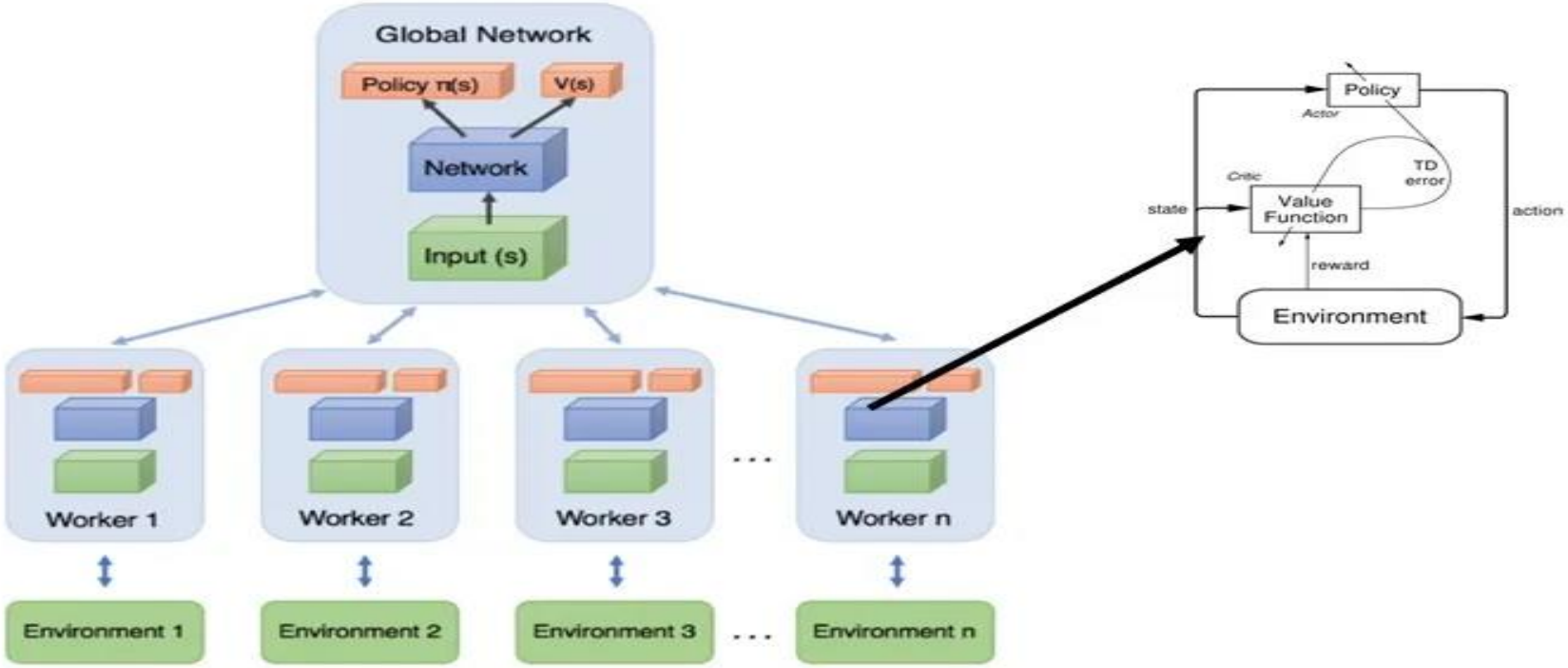
$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]$$

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

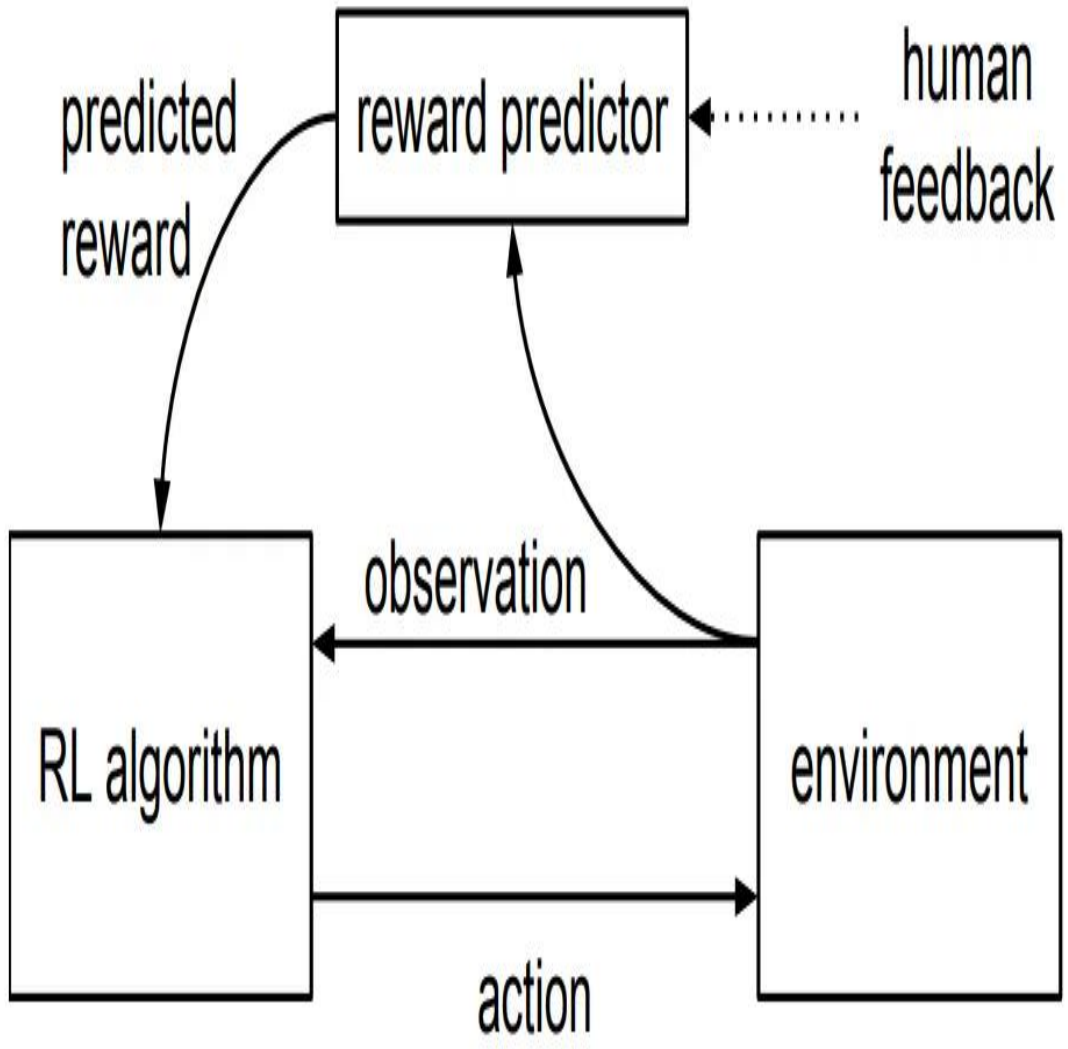
$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t))$$

$$= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

Asynchronous Advantage Actor Critic (A3C)



- implements parallel training where multiple workers in parallel environments independently update a global value function
- Key benefit of having asynchronous actors is effective and efficient exploration of the state space.

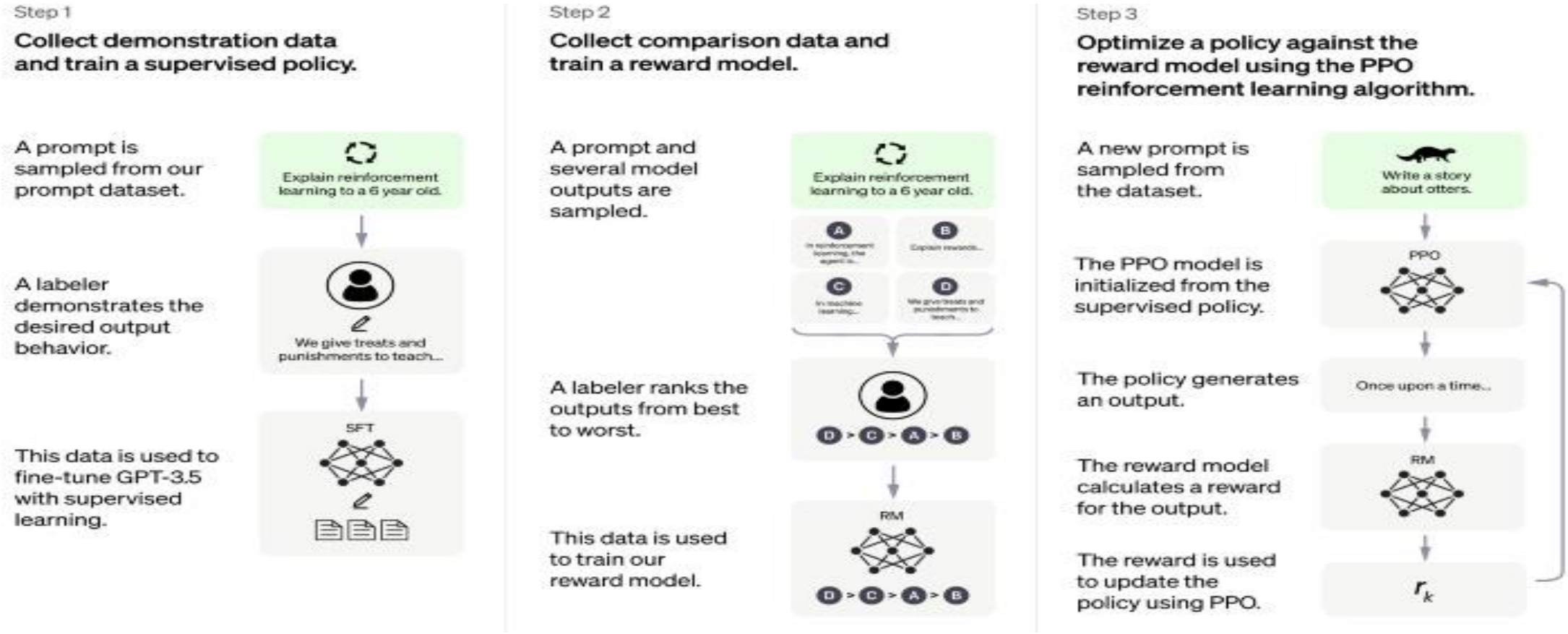


ChatGPT- Reinforcement Learning from Human Feedback

The method overall consists of three distinct steps:

1. **Supervised fine-tuning**
   - A pre-trained language model is fine-tuned on a relatively small amount of demonstration data curated by labelers, to learn a supervised policy (the SFT model) that generates outputs from a selected list of prompts
   - This represents the baseline model.
2. **Mimic human preferences**
   - Labelers are asked to vote on a relatively large number of the SFT model outputs, creating a new dataset consisting of comparison data.
   - A new model is trained on this dataset
   - This is referred to as the reward model (RM).
3. **Proximal Policy Optimization (PPO)**
   - The reward model is used to further fine-tune and improve the SFT model.
   - The outcome of this step is the so-called policy model.

How ChatGPT works



Proximal Policy Optimization (PPO) algorithms are policy gradient methods

References

1. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010

2. Richard S Sutton, Andrew G Barto, Reinforcement Learning, second edition, MIT Press

3. https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/home/week/1