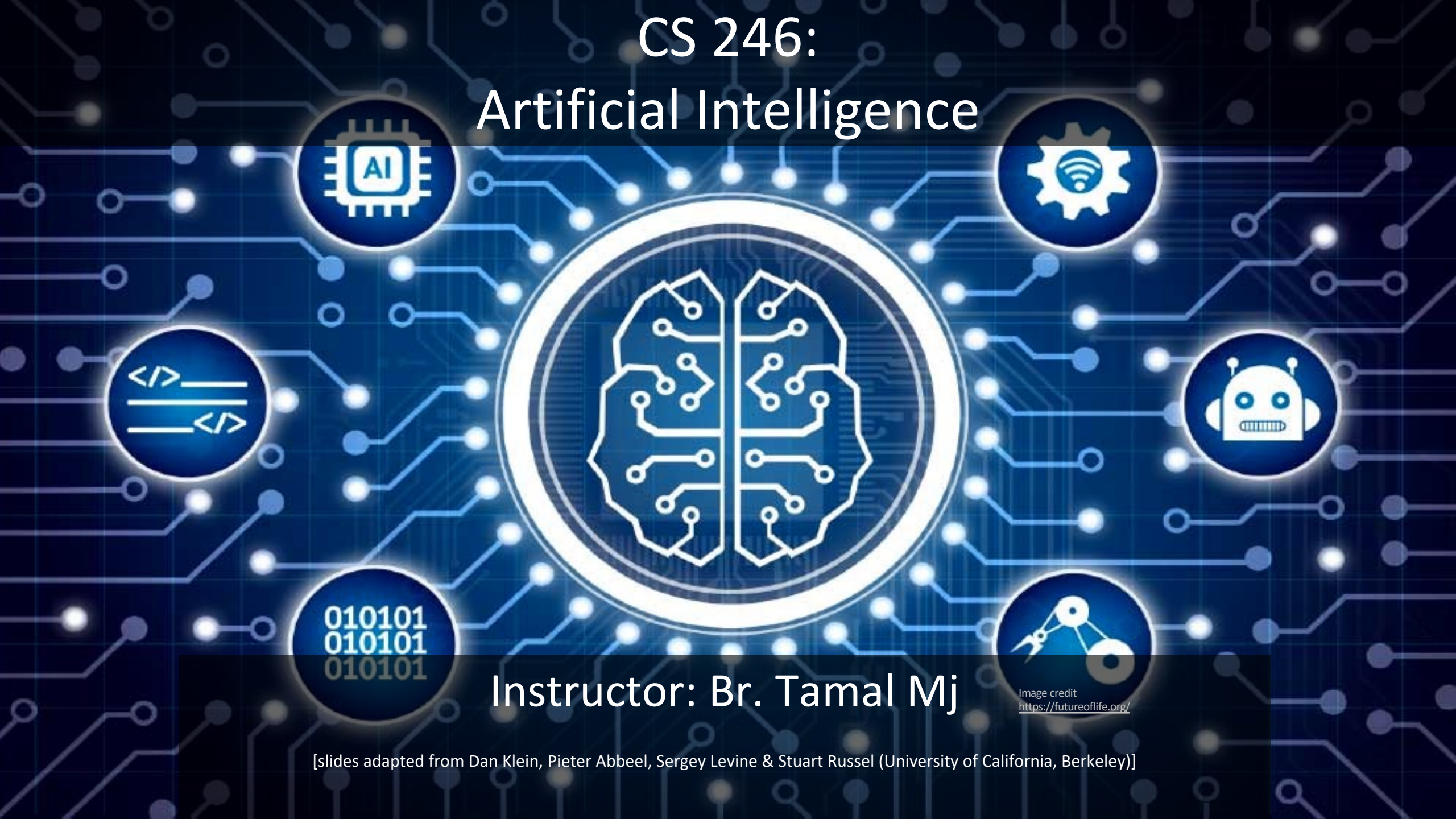
CS 246:
Artificial Intelligence



Instructor: Br. Tamal Mj

Image credit
https://futureoflife.org/

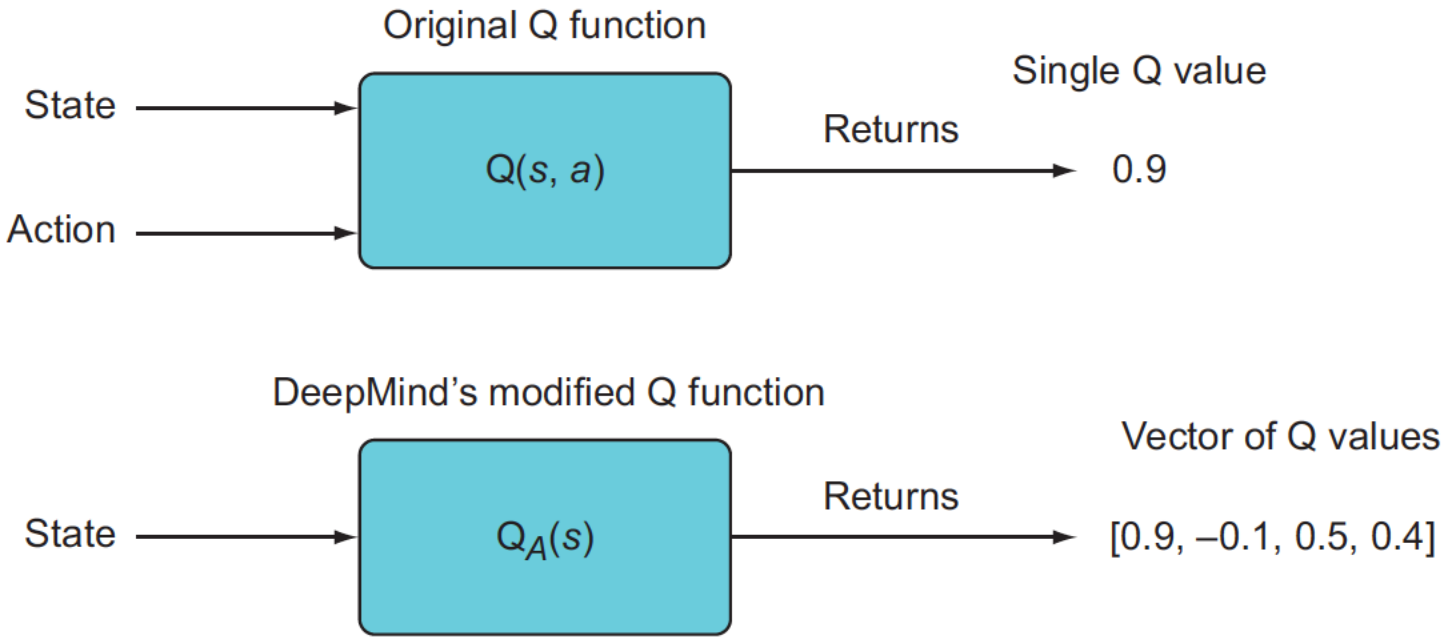
[slides adapted from Dan Klein, Pieter Abbeel, Sergey Levine & Stuart Russel (University of California, Berkeley)]

Om Saha Naav[au]-Avatu
Saha Nau Bhunaktu
Saha Viiryam Karavaavahai
Tejasvi Naav[au]-Adhiitam-
Astu Maa Vidvissaavahai
Om Shaantih Shaantih
Shaantih

Om, May we all be protected
May we all be nourished
May we work together with great energy
May our intelect be sharpened (may our study be effective)
Let there be no Animosity amongst us
Om, peace (in me), peace (in nature), peace (in divine forces)

Deep Reinforcment Learning (DRL)

Q-function for Deep RL



The original Q function accepts a state-action pair and returns the value of that state-action pair—a single number. DeepMind used a modified vector-valued Q function that accepts a state and returns a vector of state-action values, one for each possible action given the input state. The vector-valued Q function is more efficient, since you only need to compute the function once for all the actions.

State

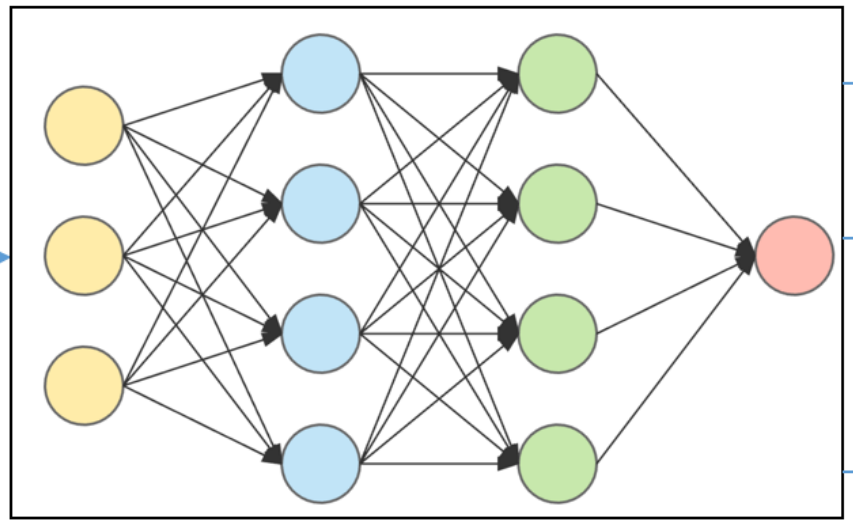
Action

Q Table	
State-Action	Value
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

Q-Value

Q Learning

State



Q-Value Action 1

Q-Value Action 2

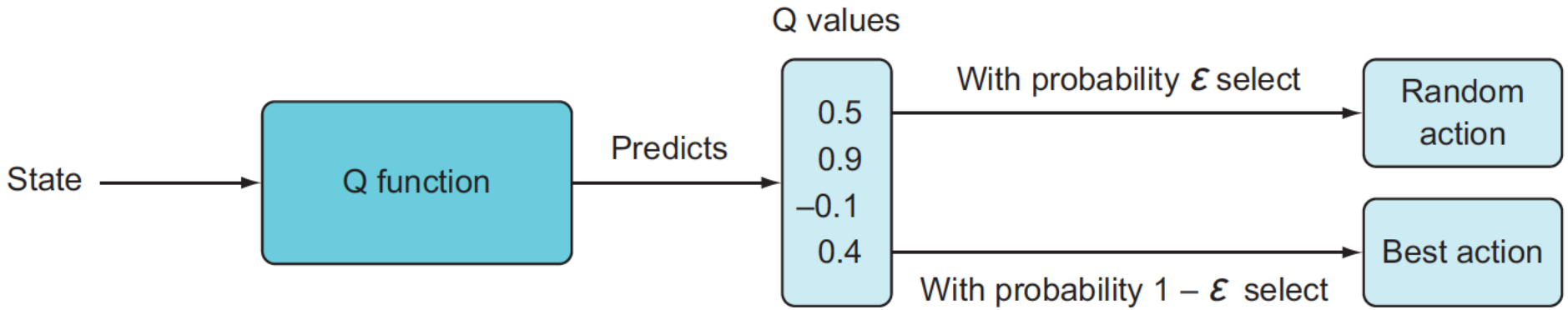
-

-

Q-Value Action N

Deep Q Learning

Epsilon Greedy Action Selection



In an epsilon-greedy action selection method, we set the epsilon parameter to some value, e.g., 0.1, and with that probability we will randomly select an action (completely ignoring the predicted Q values) or with probability 1 – epsilon = 0.9, we will select the action associated with the highest predicted Q value. An additional helpful technique is to start with a high epsilon value, such as 1, and then slowly decrement it over the training iterations.

Issues with Approx. 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:

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

Chasing a nonstationary target!

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

$$s \leftarrow s'$$

Updates are correlated within a trajectory!

The target is continuously changing with each iteration.

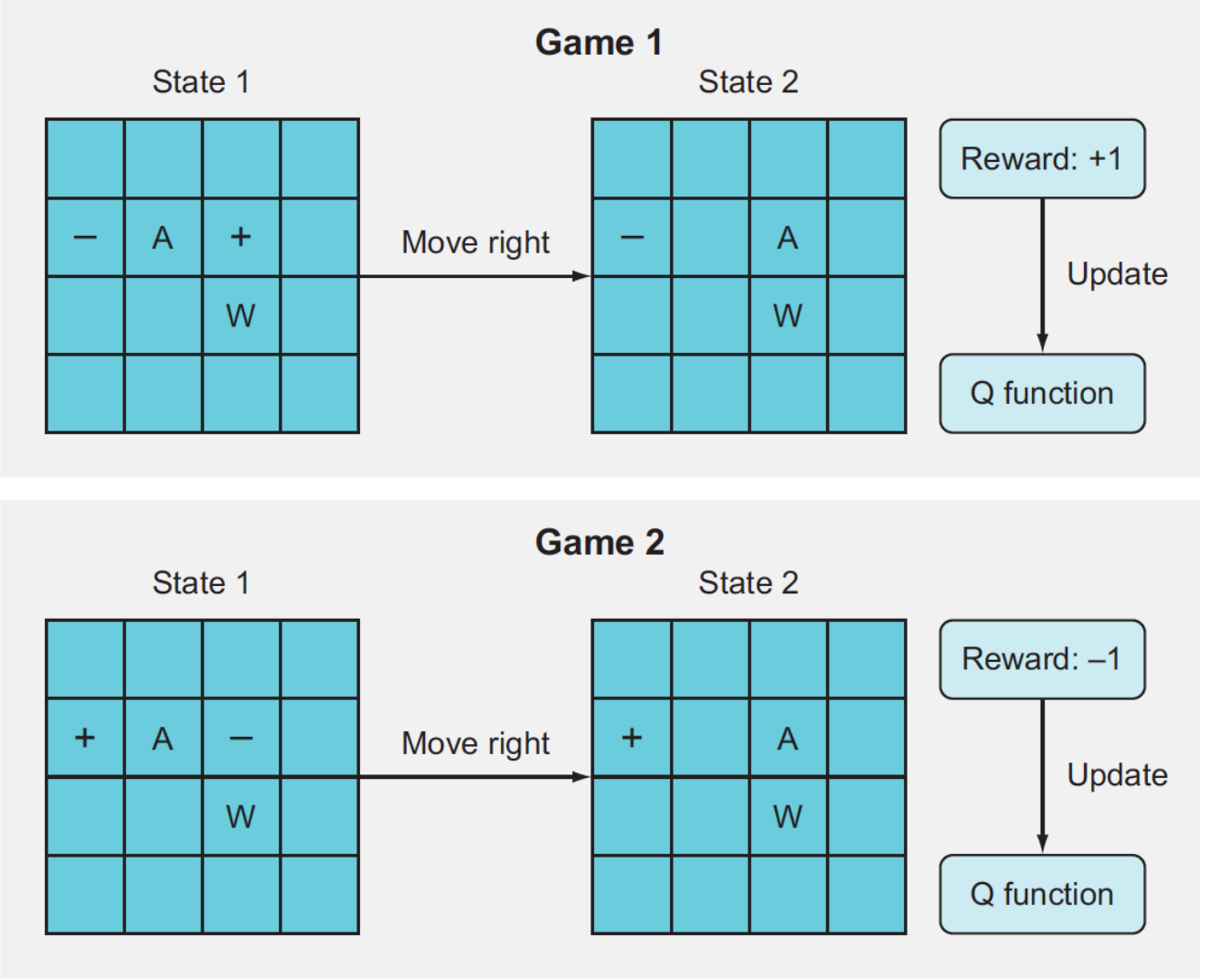
In deep learning, the target variable does not change and hence the training is stable, which is just not true for RL.

Catastrophic Forgetting

The idea of catastrophic forgetting is that when two game states are very similar and yet lead to very different outcomes, the Q function will get “confused” and won’t be able to learn what to do.

In this example, the catastrophic forgetting happens because the Q function learns from game 1 that moving right leads to a +1 reward, but in game 2, which looks very similar, it gets a reward of –1 after moving right.

As a result, the algorithm forgets what it previously learned about game 1, resulting in essentially no significant learning at all.



Experience replay

- In state s, take action a, and observe the new state s_{t+1} and reward r_{t+1}.
- Store this as a tuple (s, a, s_{t+1}, r_{t+1}) in a list.
- Continue to store each experience in this list until you have filled the list to a specific length
- Once the experience replay memory is filled, randomly select a subset
- Iterate through this subset and calculate value updates for each subset; store these in a target array (such as Y) and store the state, s, of each memory in X.
- Use X and Y as a mini-batch for batch training. For subsequent epochs where the array is full, just overwrite old values in your experience replay memory array.

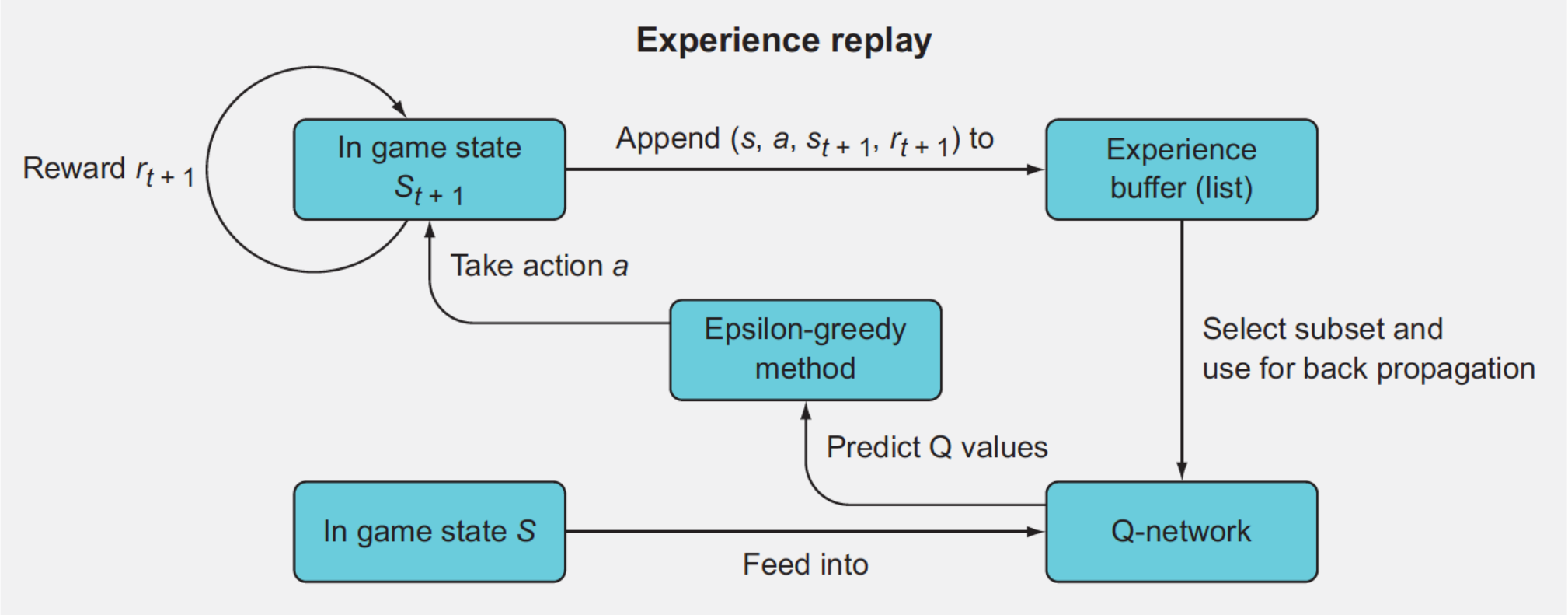


Figure 3.13 This is the general overview of experience replay, a method for mitigating a major problem with online training algorithms: catastrophic forgetting. The idea is to employ mini-batching by storing past experiences and then using a random subset of these experiences to update the Q-network, rather than using just the single most recent experience.

Improving Stability with a Target Network

- Initialize the Q-network with parameters (weights) θ_Q
- Initialize the target network as a copy of the Q-network, but with separate parameters θ_T, and set $\theta_T = \theta_Q$.
- Use the ϵ-greedy strategy with the Q-network's Q values to select action a.
- Observe the reward and new state r_{t+1}, s_{t+1}
- The target network's Q value will be set to
  - r_{t+1} if the episode has just been terminated or
  - $r_{t+1} + \gamma \, max Q_{\theta_T}(S_{t+1})$ otherwise
- Backpropagate the target network's Q value through the Q-network (not the target network).
- For Every C iterations, set $\theta_T = \theta_Q$

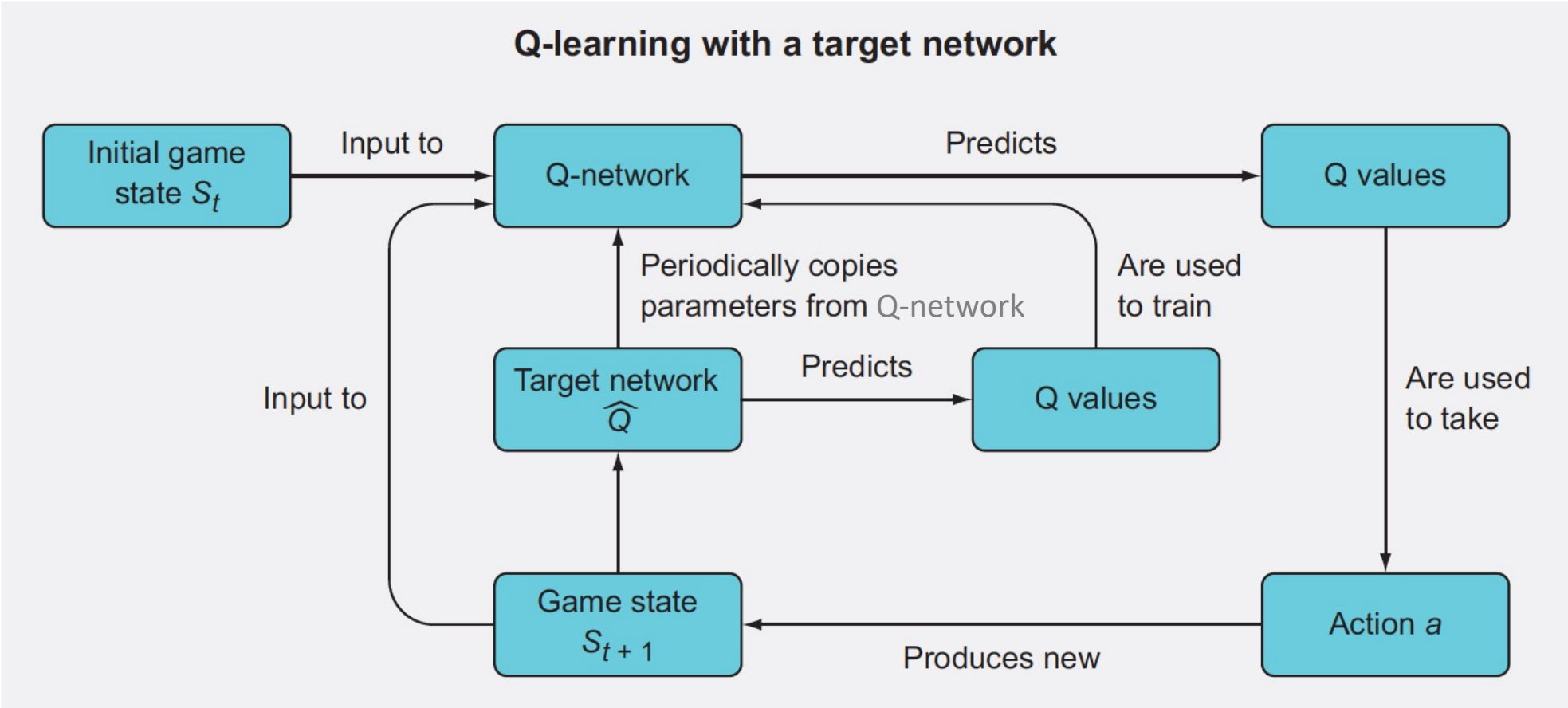
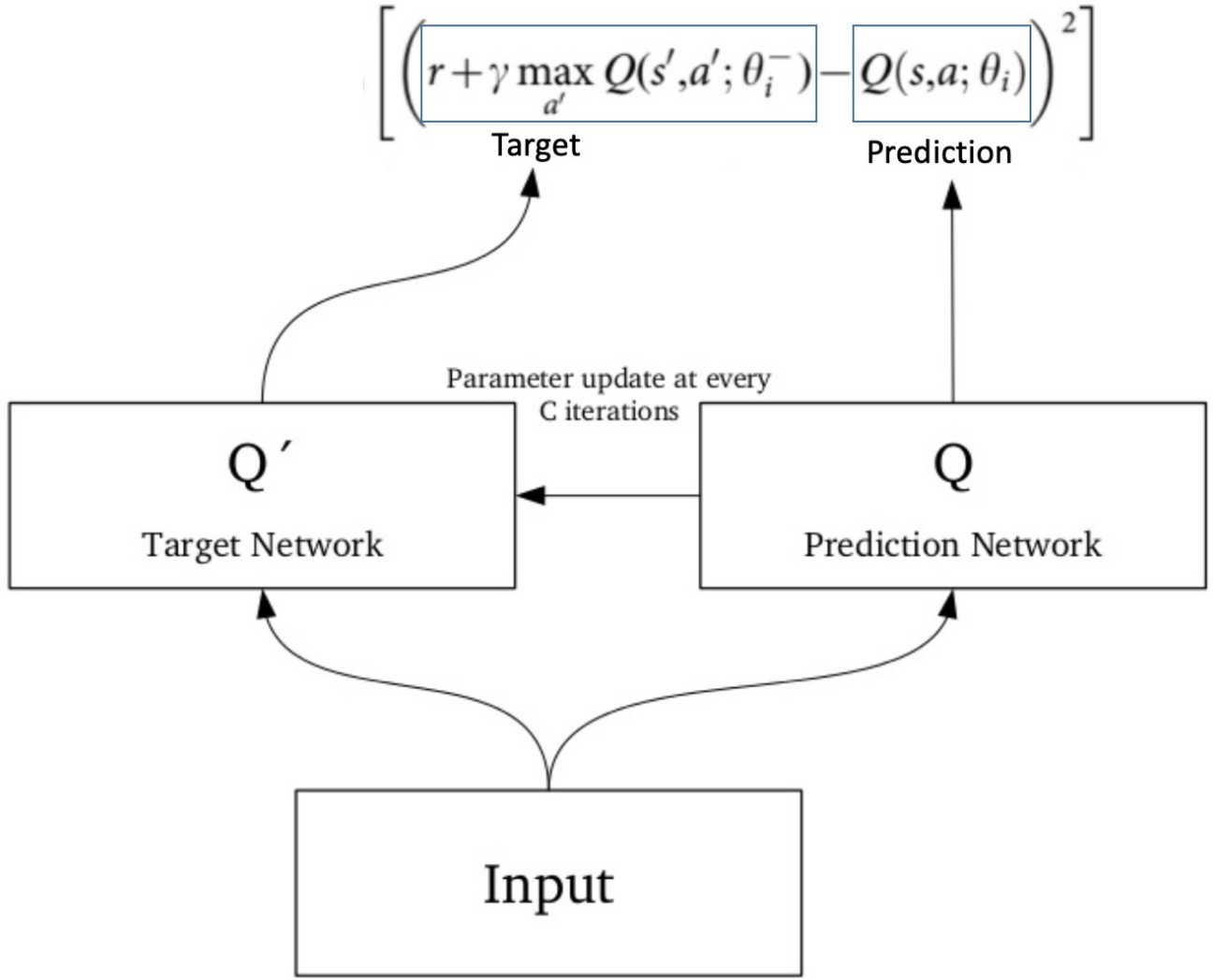


Figure 3.15 This is the general overview for Q-learning with a target network. It's a fairly straightforward extension of the normal Q-learning algorithm, except that you have a second Q-network called the target network whose predicted Q values are used to backpropagate through and train the main Q-network. The target network's parameters are not trained, but they are periodically synchronized with the Q-network's parameters. The idea is that using the target network's Q values to train the Q-network will improve the stability of the training.



https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right]. \quad (3)$$

Paper: Playing Atari with Deep Reinforcement Learning

Algorithm 1: deep Q-learning with experience replay. with a Target Network

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

Paper: Human-level control through deep reinforcement learning

Policy Gradient Methods

- What if we skip selecting a policy on top of the DQN and instead train a neural network to output an action directly?
- If we do that, our neural network ends up being a policy function, or a policy network.
- Remember that a policy function, $\pi: s \rightarrow p(a|s)$, accepts a state and returns the best action.
- More precisely, it will return a probability distribution over the actions, and we can sample from this distribution to select actions.

Policy Gradient Methods

Q-Network

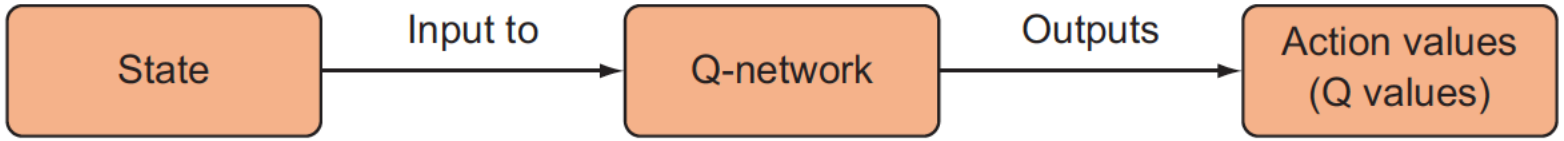


Figure 4.1 A Q-network takes a state and returns Q values (action values) for each action. We can use those action values to decide which actions to take.

Policy Network

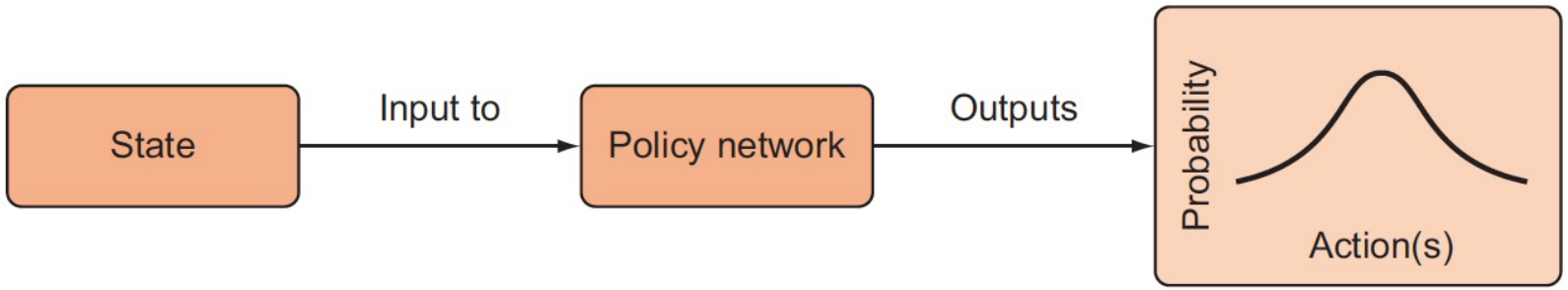


Figure 4.2 A policy network is a function that takes a state and returns a probability distribution over the possible actions.

Advantages of Policy Gradient Methods

- **No Action Strategy**: Directly samples actions, avoiding epsilon-greedy.
- **Simpler Training**: No need for experience replay or target networks like DQN.
- **Direct Optimization**: Improves the policy directly, rather than estimating state-action values.
- **Stochastic Policies**: Naturally accommodates environments requiring exploration and randomness.
- **Better for Continuous Actions**: Works well in continuous action spaces.
- **Faster Convergence**: Often more efficient in complex action spaces..

Stochastic policy gradient

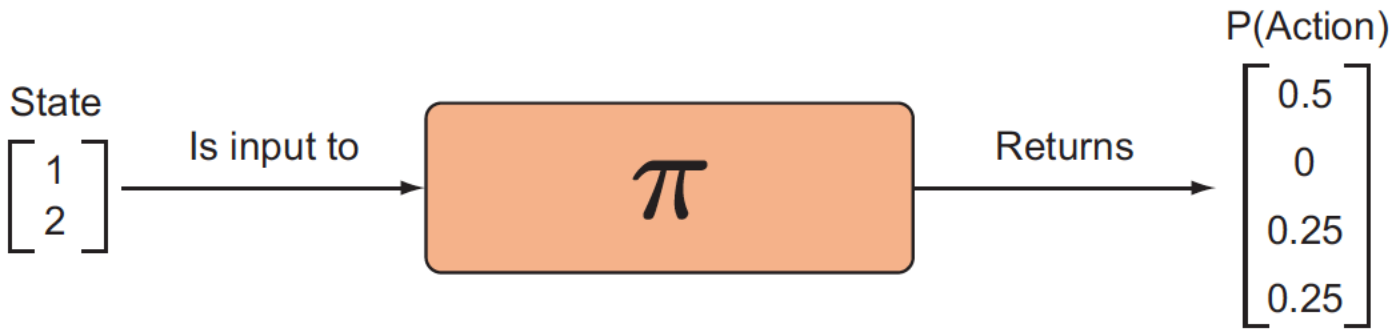
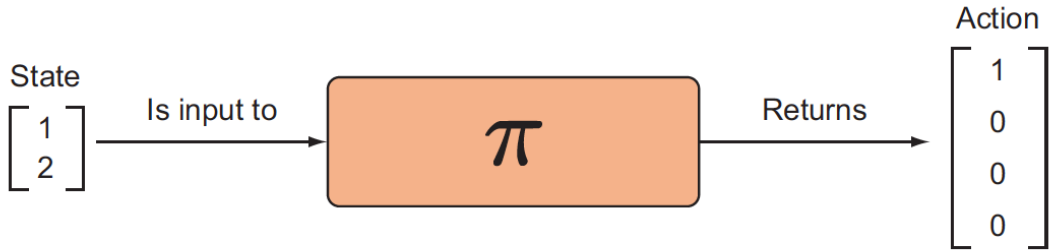


Figure 4.3 A stochastic policy function. A policy function accepts a state and returns a probability distribution over actions. It is *stochastic* because it returns a probability distribution over actions rather than returning a deterministic, single action.

deterministic policy function



- what the best action is secretly depends on a value function, but with a policy network we're trying to avoid computing these action values directly.

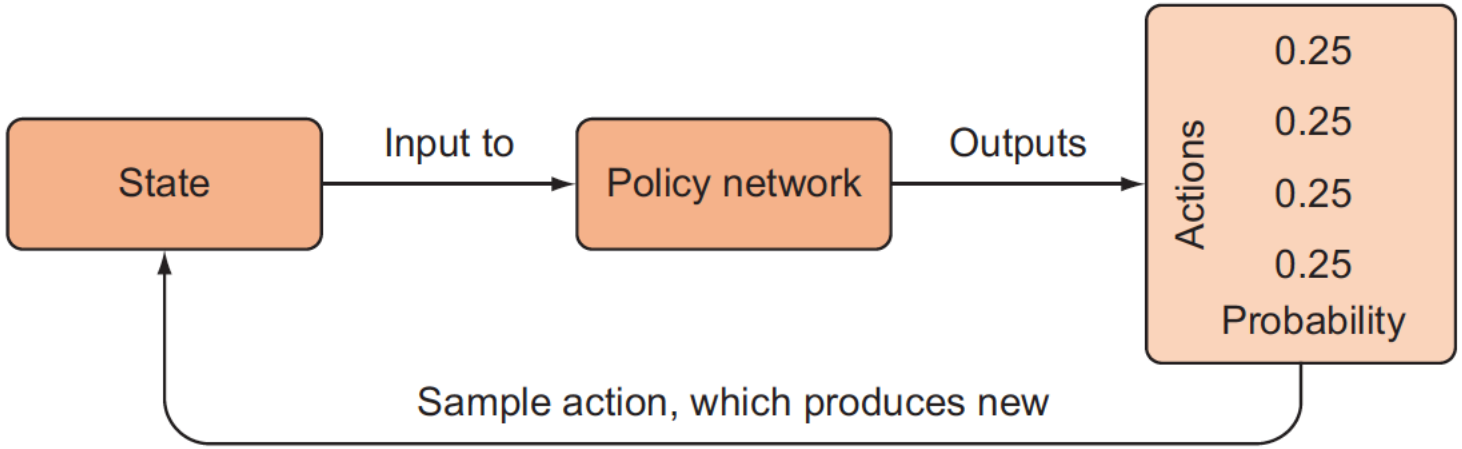


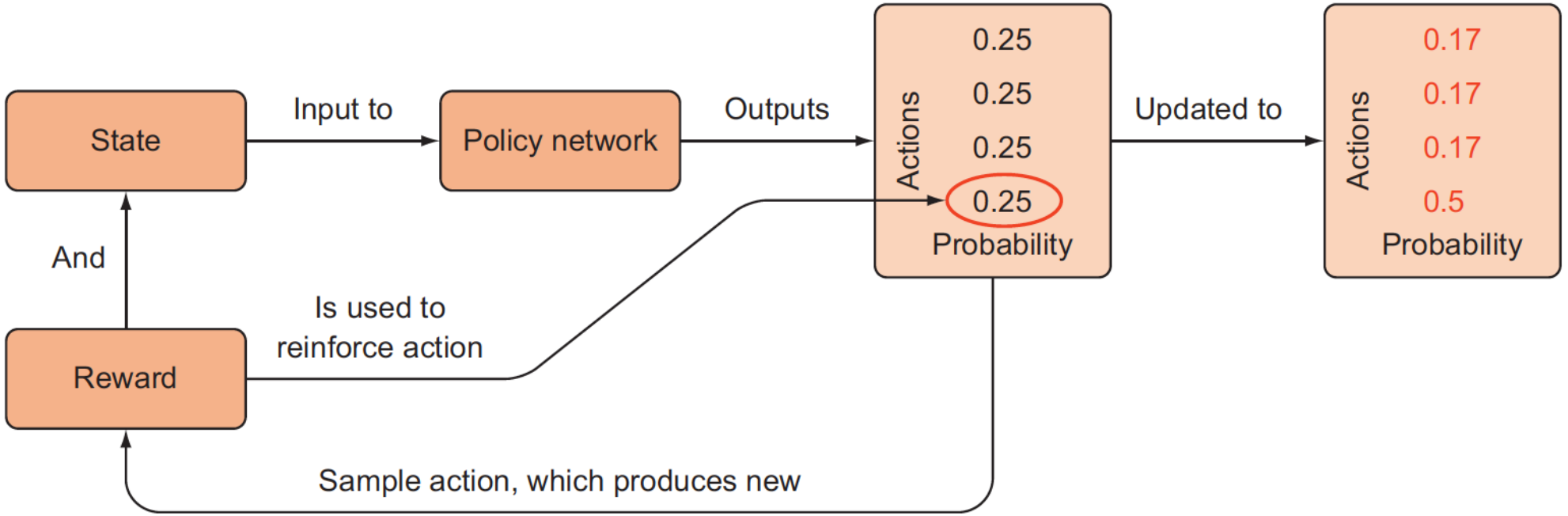
Figure 4.5 The general overview of policy gradients for an environment with four possible discrete actions. First we input the state to the policy network, which produces a probability distribution over the actions, and then we sample from this distribution to take an action, which produces a new state.

Policy gradient algorithm: Objective

- Episode:
- $\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ \ldots \ (S_{t-1}, A_{t-1}, R_t)$
- Example: $\varepsilon = (S_0, 3, -1), (S_1, 1, -1), (S_2, 3, +10)$
- What is there to learn from in this episode?
- Well, we won the game, indicated by the +10 reward in the last tuple, so our actions must have been "good" to some degree.
- Given the states we were in, we should encourage our policy network to make those actions more likely next time.
- We want to reinforce those actions that led to a high reward.

Action reinforcement

- To reinforce a good action, perform:
- minimize $-\ln\left(\pi_s(a \mid \theta)\right)$ $\Rightarrow$ [loss approaches 0 as $\pi_s(a \mid \theta)$ approaches 1]



Credit assignment

- Observation: The last action right before the reward deserves more credit for winning the game than does the first action
- Final objective function: Minimize $-\gamma_t \cdot G_t \cdot ln\, \pi_s(a \mid \theta)$
- $\gamma_t = \gamma_0^{(T-t)}$
  - γ_t is the discount factor
- $G_t = r_t + r_{t+1} \ldots + r_{T-1} + r_T$
  - G_t = total return, or future return, at time step t.
- It is the return we expect to collect from time step t until the end of the episode

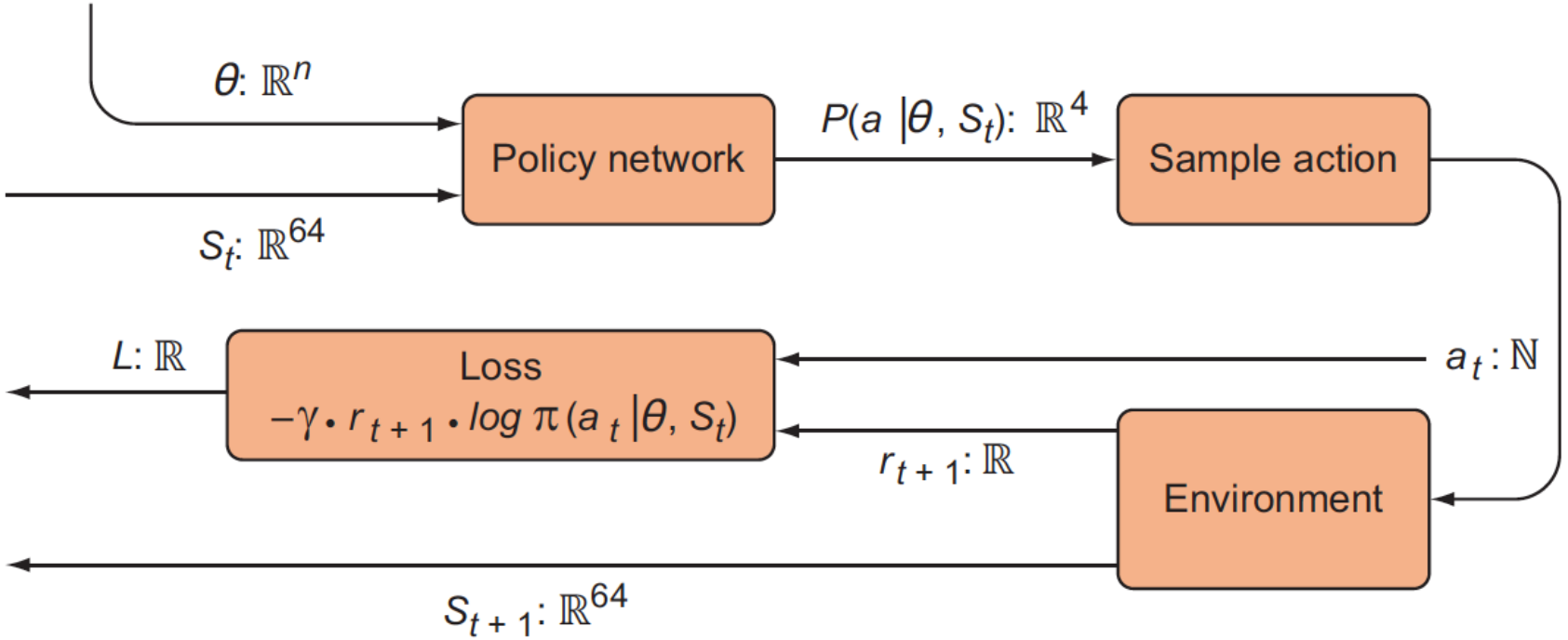


Figure 4.7 A string diagram for training a policy network for Gridworld. The policy network is a neural network parameterized by θ (the weights) that accepts a 64-dimensional vector for an input state. It produces a discrete 4-dimensional probability distribution over the actions. The sample action box samples an action from the distribution and produces an integer as the action, which is given to the environment (to produce a new state and reward) and to the loss function so we can reinforce that action. The reward signal is also fed into the loss function, which we attempt to minimize with respect to the policy network parameters.

REINFORCE Algorithm

Initialize learning rate α

Initialize weights θ of a policy network π_θ

for *episode* $= 0, \ldots, MAX\_EPISODE$ **do**

    Sample a trajectory $\tau = s_0, a_0, r_0, \ldots, s_T, a_T, r_T$

    Set $\nabla_\theta J(\pi_\theta) = 0$

    for $t = 0, \ldots, T$ **do**

        $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t$

        $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$

    end for

    $\theta = \theta + \alpha \nabla_\theta J(\pi_\theta)$

end for

ACTOR-CRITIC METHODS

Limitations of REINFORCE

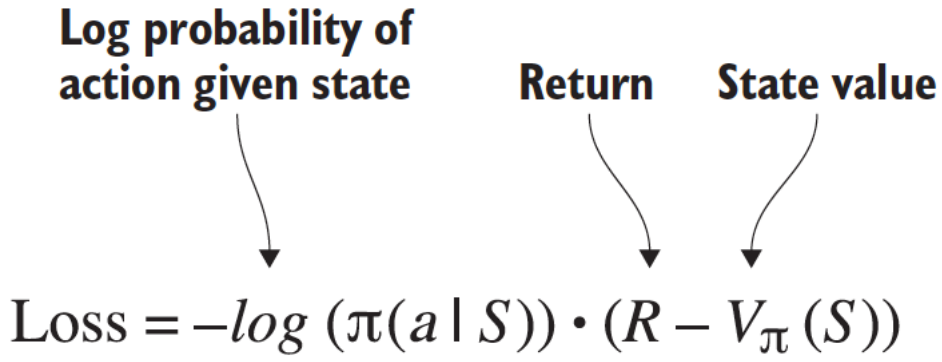
- **High Variance**:
  - Because of the stochastic nature; causes unstable and slow learning.
- **Inefficient Data Use**:
  - Trajectories are used only once for policy updates
- **Delayed Rewards**:
  - Updates rely on cumulative episode rewards, slowing down learning from individual actions.
- **Requires Full Episodes**:
  - Policy updates happen only after complete episodes, making it inefficient for long episodes.

Combining the value and policy function

- **DQN Example**: Predicts values for actions and chooses the highest-valued action.
- **Policy Gradient**: Reinforces actions based on rewards (positive or negative), learning the best actions indirectly.
- **Combined Approach**: Merges the strengths of Q-learning and policy gradients for better performance.
- Two challenges:
  - Improve the sample efficiency by updating more frequently.
  - Decrease the variance of the reward

Combined value-policy approach

- Use a value learner to reduce the variance in the rewards for training the policy.
- **Modification**: Instead of minimizing the REINFORCE loss based solely on the observed return, R, a baseline value is added to stabilize the loss.



$A(s,a)$ = R – V_π(S) is termed the advantage.
Intuitively, the advantage quantity tells you how much better an action is, relative to what you expected.

Actor-critic methods

- Actor: refers to the policy
- Critic: refers to the value function [Tell how good is the action]
- Since we're using R – V (S) to train the policy rather than just V (S), this is called advantage actor-critic

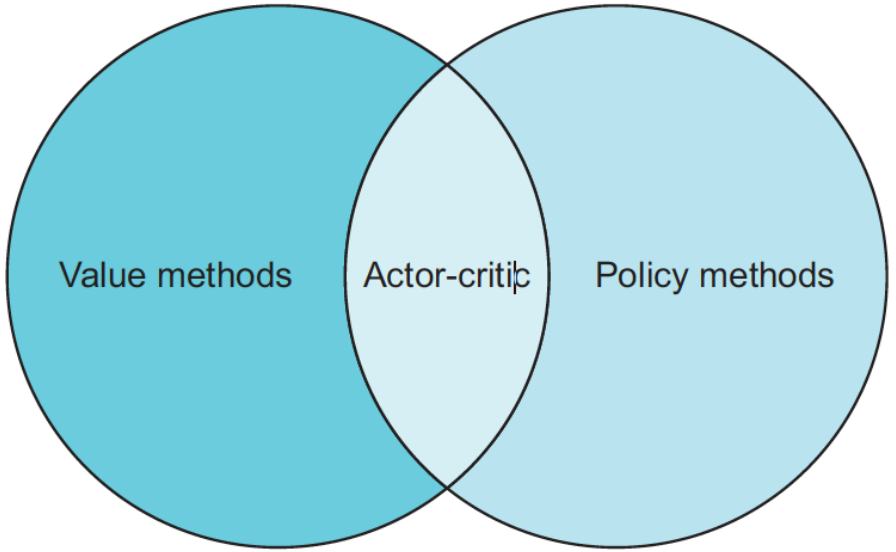


Figure 5.2 Q-learning falls under the category of value methods, since we attempt to learn action values, whereas policy gradient methods like REINFORCE directly attempt to learn the best actions to take. We can combine these two techniques into what's called an actor-critic architecture.

Overview of actor-critic models

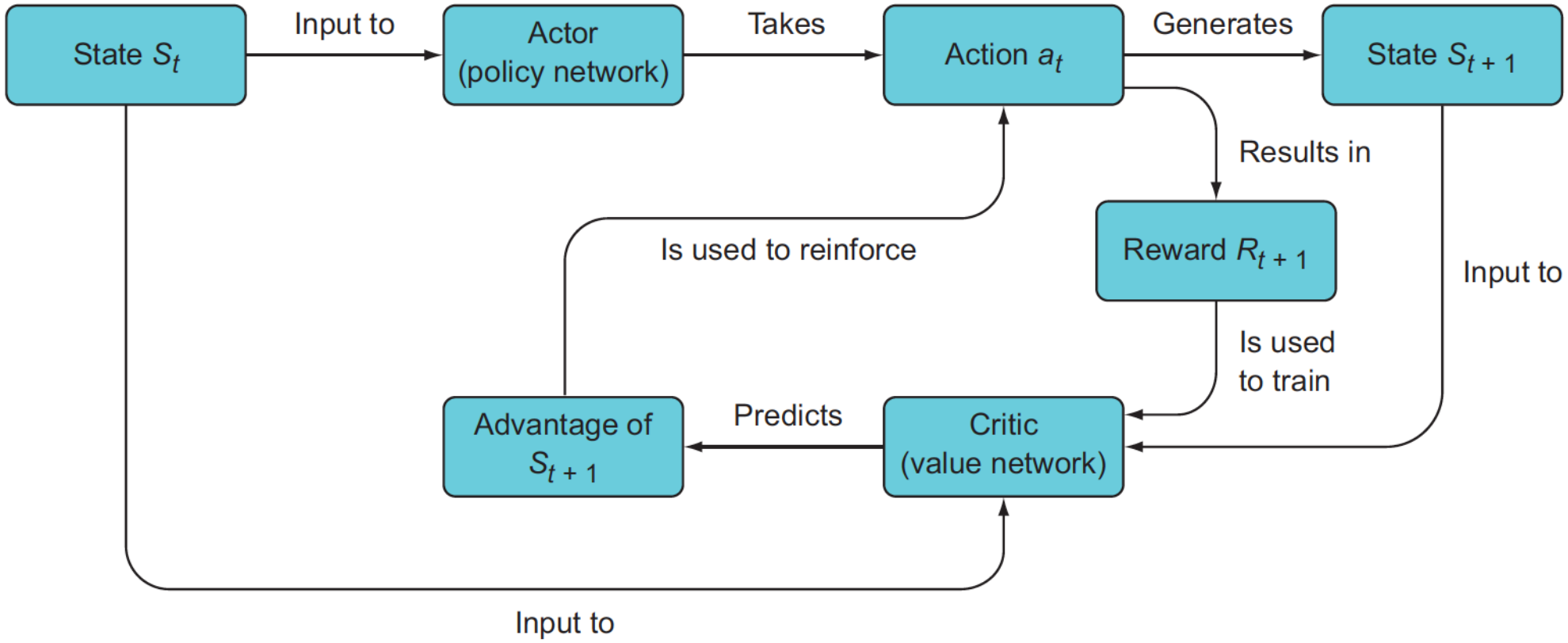


Figure 5.6 The general overview of actor-critic models. First, the actor predicts the best action and chooses the action to take, which generates a new state. The critic network computes the value of the old state and the new state. The relative value of S_{t+1} is called its advantage, and this is the signal used to reinforce the action that was taken by the actor.

Actor Loss (Policy Gradient Loss)

The actor's objective is to maximize the expected return by updating the policy. The loss function for the actor is typically based on the policy gradient theorem and is given by:

$$L_{\text{actor}} = -\log(\pi(a|s)) \cdot A(s,a)$$

Where:

- $\log(\pi(a|s))$ is the log probability of taking action a in state s under the current policy π.
- $A(s,a) = R - V(s)$ is the advantage, which measures how much better the action was compared to the expected value.

Critic Loss (Value Function Loss)

The critic's role is to estimate the value function $V(s)$, which is learned by minimizing the error between the estimated value $V(s)$ and the actual return R observed:

$$L_{\text{critic}} = \frac{1}{2}\left(R - V(s)\right)^2$$

This is a typical squared error loss that pushes the critic to correctly estimate the value function for state s.

Total Loss (Combined Loss)

In Advantage Actor-Critic (A2C), the total loss combines both the actor and critic losses:

$$L_{\text{total}} = L_{\text{actor}} + \alpha L_{\text{critic}}$$

Where α is a hyperparameter that balances the actor and critic losses, ensuring that both are trained simultaneously.

To summarize:

- **Actor loss** uses the advantage $A(s, a) = R - V(s)$ to adjust the policy.
- **Critic loss** minimizes the difference between the actual return R and the estimated value $V(s)$.

Summary

- **DQN**: Q-learning with deep neural networks
  - Learns Q-values for each action and selects the highest-value action.
  - **Techniques**: Experience Replay, Target Network for stability.
- **REINFORCE**: Policy gradient method
  - Directly optimizes policy by reinforcing actions based on cumulative rewards.
  - **Challenges**: High variance, inefficient data use.
- **Actor-Critic**: Combines policy (actor) and value (critic) learning
  - **Key Idea**: Actor selects actions, critic evaluates them to reduce variance.
  - **Benefits**: Faster and more stable learning than REINFORCE.

Acknowledgements

- Book: Deep Reinforcement Learning in Action

  by Brandon Brown and Alexander Zai