SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
TOPIC-1 : THE MARKOV PROPERTY

In the contextual bandit problem, our neural network led us to choose the best action given a state without reference to any other prior states. We just gave it the current state, and it produced the expected rewards for each possible action. This is an important property in reinforcement learning called the Markov property. A game (or any other control task) that exhibits the Markov property is said to be a Markov decision process (MDP). With an MDP, the current state alone contains enough information to choose optimal actions to maximize future rewards. Modeling a control task as an MDP is a key concept in reinforcement learning.

The MDP model simplifies an RL problem dramatically, as we do not need to take into account all previous states or actions—we don't need to have memory, we just need to analyze the present situation. Hence, we always attempt to model a problem as (at least approximately) a Markov decision process. The card game Blackjack (also known as 21) is an MDP because we can play the game successfully just by knowing our current state (what cards we have, and the dealer's one face-up card).

To test the understanding of the Markov property, consider each control problem or decision task in the following list and see if it has the Markov property or not:

Driving a car
Deciding whether to invest in a stock or not
Choosing a medical treatment for a patient
Diagnosing a patient's illness
Predicting which team will win in a football game
Choosing the shortest route (by distance) to some destination
Aiming a gun to shoot a distant target

Here are our answers and brief explanations:

Driving a car can generally be considered to have the Markov property because you don't need to know what happened 10 minutes ago to be able to optimally drive your car. You just need to know where everything is right now and where you want to go.

Deciding whether to invest in a stock or not does not meet the criteria of the Markov property since you would want to know the past performance of the stock in order to make a decision.

Choosing a medical treatment seems to have the Markov property because you don't need to know the biography of a person to choose a good treatment for what ails them right now.

In contrast, diagnosing (rather than treating) would definitely require knowledge of past states. It is often very important to know the historical course of a patient's symptoms in order to make a diagnosis.

Predicting which football team will win does not have the Markov property, since, like the stock example, you need to know the past performance of the football teams to make a good prediction.

Choosing the shortest route to a destination has the Markov property because you just need to know the distance to the destination for various routes, which doesn't depend on what happened yesterday.

Aiming a gun to shoot a distant target also has the Markov property since all you need to know is where the target is, and perhaps current conditions like wind velocity and the particulars of your gun. You don't need to know the wind velocity of yesterday.

We hope you can appreciate that for some of those examples you could make arguments for or against it having the Markov property. For example, in diagnosing a patient, you may need to know the recent history of their symptoms, but if that is documented in their medical record and we consider the full medical record as our current state, then we've effectively induced the Markov property. This is an important thing to keep in mind: many problems may not naturally have the Markov property, but often we can induce it by jamming more information into the state.

DeepMind's deep Q-learning (or deep Q-network) algorithm learned to play Atari games from just raw pixel data and the current score. Do Atari games have the Markov property? Not exactly. In the game Pacman, if our state is the raw pixel data from our current frame, we have no idea if the enemy a few tiles away is approaching us or moving away from us, and that would strongly influence our choice of actions to take. This is why DeepMind's implementation actually feeds in the last four frames of gameplay, effectively changing a non-MDP into an MDP. With the last four frames, the agent has access to the direction and speed of all players.

Figure gives a lighthearted example of a Markov decision process using all the concepts we've discussed so far. You can see there is a three-element state space S = {crying baby, sleeping baby, smiling baby}, and a two-element action space A = {feed, don't feed}. In addition, we have transition probabilities noted, which are maps from an action to the probability of an outcome state (we'll go over this again in the next section). Of course, in real life, you as the agent have

no idea what the transition probabilities are. If you did, you would have a model of the environment. As you'll learn later, sometimes an agent does have access to a model of the environment, and sometimes not. In the cases where the agent does not have access to the model, we may want our agent to learn a model of the environment (which may just approximate the true, underlying model).

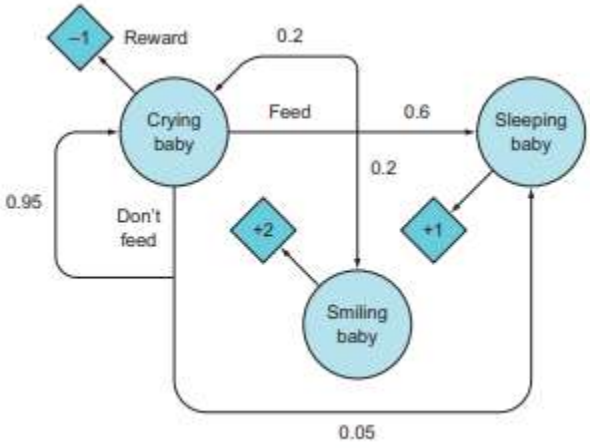


Figure A simplified MDP diagram with three states and two actions. Here we model the parenting decision process for taking care of an infant. If the baby is crying, we can either administer food or not, and with some probability the baby will transition into a new state, and we'll receive a reward of -1, +1, or +2 (based on the baby's satisfaction).

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
TOPIC-2 : PREDICTING FUTURE REWARDS: VALUE AND POLICY FUNCTIONS

A reinforcement learning algorithm essentially constructs an agent, which acts in some environment. The environment is often a game, but is more generally whatever process produces states, actions, and rewards. The agent has access to the current state of the environment, which is all the data about the environment at a particular time point, $s_t \in S$. Using this state information, the agent takes an action, $a_t \in A$, which may deterministically or probabilistically change the environment to be in a new state, $s_t +1$.

The probability associated with mapping a state to a new state by taking an action is called the transition probability. The agent receives a reward, rt, for having taken action in state st leading to a new state, st +1. And we know that the ultimate goal of the agent (our reinforcement learning algorithm) is to maximize its rewards. It's really the state transition, st → st +1, that produces the reward, not the action per se, since the action may probabilistically lead to a bad state.

Policy functions

A policy, π, is the strategy of an agent in some environment. For example, the strategy of the dealer in Blackjack is to always hit until they reach a card value of 17 or greater. It's a simple fixed strategy. In our n-armed bandit problem, our policy was an epsilon-greedy strategy. In general, a policy is a function that maps a state to a probability distribution over the set of possible actions in that state.

The policy function defines the behavior of the agent. It maps states (`s`) to actions (`a`), indicating what action the agent should take in a given state. The policy can be deterministic or stochastic:

- **Deterministic policy:** Always returns the same action for a state.
- **Stochastic policy:** Returns a probability distribution over actions for each state.

Example:

In a game of chess:

- The **policy function** could dictate that if the agent's knight is under threat, it should move the knight to a safe square.
- In a **deterministic policy**, it will always move the knight to the same safe square.
- In a **stochastic policy**, it might choose between a few safe squares with different probabilities.

- **Example**: In a grid world where an agent can move up, down, left, or right, a simple policy function could be:
    - **Deterministic Policy**: If the agent is in state (2, 3), it always moves right to (2, 4).
    - **Stochastic Policy**: If the agent is in state (2, 3), it moves right with a probability of 0.7, down with a probability of 0.2, and left with a probability of 0.1.

Table: The policy function

Math	English
π, $s \rightarrow Pr(A \mid s)$, where $s \in S$	A policy, π, is a mapping from states to the (probabilistically) best actions for those states.

In the mathematical notation, s is a state and Pr(A | s) is a probability distribution over the set of actions A, given state s. The probability of each action in the distribution is the probability that the action will produce the greatest reward.
Example: In a simple grid world, the policy function might map each grid cell (state) to a probability distribution over the possible actions (move up, down, left, or right). For instance, π(move right | cell_1) = 0.7, π(move up | cell_1) = 0.3, and π(move down | cell_1) = 0 could indicate a 70% chance of moving right and 30% chance of moving up from cell 1, with no chance of moving down.

Optimal policy
The policy is the part of our reinforcement learning algorithm that chooses actions given its current state. The optimal policy—it's the strategy that maximizes rewards.

The optimal policy is the best possible policy that maximizes the expected cumulative reward for the agent. It represents the most efficient way of acting in each state to achieve the highest possible return.

Example:

In a self-driving car system:

- The **optimal policy** might dictate the best speed and direction to maximize safety and minimize time. If the car needs to stop at a red light, the optimal policy would instruct the car to stop.
- In a game of chess, the **optimal policy** would be the strategy that consistently wins against any opponent, maximizing the expected reward (points) over many games.
- An agent following the **optimal policy** in chess might sacrifice a pawn in one state to win the game in a later state, maximizing its cumulative reward (winning the game).

Table: The optimal policy

Math	English
$\pi^* = \text{argmax}\ E(R \mid \pi)$,	If we know the expected rewards for following any possible policy, π, the optimal policy, π^*, is a policy that, when followed, produces the maximum possible rewards.

Remember, a particular policy is a map or function, so we have some sort of set of possible policies; the optimal policy is just an argmax (which selects the maximum) over this set of possible policies as a function of their expected rewards.

The whole goal of a reinforcement learning algorithm (our agent) is to choose the actions that lead to the maximal expected rewards. But there are two ways we can train our agent to do this:
Directly—We can teach the agent to learn what actions are best, given what state it is in.
Indirectly—We can teach the agent to learn which states are most valuable, and then to take actions that lead to the most valuable states.

This indirect method leads us to the idea of value functions

Value function
Value functions are functions that map a state or a state-action pair to the expected value (the expected reward) of being in some state or taking some action in some state. You may recall from statistics that the expected reward is just the long-term average of rewards received after being in some state or taking some action. When we speak of the value function, we usually mean a state-value function.

Table:The state-value function

Math	English
$V_\pi: s \rightarrow E(R \mid s, \pi)$,	A value function, V_π is a function that maps a state, s, to the expected rewards, given that we start in state s and follow some policy, π.

This is a function that accepts a state, s, and returns the expected reward of starting in that state and taking actions according to our policy, π. It may not be immediately obvious why the value function depends on the policy. Consider that in our contextual bandit problem, if our policy was to choose entirely random actions (i.e., sample actions from a uniform distribution), the value (expected reward) of a state would probably be pretty low, since we're definitely not choosing the best possible actions. Instead, we want to use a policy that is not a uniform distribution over the actions, but is the probability distribution that would produce the maximum rewards when sampled. That is, the policy is what determines observed rewards, and the value function is a reflection of observed rewards.

The state-value function measures how good it is for an agent to be in a particular state, considering the expected cumulative rewards the agent will collect from that state onward, assuming it follows a certain policy (π).

Example:

Imagine an agent navigating a grid-world where the goal is to reach a treasure:

- If the agent is only 1 step away from the treasure, the **state-value** of that position would be very high because the agent is close to receiving the reward (e.g., `V(s) = +10`).
- If the agent is 10 steps away from the treasure, the **state-value** of that position would be lower since the reward is farther away (e.g., `V(s) = +2`).

In a game of chess:

- The **state-value** of a board position could represent how favorable that position is for winning the game. A checkmate position would have a very high value, while being on the verge of losing would have a very low value.

Q function

In our first n-armed bandit problem, you were introduced to state-action-value functions. These functions often go by the name Q function or Q value, which is where deep Q-learning comes from, since, as you'll see in the next chapter, deep learning algorithms can be used as Q functions.

The **action-value function** (also called the **Q-function**) measures how good it is for the agent to take a specific action (`a`) in a given state (`s`), considering the expected cumulative rewards from that point onward if it continues to follow a certain policy (π).

Table: The action-value (Q) function

Math	English
$Q_\pi(s\|a) \rightarrow E(R\|a,s,\pi)$.	Q_π is a function that maps a pair, (s, a), of a state, s, and an action, a, to the expected reward of taking action a in state s, given that we're using the policy (or "strategy") π.

Example:

In the same grid-world example:

- **If the agent is 1 step away from the treasure and has two actions, move left and move right:**
  - **If move right takes the agent directly to the treasure, the action-value Q(s, move right) will be very high (e.g., Q(s, move right) = +10).**
  - **If move left takes the agent farther from the treasure, the action-value Q(s, move left) will be lower (e.g., Q(s, move left) = +3).**

In a self-driving car system:

- **In a state where the car is approaching a traffic light that is about to turn red:**
  - **The action-value of applying the brakes (Q(s, brake)) might be high because stopping will avoid a fine or an accident.**
  - **The action-value of accelerating (Q(s, accelerate)) will be low since it may cause the car to run the red light.**

Summary

- **Policy** (`π(s)`) decides what actions to take.
- **Optimal policy** (`π*(s)`) maximizes long-term rewards.
- **Value function** (`V(s)`) tells how good it is to be in a state.
- **Q-value function** (`Q(s, a)`) tells how good it is to take a specific action in a state.

By combining policy and value functions, RL algorithms train agents to maximize rewards, leading to improved performance in complex tasks like game playing, decision-making, and robotic control.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
TOPIC-3 : THE Q FUNCTION

In reinforcement learning, the Q function plays a central role in determining the best actions to take in a given state. The Q function, denoted as $Q\pi(s,a)$, represents the expected reward of taking action 'a' in state 's'and then following a specific policy thereafter.

Key Concepts:

1. **State (s):**

The state is the information that our agent receives and uses to make a decision about what action to take. It could be the raw pixels of a video game, sensor data from an autonomous vehicle, or, in the case of Gridworld, a tensor representing the positions of all the objects on the grid.

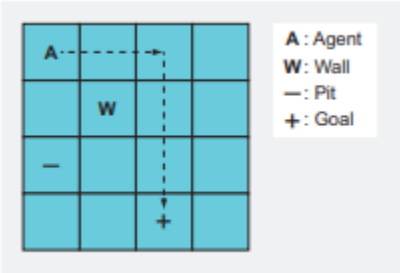


Figure: This is a simple Gridworld game setup. The agent (A) must navigate along the shortest path to the goal tile (+) and avoid falling into the pit (–).

2. **Policy (π):**

The policy, denoted π, is the strategy our agent follows when provided a state. For example, a policy in Blackjack might be to look at our hand (the state) and hit or stay randomly. Although this would be a terrible policy, the important point to stress is that the policy confers which actions we take. A better policy would be to always hit until we have 19.

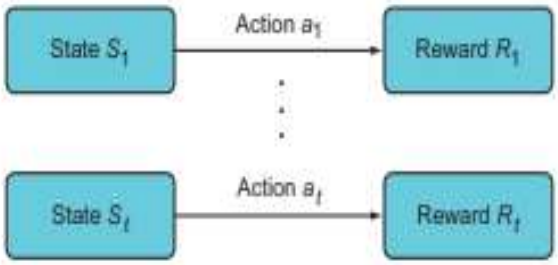
3. **Reward (R):**

The reward is the feedback our agent gets after taking an action, leading us to a new state. For a game of chess, we could reward our agent +1 when it performs an action that leads to a

checkmate of the other player and –1 for an action that leads our agent to be checkmated. Every other state could be rewarded 0, since we do not know if the agent is winning or not.

4. **Action (a):**

Our agent makes a series of actions based upon its policy π, and repeats this process until the episode ends, thereby we get a succession of states, actions and the resulting rewards.



State value or Value function:
We call the weighted sum of the rewards while following a policy from the starting state S1 the value of that state, or a state value. We can denote this by the value function Vπ(s), which accepts an initial state and returns the expected total reward.

$$V_\pi(s) = \sum_{i=1}^{t} w_i R_i = w_1 R_1 + w_2 R_2 + \ldots + w_t R_t$$

The coefficients w1, w2, etc., are the weights we apply to the rewards before summing them. For example, we often want to weight more recent rewards greater than distant future rewards. This weighted sum is an expected value, a common statistic in many quantitative fields, and it's often concisely denoted E[R |π,s], read as "the expected rewards given a policy π and a starting state s."

Q-Function:
Similarly, there is an action-value function, Qπ(s,a), that accepts a state S and an action A and returns the value of taking that action given that state; in other words, E[R |π,s,a]. Some RL algorithms or implementations will use one or the other.

The action-value (Q) function

Math	English
$Q_\pi(s\|a) \rightarrow E(R\|a,s,\pi)$,	Q_π is a function that maps a pair, (s, a), of a state, s, and an action, a, to the expected reward of taking action a in state s, given that we're using the policy (or "strategy") π.

Importantly, if we base our algorithm on learning the state values (as opposed to action values), we must keep in mind that the value of a state depends completely on our policy, π. Using Blackjack as an example, if we're in the state of having a card total of 20, and we have two possible actions, hit or stay, the value of this state is only high if our policy says to stay when we have 20. If our policy said to hit when we have 20, we would probably bust and lose the game, so the value of that state would be low. In other words, the value of a state is equivalent to the value of the highest action taken in that state.

Q-Learning:

Q-learning is a method to learn the optimal Q-function, which helps the agent find the best actions without needing to know the environment's dynamics. This method iteratively updates the Q-values based on the agent's experiences, moving toward optimal decision-making over time.

In 2013, DeepMind's breakthrough paper "Playing Atari with Deep Reinforcement Learning" showed how to use a deep neural network to approximate the Q-function (Deep Q-Network or DQN), significantly advancing reinforcement learning applications.

Applications of Q-Function:

- **Games like Gridworld:** The Q function helps the agent learn to navigate efficiently by learning the optimal path based on rewards.
- **Robotics:** Helps robots determine the best actions to complete tasks.
- **Autonomous Driving:** Assists self-driving cars in learning optimal driving behaviors based on the rewards from actions.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
TOPIC-4 : NAVIGATING WITH Q-LEARNING: Q-LEARNING? AND TACKLING GRIDWORLD

1. Q-learning

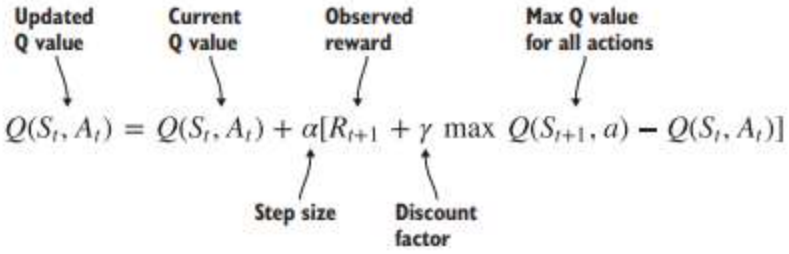
Q-learning is a model-free reinforcement learning algorithm aimed at learning the optimal action-selection policy for an agent interacting with an environment. It focuses on action-value functions that evaluate the expected utility of actions taken in particular states. The central goal is to find the Q-function that maps each state-action pair to a predicted future reward. The formula for Q-learning is based on updating the Q-value estimates over time as more rewards are observed from different actions and states.

Key Concepts in Q-learning:

- State (s): Represents the current situation or environment the agent is in.
- Action (a): Refers to the choices the agent can make in the environment.
- Q-value: Represents the expected future reward for taking an action in a given state.
- Update Rule: The algorithm updates Q-values based on the difference between the predicted and observed rewards. The general update rule is:

Table 3.1 Q-learning update rule

Math



Pseudocode

```
def get_updated_q_value(old_q_value, reward, state, step_size, discount):
     term2 = (reward + discount * max([Q(state, action) for action in
             actions])
     term2 = term2 - old_q_value
     term2 = step_size * term2
     return (old_q_value + term2)
```

English

The Q value at time *t* is updated to be the current predicted Q value plus the amount of value we expect in the future, given that we play optimally from our current state.

Pseudocode

def get_updated_q_value(old_q_value, reward, state, step_size, discount):

term2 = (reward + discount * max([Q(state, action) for action in actions])

term2 = term2 – old_q_value

term2 = step_size * term2

return (old_q_value + term2)

English

The Q value at time t is updated to be the current predicted Q value plus the amount of value we expect in the future, given that we play optimally from our current state.

In essence, Q-learning works by allowing the agent to evaluate its actions over time by learning from the rewards (or penalties) it receives from the environment. The algorithm uses a combination of exploration (choosing actions at random to discover new states) and exploitation (choosing the action with the highest known Q-value).

2. Tackling Gridworld

The Gridworld problem is a classic example used to demonstrate Q-learning. In this problem, an agent must navigate a grid (e.g., a 4x4 grid) to reach a goal while avoiding obstacles and minimizing negative rewards. The environment is typically discrete, meaning actions are taken in a step-by-step manner, and the game progresses one turn at a time.

Steps in Tackling Gridworld with Q-learning:

1. Initialize the Q-values: We start the game in some state that we'll call St. The state includes all the information about the game that we have. For our Gridworld example, the game state is represented as a $4 \times 4 \times 4$ tensor. Initially, all state-action pairs are given a Q-value of zero. The agent does not know which actions will lead to rewards or penalties.
2. Start the Game: We feed the St data and a candidate action into a deep neural network (or some other fancy machine-learning algorithm) and it produces a prediction of how valuable taking that action in that state is.

Figure 3.2 The Q function could be any function that accepts a state and action and returns the value (expected rewards) of taking that action given that state.

   Remember, the algorithm is not predicting the reward we will get after taking a particular action; it's predicting the expected value (the expected rewards), which is the long-term average reward we will get from taking an action in a state and then continuing to behave according to our policy π. We do this for several (perhaps all) possible actions we could take in this state.

   The agent starts in a random state within the grid, and its task is to reach the goal. At each step:

   - The agent observes the current state s_t.
   - It selects an action a_t, either based on past experience (exploitation) or randomly (exploration).

3. Update the Q-value: We take an action, perhaps because our neural network predicted it is the highest value action or perhaps we take a random action. We'll label the action At. We are now in a new state of the game, which we'll call St+1, and we receive or observe a reward, labelled Rt+1. We want to update our learning algorithm to reflect the actual reward we received, after taking the action it predicted was the best. Perhaps we got a negative reward or a really big reward, and we want to improve the accuracy of the algorithm's predictions.

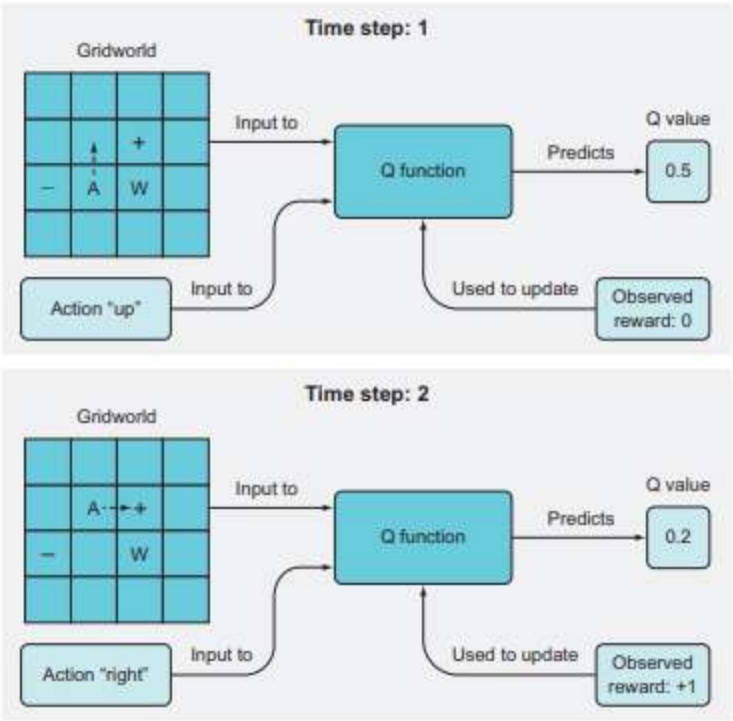


Figure:Schematic of Q-learning with Gridworld. The Q function accepts a state and an action, and returns the predicted reward (value) of that state action pair. After taking the action, we observe the reward, and using the update formula, we use this observation to update the Q function so it makes better predictions.

After taking an action, the agent observes the new state st+1 and receives a reward Rt+1. The Q-value for the state-action pair is updated using the formula provided earlier.

4. Repeat: Now we run the algorithm using St+1 as input and figure out which action our algorithm predicts has the highest value. We'll call this value Q(St+1,a). To be clear, this is a single value that reflects the highest predicted Q value, given our new state and all possible actions.

   The agent continues to interact with the grid, updating its Q-values based on the rewards and penalties it encounters, aiming to learn the optimal policy that maximizes future rewards.

5. Now we have all the pieces we need to update the algorithm's parameters. We'll perform one iteration of training using some loss function, such as meansquared error, to minimize the difference between the predicted value from our algorithm and the target prediction of $Q(St,At) + \alpha *[Rt+1 + \gamma *maxQ(St+1,A) – Q(St,At)]$.

Game Dynamics:

- The agent is trained with the goal of reaching a specific target location on the grid.
- Rewards: Positive rewards are given for reaching the goal, and negative rewards are given for hitting obstacles or wasting moves.
- The algorithm starts with no knowledge about the grid or where the goal is, but over time it learns the most efficient paths.

Example:

If the agent is in state St, it might take one of several actions (move up, down, left, or right). Based on the reward and the new state, it updates its Q-value. After many episodes of the game, the agent will have learned a set of optimal actions (a policy) to get from any starting state to the goal while accumulating the highest possible reward.

By using this method, Q-learning can teach the agent to learn from scratch—it starts without any knowledge of the environment but becomes more adept at reaching the goal as it receives feedback from its actions.

In conclusion, Q-learning allows an agent to explore and learn the best policy for a given environment through trial and error, with minimal prior knowledge. The Gridworld example highlights how this learning algorithm can be applied to solve a navigation problem in a simple grid-based environment, which serves as a stepping stone for more complex reinforcement learning tasks

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
TOPIC-5 : NAVIGATING WITH Q-LEARNING: HYPERPARAMETERS AND DISCOUNT FACTOR

In reinforcement learning, particularly in Q-learning, hyperparameters play a critical role in shaping the learning process and determining the effectiveness of the algorithm. Two of the most important hyperparameters are the **learning rate** and the **discount factor**. These parameters influence how quickly an agent learns from its environment and how it evaluates future rewards. Below is a detailed explanation of these hyperparameters, focusing on their role in Q-learning.

1. Hyperparameters

Hyperparameters are variables that control the learning process of the algorithm but are not updated through the training process itself. They are set before the learning begins and remain fixed during the training phase. In Q-learning, the two most important hyperparameters are the **learning rate (α)** and the **discount factor (γ)**.

a) Learning Rate (α)

The **learning rate (α)** controls how much the Q-learning algorithm adjusts the Q-value estimates after each update. It essentially dictates how responsive the agent is to new experiences.

- **Low learning rate**: If α is small, the algorithm will only make small updates to the Q-values after each learning step. This leads to slower learning, but it helps in stabilizing the learning process by making incremental improvements over time.
- **High learning rate**: If α is large, the algorithm will make significant updates to the Q-values after every action. This can speed up learning but may cause instability, as the agent may overreact to single experiences, especially if they are outliers or noisy.

In general, the learning rate is a trade-off between **convergence speed** and **stability**. A very high learning rate can lead to erratic learning behavior, while a very low learning rate might slow down the learning process to an unacceptable degree.

2. Discount Factor (γ)

The **discount factor (γ)** is a parameter that determines how much the agent values future rewards over immediate rewards. In Q-learning, future rewards are discounted by this factor, which ranges between 0 and 1.

a) The Role of γ

The discount factor serves as a weight applied to future rewards to prioritize them relative to immediate rewards. It allows the agent to make decisions that not only maximize the **immediate reward** but also consider the **long-term value** of taking an action.

For example:

- A **γ value close to 1** means the agent will heavily consider future rewards. The agent focuses on long-term gains and is willing to sacrifice immediate benefits for better outcomes in the future.
- A **γ value close to 0** means the agent will prioritize short-term rewards, largely ignoring the potential long-term consequences of its actions. The agent becomes **myopic**, focusing only on immediate rewards.

The discount factor controls the balance between **immediate gratification** and **long-term strategy**. A well-tuned γ ensures that the agent achieves an optimal balance, avoiding both shortsighted decisions and overly deferred rewards.

3. The Discount Factor in Practice

To illustrate how the discount factor works, consider a scenario where the agent faces two action choices:

- **Action A**: It results in an immediate reward of 0, followed by a reward of +1 in the next time step.
- **Action B**: It provides an immediate reward of +1, followed by a reward of 0.

Even though both actions lead to the same cumulative reward of +1, the timing of the rewards will influence the agent's decision depending on the value of γ. If γ is close to 1, the agent might view both actions as nearly equal, since it heavily weighs the future reward. If γ is closer to 0, the agent will favor **Action B**, which provides the immediate reward, because future rewards have less importance.

Example:

Consider an agent in a gridworld environment:

- If $\gamma=1$, the agent tries to optimize for future rewards, considering all steps it will take throughout the game.
- If $\gamma=0.5$, the agent discounts future rewards by half, focusing more on immediate rewards while still giving some weight to future outcomes.
- If $\gamma=0$, the agent acts **greedily**, valuing only the immediate next step and not caring about future consequences at all.

4. Hyperparameters in Gridworld

In the Gridworld example, where the goal is to navigate a grid to reach a reward while avoiding penalties, the setting of α and γ plays a crucial role. If the learning rate is set too high, the agent might quickly settle on suboptimal paths because it is learning too fast. On the other hand, if the discount factor is too low, the agent might fail to consider future rewards, choosing paths that give immediate but less overall reward.

Hyperparameter Tuning

Unfortunately, there is no fixed rule for setting hyperparameters like α and γ. These must be **tuned empirically** by running the algorithm with different values and observing performance. In complex environments, this can involve running many experiments to find the optimal balance.

In Q-learning, the **learning rate** and **discount factor** are critical hyperparameters that significantly impact how an agent learns from its environment. The **learning rate** controls how quickly the agent updates

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
THEORY & LAB TOPIC-6 : NAVIGATING WITH Q-LEARNING: BUILDING THE NETWORK

This section focuses on the process of constructing a neural network that will serve as the Q function for the Gridworld game. The network will take the game state as input and output Q values for each possible action.

Overview of the Neural Network Architecture

The neural network designed for the Gridworld game consists of three layers:

- Input Layer: This layer accepts the game state represented as a vector. Given that the game state is a $4 \times 4 \times 4$ tensor, it can be flattened into a 1-dimensional vector of length 64 (since $4 \times 4 \times 4 = 64$).
- Hidden Layer: The hidden layer processes the input data. In our implementation, we will use a single hidden layer with 150 neurons. However, you are encouraged to experiment with the number of hidden layers and their sizes to potentially improve performance.
- Output Layer: The output layer consists of 4 neurons, each corresponding to one of the possible actions in the game (up, down, left, right). The output values represent the Q values for each action given the current state.

Neural Network as the Q Function

The network uses Q-learning to predict the best actions by maximizing long-term rewards. The neural network learns from the game states, which are represented as tensors of dimensions 4x4x4. Each matrix slice in the tensor represents the position of objects like the player, the goal, the pit, and the wall.

The game states are converted into a flat 64-length vector, which is fed into the neural network. Based on this input, the network outputs a 4-length vector representing the Q-values for the possible actions: up, down, left, or right.

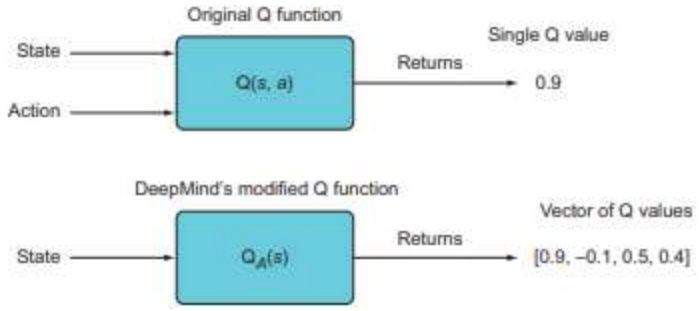


Figure: The original Q function accepts a state-action pair and returns the value of that state-action pair—a single number. DeepMind used a modified vector-valued Q function that accepts a state and returns a vector of state-action values, one for each possible action given the input state. The vector-valued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

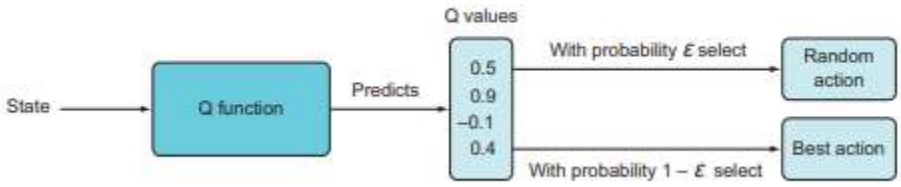


Figure: In an epsilon-greedy action selection method, we set the epsilon parameter to some value, e.g., 0.1, and with that probability we will randomly select an action (completely ignoring the predicted Q values) or with probability 1 – epsilon = 0.9, we will select the action associated with the highest 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.

Implementation Steps

To build the neural network, follow these steps:

Import Necessary Libraries: Ensure you have PyTorch installed, as we will be using it for building the neural network.

```
import torch
import torch.nn as nn
import torch.optim as optim
```

Define the Neural Network Class: Create a class that inherits from nn.Module. This class will define the layers and the forward pass.

```
class QNetwork(nn.Module):
    def __init__(self):
        super(QNetwork, self).__init__()
        self.input_layer = nn.Linear(64, 150)  # Input layer
        self.hidden_layer = nn.Linear(150, 4)   # Output layer

    def forward(self, x):
        x = torch.relu(self.input_layer(x))  # Activation function for hidden layer
        x = self.hidden_layer(x)             # Output layer
        return x
```

Instantiate the Network: Create an instance of the QNetwork class.

```
q_network = QNetwork()
```

Define the Loss Function and Optimizer: Choose a loss function and an optimizer for training the network.

```
criterion = nn.MSELoss()  # Mean Squared Error Loss
optimizer = optim.Adam(q_network.parameters(), lr=0.001)  # Adam optimizer
```

Training the Network

To train the network, you will need to implement a training loop that involves:

- Feeding the game state into the network.
- Calculating the Q values.
- Computing the loss based on the expected Q values.
- Performing backpropagation to update the network weights.

Building a neural network for the Gridworld game involves defining the architecture, implementing the forward pass, and setting up the training process. By experimenting with different configurations and training strategies, you can enhance the performance of your Q-learning agent.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
THEORY & LAB TOPIC-7 : NAVIGATING WITH Q-LEARNING: INTRODUCING THE GRIDWORLD GAME ENGINE

In this section, we will introduce the Gridworld game engine, which is used to implement a reinforcement learning (RL) environment for training models. The Gridworld engine serves as a simple environment where an agent can learn to navigate and interact with the game board.

1. Overview of the Gridworld Game

The Gridworld environment is a 4x4 grid that represents the game board. In this environment, the player or agent navigates through the grid to reach a goal while avoiding obstacles like walls and pits. The engine supports three different initialization modes for the game board:

- **Static Mode**: All objects, including the player, are initialized at fixed, predefined locations.
- **Player Mode**: The player starts at a random position on the grid, while the other objects remain in predefined locations.
- **Random Mode**: Both the player and other objects (goal, pits, and walls) are initialized at random positions. This mode makes it more challenging for the learning algorithm as there are no fixed patterns to rely on.

2. Setting Up the Game Engine

To begin using the Gridworld engine, import the necessary module from the provided Python file. The file for the Gridworld engine is included in the GitHub repository associated with this project.

Here's how to create an instance of the Gridworld game engine:

python
Copy code

```
from Gridworld import Gridworld
game = Gridworld(size=4, mode='static')
```

- The `size=4` argument specifies the dimensions of the grid (a 4x4 grid).

- The `mode` argument specifies how the game is initialized, with options for `static`, `player`, or `random` mode as explained above.

3. Game Interaction

Once the game instance is created, you can interact with the environment using the following methods:

- **`game.display()`**: Displays the current state of the game board.
- **`game.makeMove(action)`**: Makes a move in the game, where the `action` can be one of the following letters:
  - `u` for moving up
  - `d` for moving down
  - `l` for moving left
  - `r` for moving right
- **`game.reward()`**: Returns the reward for the action performed. The reward system is as follows:
  - **-1** for non-winning moves.
  - **+10** for reaching the goal.
  - **-10** for falling into a pit.

Here's an example sequence of moves and actions:

python
Copy code

```
>>> game.display()
array([['+', '-', ' ', 'P'],
       [' ', 'W', ' ', ' '],
       [' ', ' ', ' ', ' '],
       [' ', ' ', ' ', ' ']], dtype='<U2')

>>> game.makeMove('d')
>>> game.makeMove('d')
>>> game.makeMove('l')
>>> game.display()
array([['+', '-', ' ', ' '],
       [' ', 'W', ' ', ' '],
       [' ', ' ', 'P', ' '],
```

```
       [' ', ' ', ' ', ' ']], dtype='<U2')
>>> game.reward()
-1
```

In this example:

- The player moves down twice and then left once.
- The current game state is displayed after each move.
- After each move, the reward is checked to evaluate the outcome of that action.

4. Understanding the Game State

The game state is represented as a 4x4x4 tensor. This tensor consists of four layers, each representing the position of a specific object on the board. The four objects tracked in the game are:

- The player (`P`)
- The goal (`+`)
- The pit (`-`)
- The wall (`W`)

Each matrix in the tensor contains a 1 to represent the position of an object and 0s elsewhere. For example, a 4x4 matrix in the tensor will contain a 1 at the position where the player is located, and all other positions will be 0. This structure is used to feed the game state into a neural network, which can then learn how to make optimal decisions based on the positions of the player, goal, pit, and walls.

Here's how the game state can be rendered as a NumPy array:

python
Copy code

```
>>> game.board.render_np()
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 0]],

       [[1, 0, 0, 0],
        [0, 0, 0, 0],
```

```
 [0, 0, 0, 0],
 [0, 0, 0, 0]],

[[0, 1, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]],

[[0, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]], dtype=uint8)
```

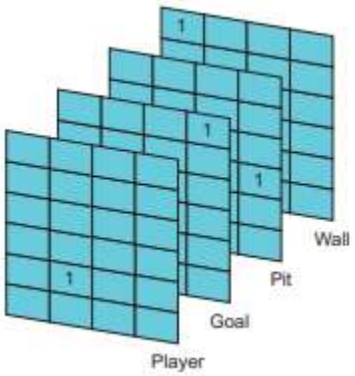


Figure: This is how the Gridworld board is represented as a numpy array. It is a 4 x 4 x 4 tensor, composed of 4 “slices” of a 4 x 4 grid. Each grid slice represents the position of an individual object on the board and contains a single 1, with all other elements being 0s. The position of the 1 indicates the position of that slice’s object.

This tensor can be flattened into a vector of 64 elements (4x4x4), which is then used as the input to the neural network for decision-making.

The Gridworld game engine provides a simple yet effective environment for training reinforcement learning models. By manipulating the initialization modes and observing the game states, players can gain insights into how agents can be trained to maximize rewards using Q-learning

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
THEORY & LAB TOPIC-8 : A NEURAL NETWORK AS THE Q FUNCTION

In this section, we delve into the implementation of a neural network that will serve as the Q function in a Q-learning reinforcement learning algorithm. The network is designed to process game states from the Gridworld environment and output Q-values for each possible action, which represent the expected future rewards for taking those actions in a given state.

1. Input Representation

The game state is represented as a 4x4x4 tensor, which contains four 4x4 matrices corresponding to the player, goal, pit, and wall positions on the Gridworld board. For easier processing by the neural network, this tensor is flattened into a 64-element vector. The input layer of the neural network is designed to accept this 64-length vector as input.

2. Neural Network Architecture

The Q-network used in this implementation is a fully connected feedforward network. The architecture consists of the following layers:

- **Input Layer**: 64 neurons (representing the flattened game state).
- **Hidden Layers**: Two hidden layers with 150 and 100 neurons, respectively. These layers use the ReLU (Rectified Linear Unit) activation function.
- **Output Layer**: 4 neurons, where each neuron corresponds to the Q-value for one of the four possible actions (up, down, left, and right).

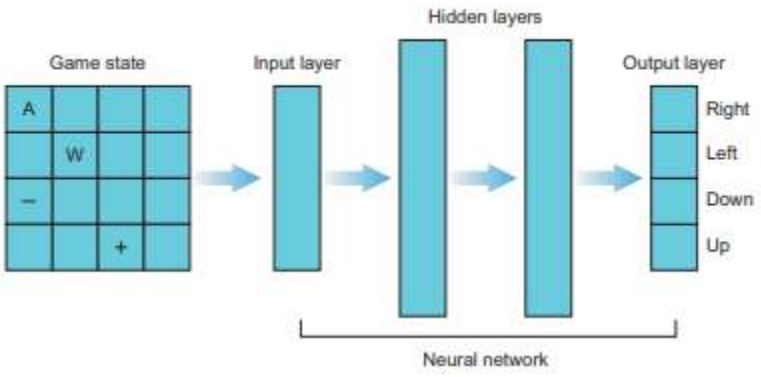


Figure: The neural network model we will use to play Gridworld. The model has an input layer that can accept a 64-length game state vector, some hidden layers (we use one, but

two are depicted for generality), and an output layer that produces a 4-length vector of Q values for each action, given the state.

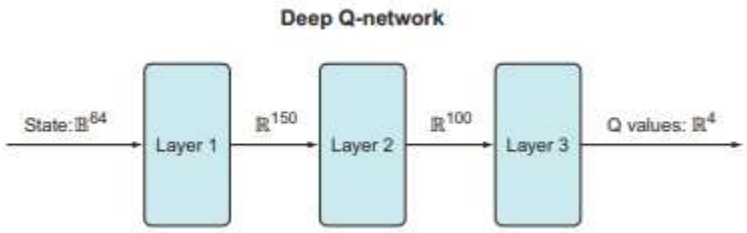


Figure: String diagram for our DQN. The input is a 64-length Boolean vector, and the output is a 4-length real vector of Q values.

Here's the PyTorch code for building the Q-network:

python
Copy code

```
import torch

l1 = 64  # Input layer size (game state vector)
l2 = 150  # First hidden layer
l3 = 100  # Second hidden layer
l4 = 4    # Output layer (Q-values for actions)

# Neural network model
model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3, l4)
)

# Loss function and optimizer
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

3. Training the Network

The training process uses Q-learning, where the agent updates its Q-values based on the rewards it receives after taking actions in the environment. The agent follows an **epsilon-greedy policy**, which balances exploration (choosing random actions) and exploitation (choosing the best-known actions based on the current Q-values). The main steps of the algorithm are:

1. For each epoch, a new game instance is created.
2. The current state is fed into the Q-network to compute the Q-values for all possible actions.
3. An action is selected based on the epsilon-greedy policy.
4. The action is taken in the game, and the reward and new state are observed.
5. The target Q-value is computed using the Bellman equation:
   $\text{target} = r_{t+1} + \gamma \cdot \max(Q(s', a'))$
   γ is the discount factor, and $Q(s', a')$ is the maximum Q-value for the next state.
6. The network is trained using this target value, and the process is repeated.

4. Main Training Loop

The training loop is implemented as follows:

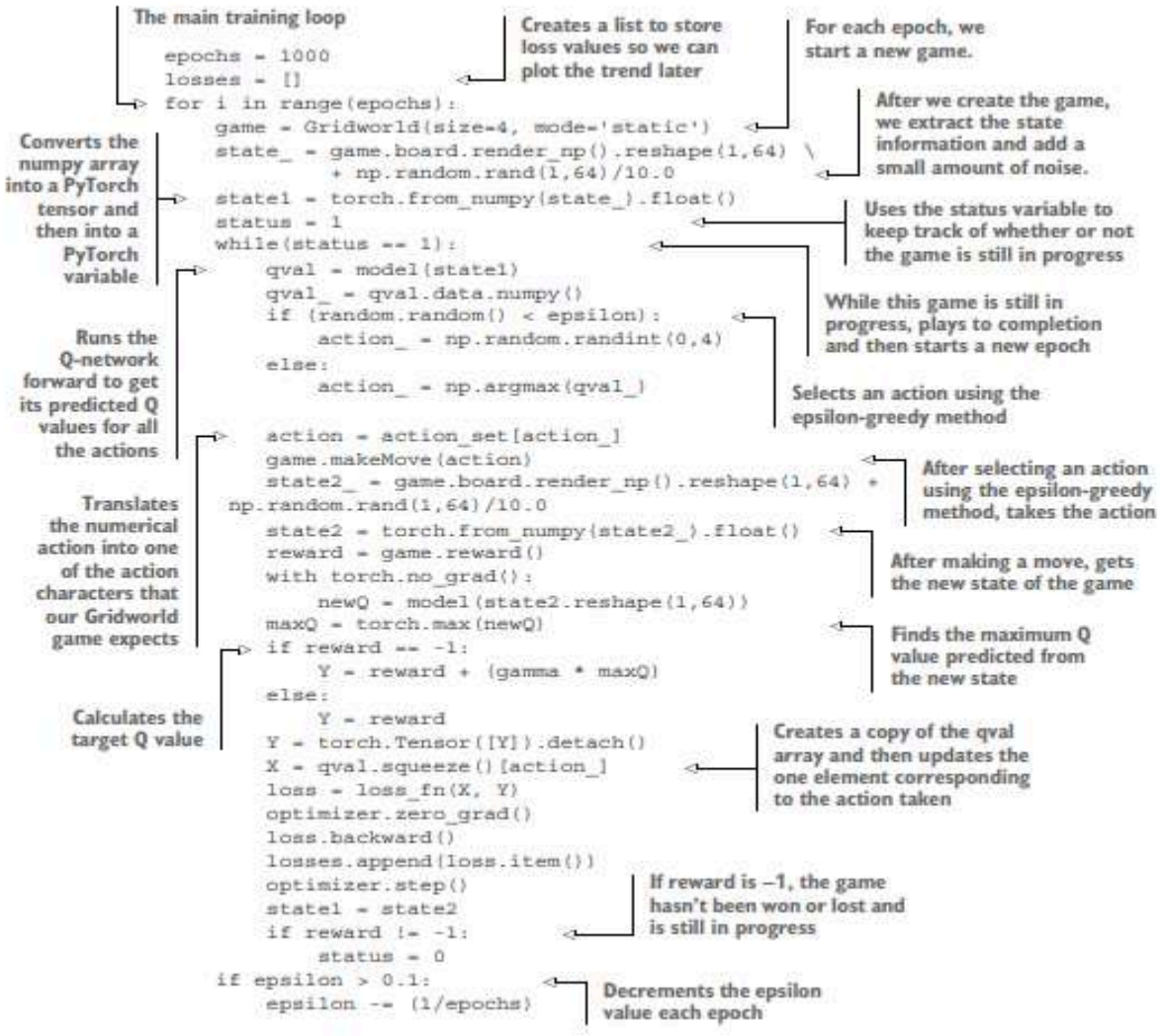
python
Copy code

```
epochs = 1000
epsilon = 1.0  # Exploration probability
gamma = 0.9    # Discount factor

for i in range(epochs):
    game = Gridworld(size=4, mode='static')
           state   =   game.board.render_np().reshape(1,   64)   +
np.random.rand(1, 64) / 10.0
    state1 = torch.from_numpy(state).float()
    status = 1

    while status == 1:
        # Predict Q-values for the current state
        qval = model(state1)
        qval_ = qval.data.numpy()
```

```
        # Choose action using epsilon-greedy policy
        if random.random() < epsilon:
            action_ = np.random.randint(0, 4)
        else:
            action_ = np.argmax(qval_)

        # Take the action
        action = action_set[action_]
        game.makeMove(action)
              state2  =  game.board.render_np().reshape(1,  64)  +
np.random.rand(1, 64) / 10.0
        state2 = torch.from_numpy(state2).float()

        # Observe reward
        reward = game.reward()

        # Compute the target Q-value
        with torch.no_grad():
            newQ = model(state2)
        maxQ = torch.max(newQ)
        Y = reward if reward != -1 else reward + gamma * maxQ

        # Update the network
        Y = torch.Tensor([Y]).detach()
        X = qval.squeeze()[action_]
        loss = loss_fn(X, Y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Update state
        state1 = state2
        if reward != -1:
            status = 0
```

```
# Decay epsilon
if epsilon > 0.1:
    epsilon -= (1 / epochs)
```

This loop trains the neural network over multiple epochs, reducing the epsilon value over time to shift from exploration to exploitation as the agent learns to play the game more effectively.



5. Evaluation and Testing

After training, the network can be tested by playing the game in different modes (static, player, random). The model should be able to learn the optimal policy for navigating the Gridworld

environment, though the level of generalization may depend on the complexity of the environment and the mode chosen for training.

The loss function typically decreases over time as the model learns, and this is visualized in the form of a loss plot

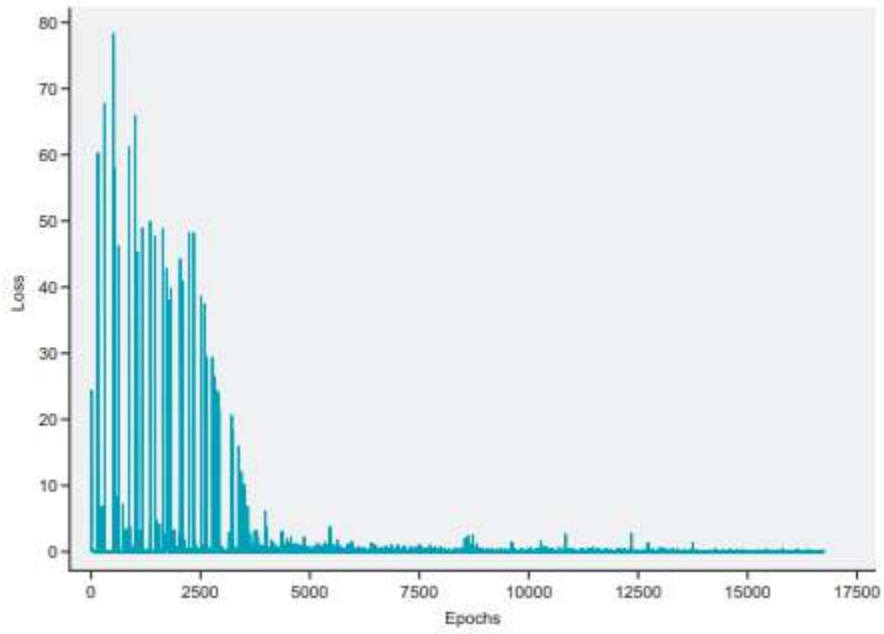


Figure: The loss plot for our first Q-learning algorithm, which is clearly down-trending over the training epochs.

Listing 3.4 Testing the Q-network

```
def test_model(model, mode='static', display=True):
    i = 0
    test_game = Gridworld(mode=mode)
    state_ = test_game.board.render_np().reshape(1,64) +
     np.random.rand(1,64)/10.0
    state = torch.from_numpy(state_).float()
    if display:
        print("Initial State:")
        print(test_game.display())
    status = 1
    while(status == 1):
        qval = model(state)
        qval_ = qval.data.numpy()
        action_ = np.argmax(qval_)
        action = action_set[action_]
        if display:
            print('Move #: %s; Taking action: %s' % (i, action))
        test_game.makeMove(action)
        state_ = test_game.board.render_np().reshape(1,64) +
     np.random.rand(1,64)/10.0
        state = torch.from_numpy(state_).float()
        if display:


            print(test_game.display())
        reward = test_game.reward()
        if reward != -1:
            if reward > 0:
                status = 2
                if display:
                    print("Game won! Reward: %s" % (reward,))
            else:
                status = 0
                if display:
                    print("Game LOST. Reward: %s" % (reward,))
        i += 1
        if (i > 15):
            if display:
                print("Game lost; too many moves.")
            break

    win = True if status == 2 else False
    return win
```

While the game is still in progress

Takes the action with the highest Q value

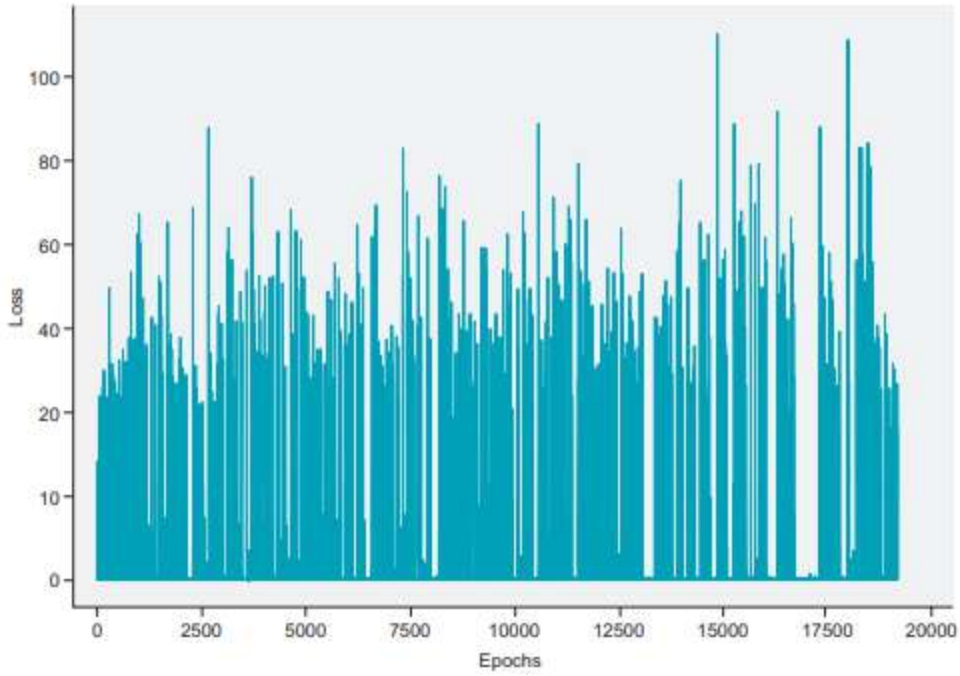


Figure The loss plot for Q-learning in random mode, which doesn't show any signs of convergence.

Using a neural network as the Q function allows for the approximation of Q values in complex environments like Gridworld. By flattening the input tensor and employing a multi-layer architecture, the network can learn to make informed decisions that maximize rewards. This approach highlights the power of deep learning in reinforcement learning applications, enabling agents to navigate and learn from their environments effectively.

UBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
THEORY & LAB TOPIC-9 : PREVENTING CATASTROPHIC FORGETTING: EXPERIENCE REPLAY

Overview: Catastrophic forgetting is a common issue in reinforcement learning (RL), particularly with gradient descent-based training methods in online environments. It occurs when an algorithm "forgets" previously learned information as it encounters new, similar, but slightly different situations. To mitigate this, a technique known as *experience replay* is employed. This strategy enhances learning stability and efficiency by enabling batch updating in an online learning scheme, thus helping to prevent catastrophic forgetting.

---

1. Understanding Catastrophic Forgetting

Catastrophic forgetting refers to the situation in online reinforcement learning when an algorithm overwrites previously learned weights, making it incapable of properly learning or retaining past knowledge. The problem arises when an algorithm is faced with new but similar environments that require different responses.

For instance:

- In game 1, the player moves right and receives a positive reward for reaching the goal.
- In game 2, the same move (right) places the player into a pit, resulting in a negative reward.

Even though the environment seems similar, the desired actions lead to different outcomes, which confuses the learning algorithm. As the algorithm updates its knowledge, it forgets previous associations, which can lead to a lack of meaningful learning.

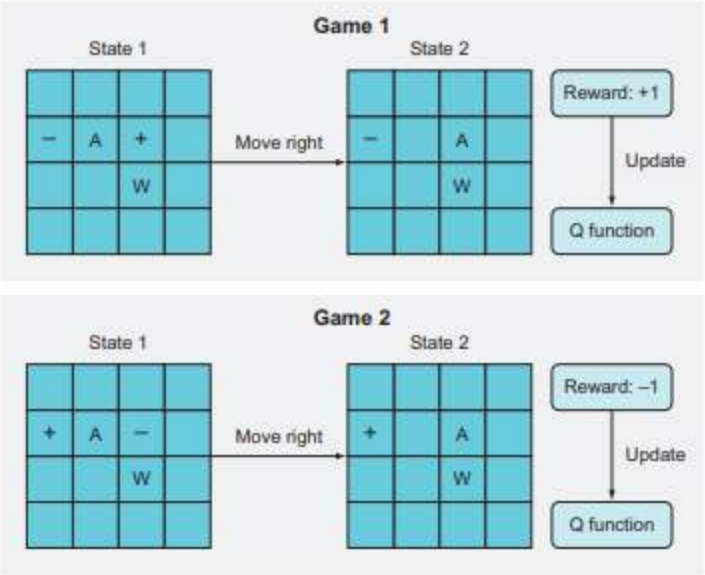


Figure: The idea of catastrophic forgetting is that when two game states are very similar and yet lead to very different outcomes, the Q function will get "confused" and won't be able to learn what to do. In this example, the catastrophic forgetting happens because the Q function learns from game 1 that 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.

---

2. What is Experience Replay?

Experience replay helps prevent catastrophic forgetting by enabling the RL model to learn from past experiences, rather than relying solely on the most recent one. The method achieves this by storing the agent's experiences and replaying them later in training. This breaks the correlation between consecutive updates, improving the stability of learning.

Key Steps in Experience Replay:

1. **Collect Experience:** When the agent takes an action in state *s*, the outcome (state, action, reward, new state) is stored in a replay buffer.
2. **Store the Experience:** The replay buffer accumulates experiences up to a predefined capacity.
3. **Random Sampling:** Once the buffer is filled, a random sample of past experiences is selected to be replayed during training.
4. **Batch Updates:** The selected experiences are used for batch updates, similar to how supervised learning operates with mini-batches.

By revisiting past experiences, the algorithm avoids overfitting to recent events and learns more generalized behaviors, addressing catastrophic forgetting.

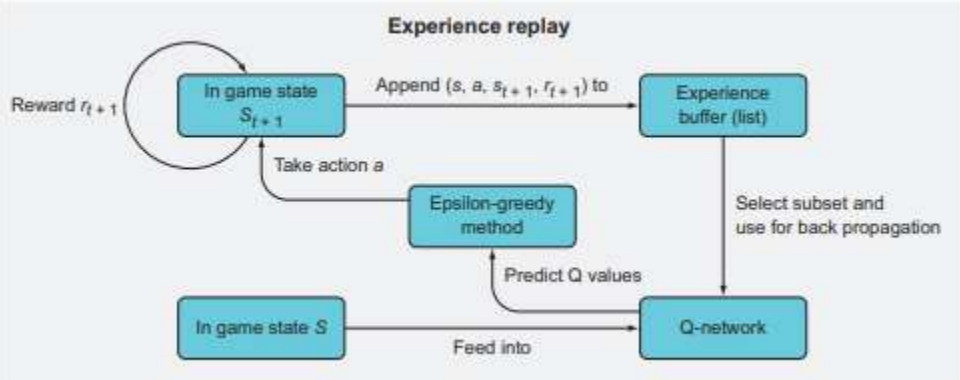


Figure: This is the general overview of experience replay, a method for mitigating a major problem with online training algorithms: catastrophic forgetting. The idea is to employ mini-batching by storing past experiences and then using a random subset of these experiences to update the Q-network, rather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

---

3. Benefits of Experience Replay

- **Breaks Correlations:** By decoupling the temporal correlations between consecutive experiences, the model achieves more stable and robust learning.
- **Improved Generalization:** The model is exposed to a diverse set of experiences, helping it learn general strategies rather than memorizing specific states.
- **Stabilizes Learning:** Experience replay enables more stable gradient updates, which leads to a smoother and more reliable learning curve.

---

4. Implementation

In the code implementation, experiences are stored in a data structure called a deque (double-ended queue), where older experiences are removed as new ones are added once the buffer is full.

A mini-batch of experiences is then randomly sampled and used to update the Q-network (the neural network used to estimate the Q-values). The algorithm computes the Q-values for both the current and next states, and the replayed experiences are used to compute the loss and adjust the model's weights.

Listing 3.5 DQN with experience replay

```
from collections import deque
epochs = 5000
losses = []
mem_size = 1000
batch_size = 200
replay = deque(maxlen=mem_size)
max_moves = 50
h = 0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) +
     np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float()
    status = 1
    mov = 0
    while(status == 1):
        mov += 1
        qval = model(state1)
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_]
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64) +
    np.random.rand(1,64)/100.0
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
        done = True if reward > 0 else False
        exp =  (state1, action_, reward, state2, done)
        replay.append(exp)
        state1 = state2

        if len(replay) > batch_size:
            minibatch = random.sample(replay, batch_size)
            state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
            action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
            reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
            state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
            done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])

            Q1 = model(state1_batch)
            with torch.no_grad():
                Q2 = model(state2_batch)

            Y = reward_batch + gamma * ((1 - done_batch) *
        torch.max(Q2,dim=1)[0])
            X = \
            Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze()
            loss = loss_fn(X, Y.detach())
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.item())
            optimizer.step()

        if reward != -1 or mov > max_moves:
            status = 0
            mov = 0
    losses = np.array(losses)
```

Sets the total size of the experience replay memory

Sets the mini-batch size

Creates the memory replay as a deque list

Sets the maximum number of moves before game is over

Computes Q values from the input state in order to select an action

Selects an action using the epsilon-greedy strategy

Creates an experience of state, reward, action, and the next state as a tuple

Adds the experience to the experience replay list

If the replay list is at least as long as the mini-batch size, begins the mini-batch training

Randomly samples a subset of the replay list

Separates out the components of each experience into separate mini-batch tensors

Recomputes Q values for the mini-batch of states to get gradients

Computes Q values for the mini-batch of next states, but doesn't compute gradients

Computes the target Q values we want the DQN to learn

If the game is over, resets status and mov number

Listing 3.6 Testing the performance with experience replay

```
max_games = 1000
wins = 0
for i in range(max_games):
    win = test_model(model, mode='random', display=False)
    if win:
        wins += 1
win_perc = float(wins) / float(max_games)
print("Games played: {0}, # of wins: {1}".format(max_games,wins))
print("Win percentage: {}".format(win_perc))
```

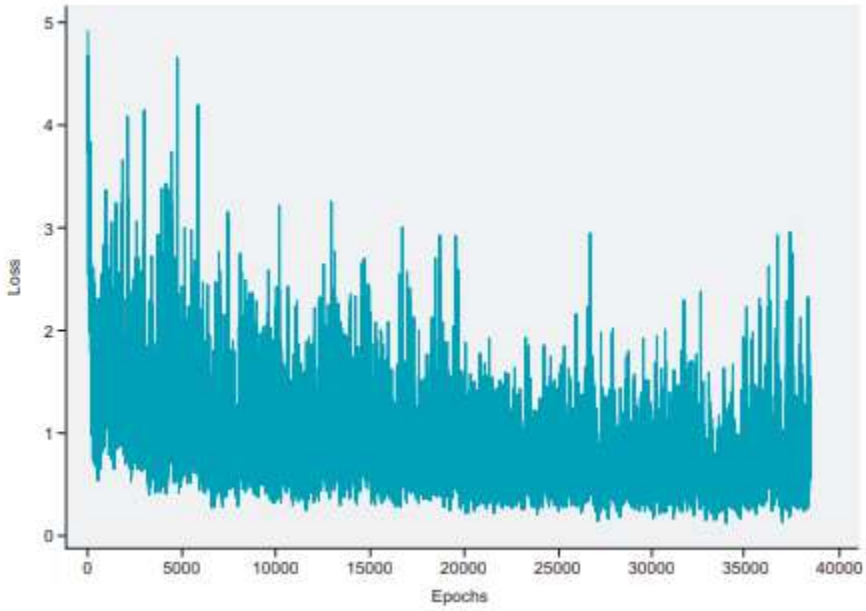


Figure: The DQN loss plot after implementing experience replay, which shows a clearly downtrending loss, but it's still very noisy.

---

5. Effectiveness in Mitigating Forgetting

By using experience replay, the Q-network is trained not only on the most recent action taken but also on a random sample of past experiences. This ensures that the agent retains knowledge of various states and their associated actions, rather than overwriting previous learning with each new experience. Experience replay has been successfully implemented in Deep Q-networks (DQN), a major advancement in RL.

In summary, experience replay is an essential method for addressing catastrophic forgetting in reinforcement learning. By leveraging stored past experiences, the method ensures more stable, efficient, and generalized learning.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-2
THEORY & LAB TOPIC-10 : IMPROVING STABILITY WITH A TARGET NETWORK

Deep Q-Networks (DQN) are powerful tools for reinforcement learning but are prone to instability during training. One major reason for this instability is frequent updates to the Q-network's parameters, leading to erratic behavior, especially when rewards are sparse. DeepMind identified this issue when training their DQN models and proposed the use of a **target network** to mitigate the problem.

1. Learning Instability

In reinforcement learning, updating the Q-network after every move can cause the model's Q-values to oscillate instead of converging. For example, a particular action might seem optimal after receiving a positive reward but might lead to negative rewards in future states. This inconsistency in rewards can lead the model to erratically update the value of actions, making it difficult for the network to stabilize its predictions.

2. Target Network Approach

To address this, DeepMind introduced the concept of a **target network**, which is essentially a copy of the Q-network but with delayed parameter updates. This design reduces the effect of sudden changes in Q-value predictions, thereby smoothing the learning process. Here's how it works:

- **Initialization**: Two networks are created at the start—the main Q-network and the target network. Both start with identical weights, but the target network's parameters are updated less frequently.
- **Training Process**: The Q-network is updated at each iteration, using the rewards from the agent's actions. However, the target network's parameters are used to compute the target Q-values, which are fed back to the Q-network for backpropagation. This stabilizes learning by reducing the impact of recent updates on the Q-values.
- **Periodic Synchronization**: Every few iterations, the target network's parameters are synchronized with the main Q-network's parameters. This periodic update allows the target network to lag behind the Q-network, providing a more stable estimate of future rewards.

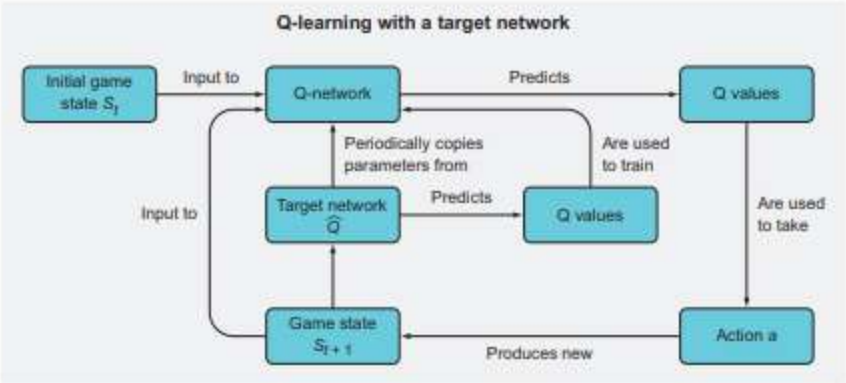


Figure : This is the general overview for Q-learning with a target network. It's a fairly straightforward extension of the normal Q-learning algorithm, except that you have a second Q-network called the target network whose predicted Q values are used to backpropagate through and train the main Q-network. 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.

3. Code Example

The target network implementation involves creating a copy of the main Q-network and synchronizing its parameters periodically. Below is a simplified example using PyTorch:

Listing 3.8 DQN with experience replay and target network

```
from collections import deque
epochs = 5000
losses = []
mem_size = 1000
batch_size = 200
replay = deque(maxlen=mem_size)
max_moves = 50
h = 0
sync_freq = 500
j=0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) +
     np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float()
    status = 1
    mov = 0
    while(status == 1):
        j+=1
        mov += 1
        qval = model(state1)
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_]
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64) +
     np.random.rand(1,64)/100.0
```

Sets the update frequency for synchronizing the target model parameters to the main DQN

```
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
        done = True if reward > 0 else False
        exp =  (state1, action_, reward, state2, done)
        replay.append(exp)
        state1 = state2

        if len(replay) > batch_size:
            minibatch = random.sample(replay, batch_size)
            state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
            action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
            reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
            state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
            done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])
            Q1 = model(state1_batch)
            with torch.no_grad():
                Q2 = model2(state2_batch)

            Y = reward_batch + gamma * ((1-done_batch) * \
            torch.max(Q2,dim=1)[0])
            X = Q1.gather(dim=1,index=action_batch.long() \
            .unsqueeze(dim=1)).squeeze()
            loss = loss_fn(X, Y.detach())
            print(i, loss.item())
            clear_output(wait=True)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.item())
            optimizer.step()

            if j % sync_freq == 0:
                model2.load_state_dict(model.state_dict())
        if reward != -1 or mov > max_moves:
            status = 0
            mov = 0

losses = np.array(losses)
```

Uses the target network to get the maximum Q value for the next state

Copies the main model parameters to the target network

Listing 3.7 Target network

```
import copy

model = torch.nn.Sequential(
    torch.nn.Linear(11, 12),
    torch.nn.ReLU(),
    torch.nn.Linear(12, 13),
    torch.nn.ReLU(),
    torch.nn.Linear(13,14)
)

model2 = model2 = copy.deepcopy(model)
model2.load_state_dict(model.state_dict())
```

Creates a second model by making an identical copy of the original Q-network model

Copies the parameters of the original model


```
sync_freq = 50

loss_fn = torch.nn.MSELoss()
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Synchronizes the frequency parameter; every 50 steps we will copy the parameters of model into model2

(Code omitted) Uses the same other settings as in listing 3.5

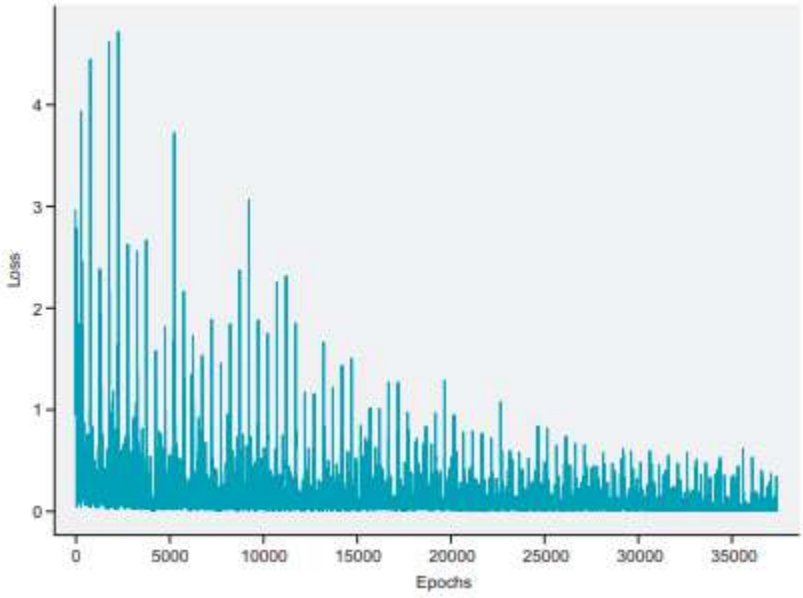


Figure: The DQN loss plot after including a target network to stabilize training. This shows a much faster training convergence than without the target network, but it has noticeable spikes of error when the target network synchronizes with the main DQN.

4. Benefits of Using a Target Network

Using a target network significantly improves the stability of the training process. In experiments, the loss plot after adding a target network shows a clearer downward trend with fewer noisy spikes. Additionally, this approach also reduces oscillations in the Q-values, leading to more stable and consistent action selection.

Although the training is still noisy, the use of a target network improves convergence speed and accuracy. Testing shows a ~3% improvement in win percentage in a Gridworld task compared to training without a target network, with performance reaching approximately 95%.

5. Conclusion

The introduction of a target network in DQN algorithms is a straightforward but powerful method to reduce instability during training. By decoupling the Q-value estimation from the immediate changes in the Q-network's parameters, the target network enables more consistent learning, ensuring that the model does not oscillate between over-optimistic or over-pessimistic action valuations.

For optimal performance, hyperparameters like the replay buffer size, batch size, target network update frequency, and learning rate need to be carefully tuned, as they can significantly affect the stability and effectiveness of the training process.