SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : A DIFFERENT APPROACH TO REINFORCEMENT LEARNING

In traditional reinforcement learning (RL), deep Q-networks (DQN) and policy gradient methods rely on backpropagation to optimize neural networks for decision-making in environments. However, these methods have drawbacks due to the complexity of tuning multiple hyperparameters, noisy gradients from stochastic gradient descent, and challenges with stability and convergence. As an alternative, evolutionary algorithms provide a backpropagation-free approach, inspired by natural selection, to train reinforcement learning models, bypassing the need for differentiability in the model architecture.

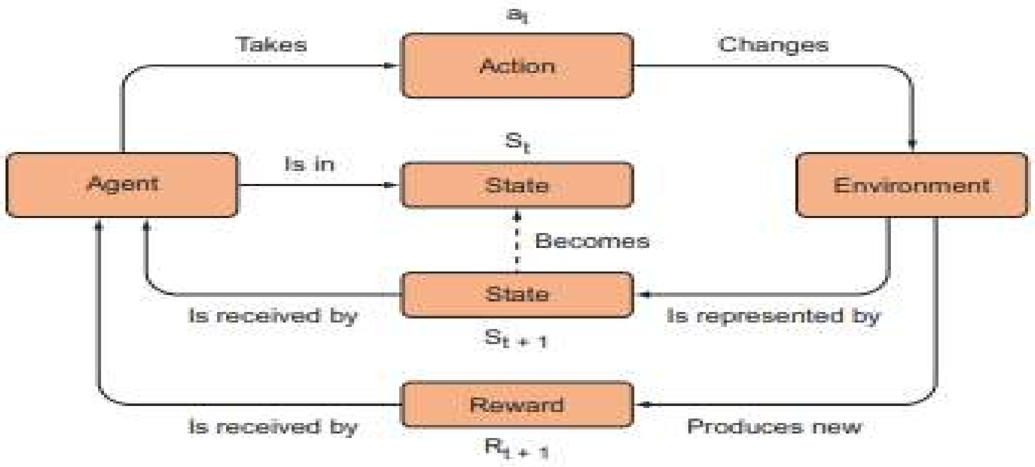


Figure: For the past algorithms that we covered, our agent interacted with environment, collected experiences, and then learned from those experiences. We repeated the same process over and over for each epoch until the agent stopped learning.

Evolutionary Algorithms for RL: Evolutionary algorithms optimize model parameters by iteratively selecting the most successful agents from a population based on a fitness score. Instead of gradient-based optimization, this approach uses genetic-inspired techniques, such as selection, mutation, and crossover, to evolve a population of agents over generations.

Fitness-Based Selection: Unlike DQN or policy gradient methods, which adjust model weights based on gradient signals, evolutionary methods score each agent's performance in the environment. Agents with higher scores have a higher probability of contributing genetic material (parameters) to the next generation, allowing for the gradual evolution of an optimal policy without backpropagation.

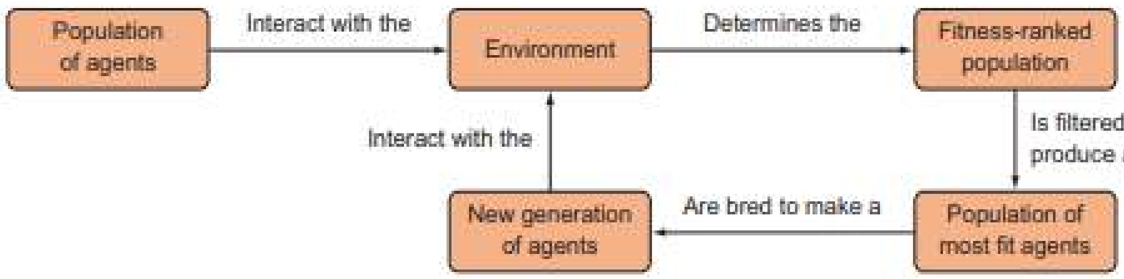


Figure: Evolutionary algorithms are different from gradient descent-based optimization techniques. With evolutionary strategies, we generate agents and pass the most favorable weights down to the subsequent agents.

Parallel Learning: Evolutionary algorithms allow for parallel evaluations of multiple agents, increasing computational efficiency. Since individual agents do not need to backpropagate gradients, this approach can easily scale across distributed systems, making it well-suited for high-performance computing environments.

Gradient-Free Optimization: Evolutionary methods are categorized as "gradient-free" algorithms, making them suitable for non-differentiable models or models with complex architectures where traditional gradient descent would struggle. This flexibility opens up new possibilities for model architectures in RL that are not constrained by differentiability requirements.

Algorithm Steps

1. Initialize Population: A population of agents is initialized, each with randomly assigned parameters (weights).
2. Evaluate Fitness: Each agent interacts with the environment, and a fitness score is calculated based on performance (e.g., total reward achieved in an episode).
3. Selection of Top Performers: The top-performing agents are selected based on fitness scores. Selection methods may include ranking, roulette wheel sampling, or tournament selection, ensuring that higher-performing agents have a greater influence on the next generation.

4. Crossover and Mutation: New agents are created by combining parameters from selected agents (crossover) and introducing slight random modifications (mutation) to maintain diversity in the population. This process encourages exploration of the solution space, much like the exploration-exploitation trade-off in traditional RL.
5. Repeat: The process is repeated for multiple generations, gradually evolving the population toward optimal policies without direct gradient-based learning.

Advantages and Applications

- Hyperparameter Simplicity: Evolutionary algorithms reduce the need for complex tuning of hyperparameters such as learning rates, optimizers, or batch sizes, which are critical in gradient-based methods.
- Non-Differentiable Models: By eliminating the dependency on backpropagation, these algorithms can be used to train models that are not differentiable, expanding the range of RL applications.
- Parallelization: Evolutionary methods are well-suited for parallel execution, as each agent can independently interact with the environment, enabling efficient scaling across distributed systems.

Example Use Case: Training Agents in Gridworld

In an example of applying evolutionary algorithms to a simple environment like Gridworld, agents are initialized with random weights and tasked with maximizing their reward. After each generation, the highest-scoring agents are selected, bred, and mutated to form a new population. Over successive generations, the agents evolve more effective strategies for navigating Gridworld without backpropagation, achieving competitive results to traditional RL methods.

Summary

Evolutionary algorithms provide a flexible, gradient-free approach to reinforcement learning, leveraging the principles of natural selection to optimize agents. This method not only reduces dependence on hyperparameter tuning but also allows for non-differentiable model architectures, expanding RL's potential applications across diverse and complex environments.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : REINFORCEMENT LEARNING WITH EVOLUTION STRATEGIE

"Reinforcement Learning with Evolution Strategies" presents an alternative optimization approach where evolutionary algorithms, rather than traditional gradient-based methods, drive agent learning. This approach harnesses principles of natural selection to optimize reinforcement learning agents by iterating through cycles of selection, breeding, and mutation.

Evolution in Theory

Natural Selection and Fitness:

- Inspired by natural selection, the concept of "fitness" is central to evolutionary algorithms. An agent's fitness corresponds to its performance, such as rewards accumulated in an RL environment. Agents with higher fitness scores are more likely to pass their characteristics to future generations.
- Like biological evolution, environmental context plays a significant role in determining which traits are beneficial, leading to fitness-based adaptation over generations.

Population and Mutations:

- Instead of individual learning through gradient descent, evolutionary strategies consider a population of agents with varied parameters.
- Small mutations are introduced in each generation, adding slight random variations to an agent's parameters, which can accumulate and drive improvements over multiple generations.
- These mutations mimic biological variations and help avoid local optima, allowing beneficial traits to persist and spread within the population.

Selection and Breeding:

- A breeding process combines parameters from the fittest agents, emulating genetic recombination. Agents with desirable traits (high fitness scores) are selected as "parents," and their traits are passed down to "offspring," forming the next generation.
- This selective process ensures that each new generation inherits optimized traits, enhancing overall performance.

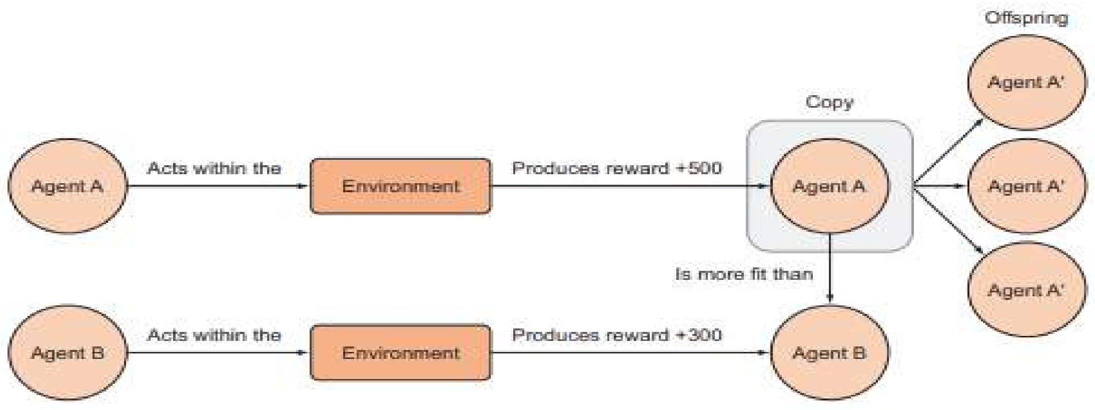


Figure: In an evolutionary algorithm approach to reinforcement learning, agents compete in an environment, and the agents that are more fit (those that generate more rewards) are preferentially copied to produce offspring. After many iterations of this process, only the most fit agents are left.

Evolution in Practice

Genetic Algorithm Implementation:

- The process begins by generating an initial population with random parameters. Each agent in the population is evaluated in the environment, and a fitness score is assigned based on performance.
- Recombination (Crossover): Selected parents recombine their parameters at random crossover points to create new parameter vectors for offspring.
- Mutation: Mutations are applied with low probability to some agents, introducing new genetic diversity. This random adjustment can be as simple as adding noise to the parameters, which helps the algorithm avoid convergence on suboptimal solutions.

Generational Process:

- Evolutionary reinforcement learning is iterative, with each generation building upon the improvements of the previous one. After evaluating the agents and generating offspring, the process repeats over multiple generations.
- The goal is for each generation's average fitness to improve, ultimately leading to a set of high-performing agents well-suited for the environment.

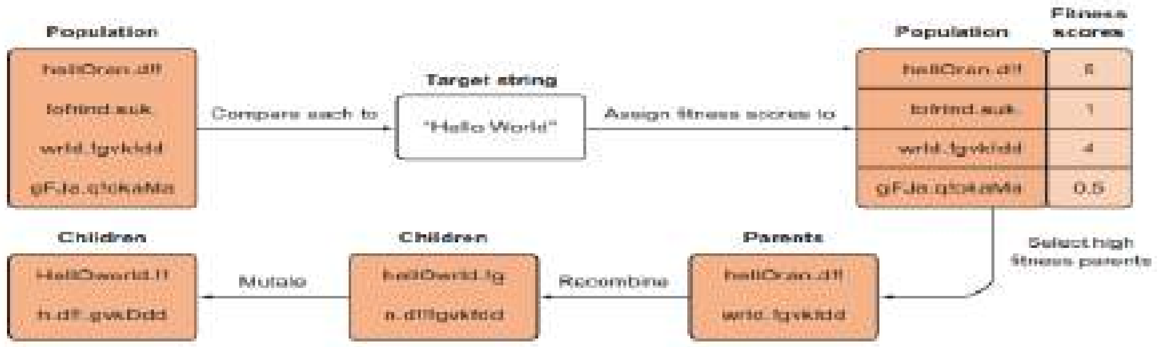


Figure: A string diagram outlining the major steps in a genetic algorithm for evolving a set of random strings toward a target string. We start with a population of random strings, compare each to the target string, and assign a fitness score to each string based on how similar it is to the target string. We then select high-fitness parents to “mate” (or recombine) to produce children, and then we mutate the children to introduce new genetic variance. We repeat the process of selecting parents and producing children until the next generation is full (when it’s the same size as the starting population).

Evolutionary reinforcement learning, both in theory and practice, leverages concepts of natural selection, mutation, and generational improvement to train agents in environments where gradient-based methods may not be feasible. This approach ensures diverse solutions, adaptable to the environment, and capable of reaching high performance without relying on traditional optimization techniques like backpropagation.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : A GENETIC ALGORITHM FOR CARTPOLE

The genetic algorithm for CartPole is an evolutionary optimization method used to train an agent to perform well in the CartPole environment. This environment challenges the agent to balance a pole upright on a moving cart, rewarding longer durations of balance. The algorithm leverages the principles of natural selection, recombination, and mutation to evolve a population of neural network agents toward better performance.

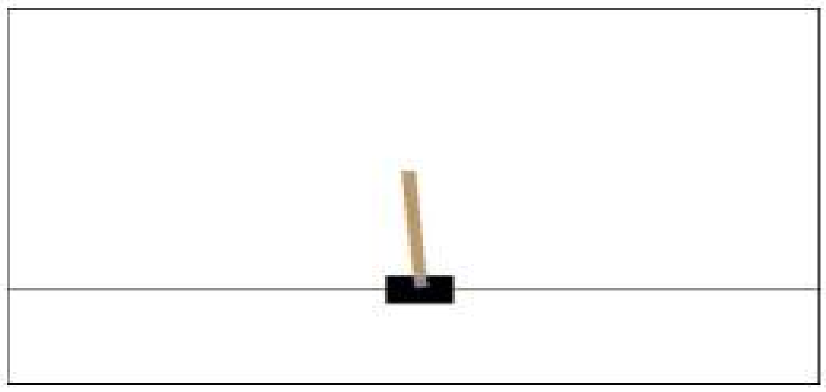


Figure: We will use the CartPole environment to test our agent. The agent is rewarded by keeping the pole upright, and it can move the cart left or right.

Genetic Algorithm

A genetic algorithm (GA) is an optimization technique inspired by the process of natural selection. It operates on a population of potential solutions, applying mechanisms such as selection, crossover, and mutation to evolve solutions over generations.

CartPole Environment

The CartPole environment is a classic reinforcement learning problem where the agent receives rewards for keeping a pole balanced on a cart. The agent's performance is measured by the number of time steps it successfully maintains the pole's upright position.

Concepts for Implementation

1. Agent Representation:

- Agents are represented as neural networks with three layers. The input is a state, and the output is a probability distribution over possible actions.
- Parameters (weights and biases) of the network are stored as a flattened vector to facilitate mutation and recombination.

2. Fitness Evaluation:
    - The fitness of an agent is determined by testing it in the CartPole environment. The fitness score corresponds to the number of time steps the agent successfully balances the pole.
3. Evolutionary Process:
    - Recombination: Combines segments of two parent agents' parameter vectors to produce offspring.
    - Mutation: Introduces randomness by altering a small portion of an agent's parameter vector.
    - Selection: Tournament-style selection ensures better-performing agents are more likely to contribute to the next generation.

Implementation Steps

Neural Network Definition:

- The network uses ReLU activation for hidden layers and log-softmax activation in the output layer. It requires unpacking the parameter vector into layer-specific matrices for computation.

Population Initialization:

- A population of agents is initialized with random parameter vectors. Each agent is stored as a dictionary containing its parameters and a fitness score.

Python Code
```python
def spawn_population(N=50, size=407):
    pop = []
    for i in range(N):
        vec = torch.randn(size) / 2.0
        pop.append({'params': vec, 'fitness': 0})
    return pop
```

Recombination:

- Two parent agents produce two children by splitting their parameter vectors at a random point and swapping segments.

Python Code
```
def recombine(x1, x2):
    split_pt = np.random.randint(len(x1['params']))
    child1 = torch.cat([x1['params'][:split_pt], x2['params'][split_pt:]])
    child2 = torch.cat([x2['params'][:split_pt], x1['params'][split_pt:]])
    return {'params': child1, 'fitness': 0}, {'params': child2, 'fitness': 0}
```

Mutation:

- Randomly changes a fraction of the parameters in the vector to maintain genetic diversity.

Python Code
```
def mutate(agent, rate=0.01):
    params = agent['params']
    indices = np.random.choice(len(params), int(rate * len(params)), replace=False)
    params[indices] = torch.randn(len(indices)) / 10.0
    agent['params'] = params
    return agent
```

Fitness Evaluation:

- Agents are tested in the CartPole environment, and their performance is scored.

Python Code
```
def test_model(agent):
    env = gym.make("CartPole-v0")
    state = torch.tensor(env.reset(), dtype=torch.float32)
    score = 0
    done = False
    while not done:
        params = unpack_params(agent['params'])
        action = model(state, params).argmax().item()
        state, _, done, _ = env.step(action)
        score += 1
    return score
```

Selection and Next Generation:

    - A tournament-style selection mechanism is used to pick parents for the next generation, balancing exploration and exploitation.

Python Code

```
def next_generation(pop, mut_rate=0.01, tournament_size=0.2):
    new_pop = []
    while len(new_pop) < len(pop):
        batch = random.sample(pop, int(len(pop) * tournament_size))
        parents = sorted(batch, key=lambda x: x['fitness'], reverse=True)[:2]
        offspring = recombine(parents[0], parents[1])
        new_pop.extend([mutate(offspring[0], mut_rate), mutate(offspring[1], mut_rate)])
    return new_pop
```

Training the Population

The algorithm iteratively evolves the population over multiple generations, tracking and optimizing the fitness scores.

Python Code

```
num_generations = 25
population_size = 500
mutation_rate = 0.01

pop = spawn_population(N=population_size, size=407)
for generation in range(num_generations):
    pop, avg_fit = evaluate_population(pop)
    print(f"Generation {generation}: Average Fitness = {avg_fit}")
    pop = next_generation(pop, mut_rate=mutation_rate)
```

Hyperparameters

Key hyperparameters that can be adjusted include:

- Population Size: The number of agents in each generation (e.g., 500).
- Mutation Rate: The probability of mutation occurring in offspring (e.g., 0.01).
- Number of Generations: The total number of iterations for evolving the population (e.g., 25).

Results

The genetic algorithm demonstrates a steady increase in the average score of the population over generations, indicating that the agents are becoming more proficient at balancing the pole. The performance can be visualized through plots showing the average fitness over time .

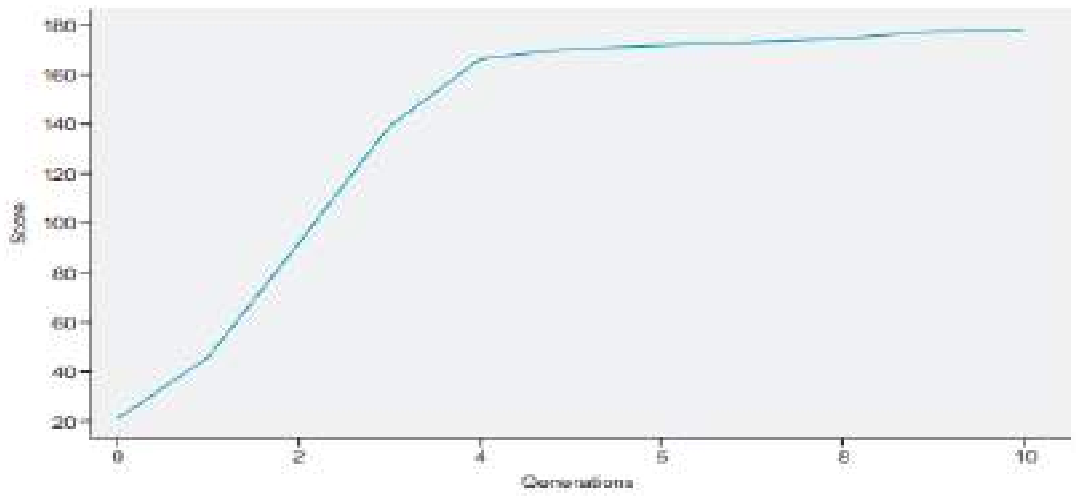


Figure: The average score of the population over generations in a genetic algorithm used to train agents to play CartPole.

The genetic algorithm effectively optimizes the performance of agents in the CartPole environment by leveraging evolutionary strategies. By iteratively selecting, recombining, and mutating agents, the algorithm can discover high-performing solutions that excel in maintaining the balance of the pole.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : PROS AND CONS OF EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are optimization techniques inspired by biological evolution, involving mechanisms like selection, mutation, and crossover. They are often used for problems requiring exploration or when gradient-based approaches are unsuitable. Below is a detailed analysis of their advantages and disadvantages, as discussed in the provided document.

Pros of Evolutionary Algorithms

1. **Enhanced Exploration Capabilities**:
    - Evolutionary Algorithms tend to explore the solution space more thoroughly than gradient-based methods.
    - They generate a diverse population of solutions, which allows for a broader search of potential solutions.
    - This is particularly beneficial in complex landscapes where local optima may trap gradient-based methods. particularly useful when adequate exploration is crucial for the problem at hand.
2. **Gradient-Free Optimization**:
    - Evolutionary Algorithms do not require gradient information, making them suitable for problems where the objective function is not differentiable or is noisy.
    - This characteristic allows them to be applied in a wider range of scenarios compared to gradient-based methods .
3. **Robustness to Local Optima**:
    - Due to their exploratory nature, Evolutionary Algorithms are less likely to converge prematurely to local optima.
    - They maintain a diverse population of solutions, which increases the chances of finding a global optimum .
4. **Flexibility**:
    - Evolutionary Algorithms can be easily adapted to different types of problems and can incorporate various types of representations and operators, making them versatile tools for optimization .

Cons of Evolutionary Algorithms

1. **Sample Inefficiency**:

   - EAs are often more sample-intensive than gradient-based methods. They require a large number of evaluations to determine the fitness of individuals in the population, which can be computationally expensive and time-consuming .
2. **Computational Cost**:
   - The need to evaluate many individuals in each generation can lead to high computational costs, especially in environments where data collection is expensive, such as robotics or autonomous vehicles .
3. **Parameter Sensitivity**:
   - The performance of EAs can be sensitive to the choice of parameters, such as population size, mutation rates, and selection methods. Poor parameter choices can lead to suboptimal performance .
4. **Convergence Speed**:
   - While EAs are good at exploration, they may converge more slowly than gradient-based methods, particularly in well-defined optimization problems where gradients are available .
5. **Impracticality for Real-World Applications Without Simulators**:
   - For domains like robotics, the cost of physical experiments can be prohibitive. Without simulators, the feasibility of applying evolutionary algorithms decreases significantly.

Evolutionary algorithms offer a powerful alternative to traditional optimization methods, particularly in scenarios requiring extensive exploration and flexibility. However, their sample inefficiency and computational costs can be significant drawbacks, especially in data-intensive applications. Understanding the pros and cons of EAs is crucial for selecting the appropriate optimization strategy for a given problem.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : EVOLUTIONARY ALGORITHMS AS A SCALABLE ALTERNATIVE

Evolutionary algorithms (EAs) draw inspiration from biological evolution to solve optimization problems. These algorithms simulate processes like mutation, selection, and recombination to evolve solutions over generations. When implemented with simulators, EAs become an efficient and scalable alternative for optimization tasks, especially in reinforcement learning and training autonomous agents

Evolutionary Algorithms:

- EAs operate by generating a population of potential solutions, selecting the best performers, and iteratively improving them through processes analogous to natural selection, mutation, and reproduction.
- They are capable of optimizing nondifferentiable or discrete functions, which is a limitation of gradient-based methods.

Evolutionary Strategies (ES):

- A subclass of evolutionary algorithms that focuses on generating new individuals through noise addition and weighted sums rather than biological mating and recombination.
- ES can efficiently scale with the addition of computational resources.

Key Advantages

No Gradient Calculation: EAs bypass the need for gradient computations, unlike gradient-based methods that rely on backpropagation. This makes EAs faster for certain network complexities, reducing computation time by 2–3 times.

Parallelization Potential: Evolutionary algorithms are highly parallelizable. The fitness of multiple agents can be evaluated simultaneously, significantly accelerating training times when distributed across multiple machines.

Scaling Evolutionary Algorithms

OpenAI demonstrated the scalability of EAs in their 2017 paper, achieving remarkable results by leveraging distributed computing:

- On a single machine with 18 CPU cores, a 3D humanoid agent learned to walk in 11 hours.
- By scaling to 80 machines (1,440 CPU cores), the same task was completed in under 10 minutes.

Why Gradient-Based Approaches Struggle to Scale

Gradient-based methods often involve:

- High communication overhead, with gradients passed between machines during each update cycle.

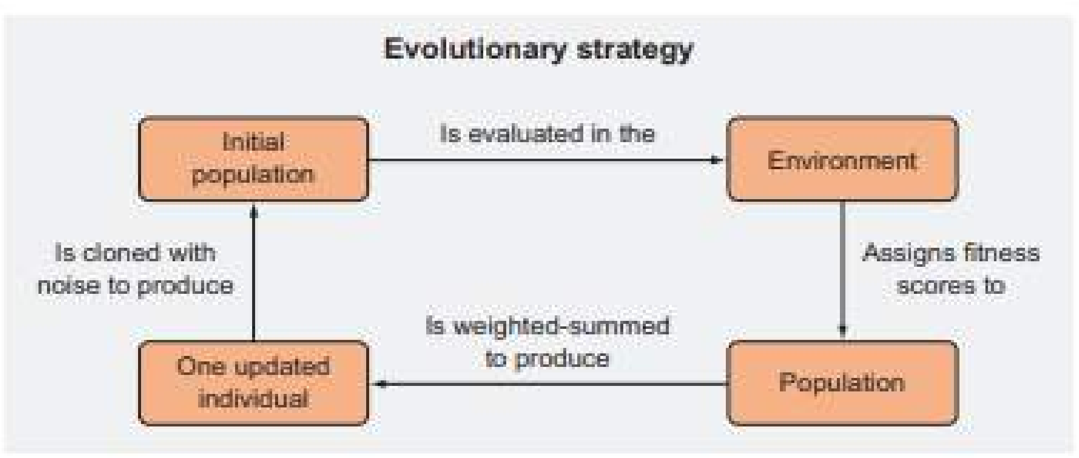


Figure: In an evolutionary strategy we create a population of individuals by repeatedly adding a small amount of random noise to a parent individual to generate multiple variants of the parent. We then assign fitness scores to each variant by testing them in the environment, and then we get a new parent by taking a weighted sum of all the variants.

Parallel vs. Serial Processing

- **Serial Processing**: Agents are evaluated sequentially, making the overall training time dependent on the number of agents.
- **Parallel Processing**: Each agent evaluates fitness on its dedicated machine. This approach reduces the training time to that of a single agent's evaluation, independent of population size.

For example:

- **Serial**: Evaluating 10 agents takes 5 minutes (30 seconds per agent).
- **Parallel**: Evaluating 10 agents on 10 machines takes only 30 seconds.

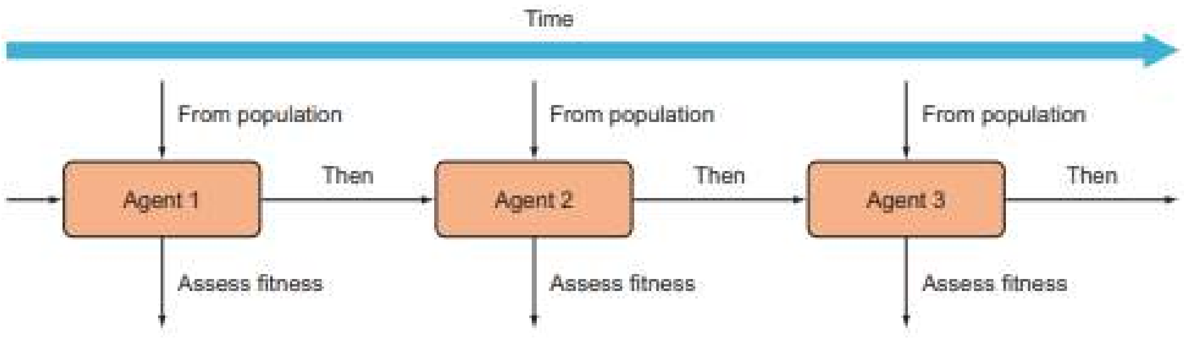


Figure: Determining the fitness of an agent is often the slowest step in a training loop and requires that we run the agent through the environment (possibly many times). If we are doing this on a single computer, we will be doing this in serial—we have to wait for one to finish running through the environment before we can start determining the fitness of the second agent. The time it takes to run this algorithm is a function of the number of agents and the time it takes to run through the environment for a single agent.

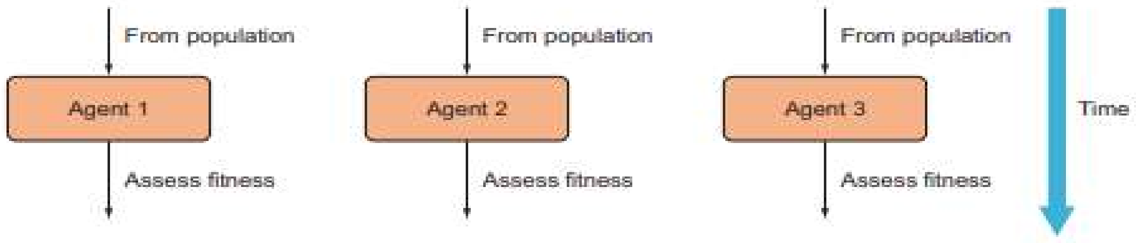


Figure: If we have multiple machines at our disposal, we can determine the fitness of each agent on its own machine in parallel with each other. We do not have to wait for one agent to finish running through the environment before starting the next one. This will provide a huge speed up if we are training agents with a long episode length. You can see now that this algorithm is only a function of the time it takes to assess the fitness of a single agent, and not the number of agents we are assessing.

Distributed Computing

Modern distributed computing techniques further enhance EAs:

1. **Shared Random Seeds**: Allow reproducibility across machines without transferring entire data sets.

2. **Efficient Communication**: Instead of sharing large parameter vectors, only fitness scores (single numbers) are exchanged, minimizing communication overhead.
3. **Linear Scaling**: By reducing inter-machine communication, EAs maintain linear scaling efficiency, achieving consistent performance boosts with added resources.

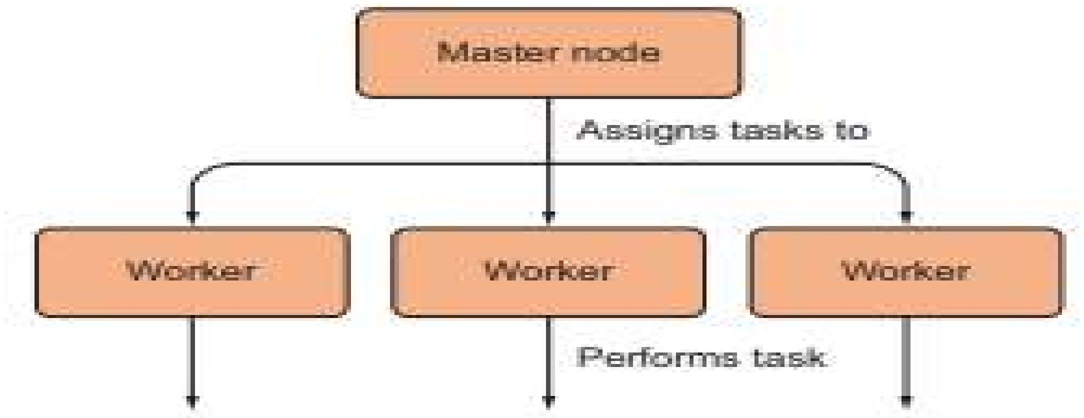


Figure: A general schematic for how distributed computing works. A master node assigns tasks to worker nodes; the worker nodes perform those tasks and then send their results back to the master node (not shown).

Comparison with Gradient-Based Methods

Feature	Evolutionary Algorithms	Gradient-Based Approaches
Gradient Dependence	Not required	Required
Scalability	Linear with resources	Diminishing returns at scale
Communication Overhead	Minimal (fitness scores only)	High (gradient updates per cycle)
Optimization	Works on non-differentiable and discrete functions	Limited to differentiable functions

Applications

- **Simulators**: Training agents in virtual environments significantly reduces time and costs, bypassing the need for expensive hardware setups.

- **Autonomous Systems**: EAs efficiently explore complex state spaces, making them suitable for robotics, autonomous vehicles, and other AI-driven systems.

EAs are a robust alternative to gradient-based methods, especially in environments where simulators are available. They excel in scalability, handling parallel and distributed computing efficiently. While more data-hungry, EAs offer flexibility in optimizing complex, non-differentiable problems.

The combination of simplicity, parallelism, and scalability makes evolutionary algorithms a powerful tool in modern optimization challenges.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : DRAWBACKS OF Q-LEARNING

Q-learning is a reinforcement learning (RL) algorithm that estimates the expected value of rewards for each state-action pair, known as Q-values. While widely used, Q-learning has limitations that can hinder its performance in complex environments. This documentation explores the key issues with Q-learning and introduces distributional Q-learning as a potential improvement.

Issues with Q-learning

Loss of Information

Q-learning typically computes a single point estimate (the average) for the action values. This approach discards valuable information about the variability and distribution of rewards associated with different actions. In environments where rewards can vary significantly, relying solely on the average can lead to misleading conclusions about the effectiveness of actions.

Example:
In a medical scenario involving a new drug, a bimodal distribution of outcomes might show that some patients benefit significantly while others see no effect. The expected value fails to capture this diversity, potentially leading to poor decisions.

Misrepresentation of Rewards

By averaging rewards, Q-learning may produce Q-values that are not representative of actual outcomes. For instance, if a state-action pair results in rewards of either +10 or -10, the expected value (0) does not reflect any of the true observed rewards.

In many real-world scenarios, the rewards for a given state-action pair may not cluster around a single value. Instead, they may exhibit complex patterns, such as bimodal distributions where rewards can be both very high and very low. For instance, in a medical context, a treatment might yield significant improvements for some patients while causing adverse effects in others. Averaging these outcomes could result in a misleading estimate close to zero, which does not accurately reflect the observed rewards.

Inability to Capture Risk:

Traditional Q-learning focuses on maximizing expected rewards without considering the associated risks. This limitation can be particularly problematic in applications like healthcare, where understanding the variability and potential negative outcomes of treatments is crucial for making informed decisions.

Illustrative Example

In a randomized drug trial, Q-learning might predict the average reduction in blood pressure as -13 mmHg. However:

- The treatment group displays a bimodal distribution with peaks at -2.5 mmHg and -22.3 mmHg.
- The expected value (-13 mmHg) misrepresents individual outcomes, leading to suboptimal decision-making for patients.

Transition to Distributional Q-learning

To address these limitations, distributional Q-learning has been developed. This approach aims to provide a complete distribution of action values rather than a single average. By capturing the full range of possible outcomes, distributional Q-learning allows for more nuanced decision-making and better performance in environments characterized by uncertainty and variability. Key advantages include:

- **Better Decision-Making:** Captures multimodal distributions, revealing diverse outcomes and enabling more nuanced policies.
- **Risk Sensitivity:** Allows policies to account for variability, enabling risk-averse or risk-seeking strategies as required.
- **Empirical Success:** Demonstrated superior performance in complex environments, as shown in studies such as the Rainbow DQN algorithm.

While Q-learning provides a foundational approach to RL, its reliance on expected values limits its effectiveness in scenarios with high variability or multimodal reward distributions. Distributional Q-learning offers a powerful alternative by incorporating full reward distributions, enabling better decision-making and improved performance in diverse environments.

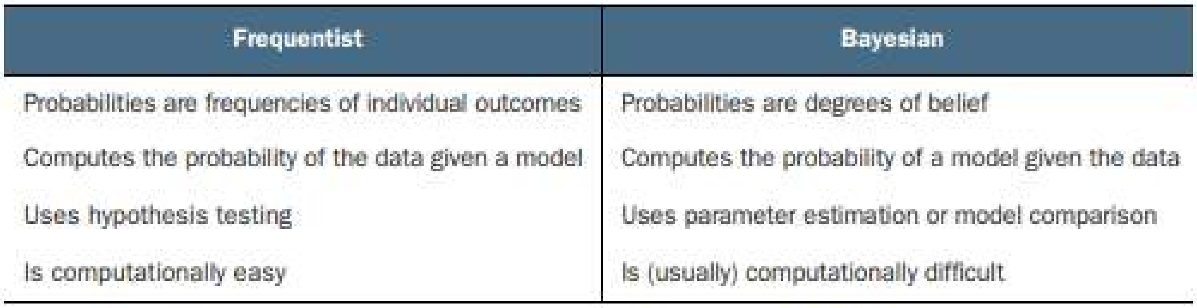
SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : PROBABILITY AND STATISTICS REVISITED

Probability and statistics form a foundational part of machine learning and reinforcement learning. This section revisits key concepts, exploring frequentist and Bayesian approaches, priors, posteriors, expectations, and variances. These tools are crucial for implementing advanced models like Distributional Deep Q-Networks (Dist-DQN), which leverage probability distributions to assess state-action values.

Frequentist vs. Bayesian Probability

1. **Frequentist Perspective**
   - Probability is viewed as the frequency of an event occurring over infinite trials.
   - Example: The probability of a coin landing heads approaches 0.5 as the number of flips increases.
   - **Limitation**: Inapplicable to one-off events (e.g., predicting election outcomes).
2. **Bayesian Perspective**
   - Probabilities represent degrees of belief, updated as new information becomes available.
   - Useful for single events where a frequentist approach lacks practicality.
   - This approach allows for the calculation of prior and posterior distributions, where the prior represents initial beliefs and the posterior reflects updated beliefs after considering new evidence.

Table: Frequentist versus Bayesian probabilities



Frequentist	Bayesian
Probabilities are frequencies of individual outcomes	Probabilities are degrees of belief
Computes the probability of the data given a model	Computes the probability of a model given the data
Uses hypothesis testing	Uses parameter estimation or model comparison
Is computationally easy	Is (usually) computationally difficult

Priors and Posteriors

- **Prior Probability**: Initial belief about the likelihood of outcomes before seeing new data.

    - Example: Assuming equal chances for all candidates in an unknown election.
- **Posterior Probability**: Updated belief after incorporating new evidence.
    - Process: Bayesian inference continuously refines priors into posteriors as data is received.

A **prior probability distribution** is established before new information is received. For instance, in a four-way race, without specific information about candidates, one might assign equal probabilities (¼) to each candidate. Upon receiving new information (e.g., candidate profiles), the prior is updated to form a **posterior probability distribution**, which can then serve as a new prior for subsequent updates.

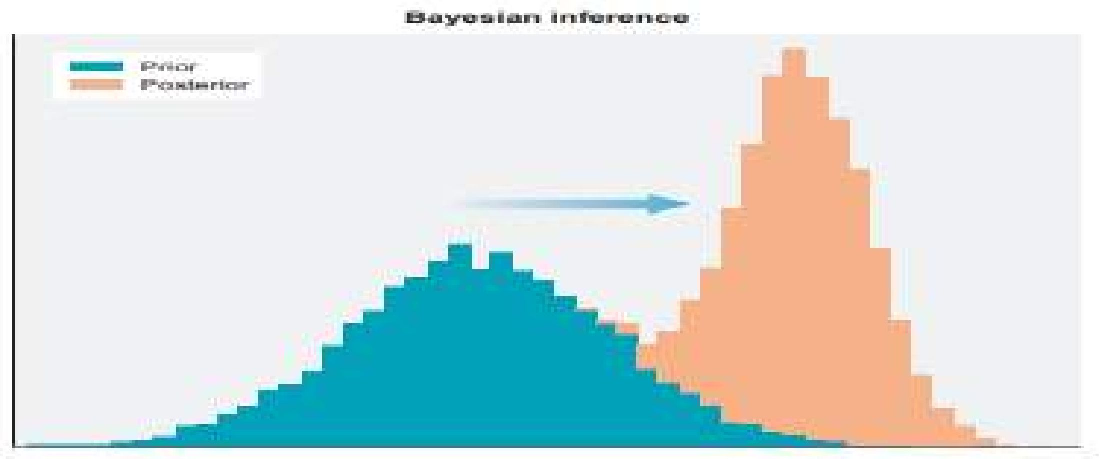


Figure: Bayesian inference is the process of starting with a prior distribution, receiving some new information, and using that to update the prior into a new, more informed distribution called the posterior distribution.

Expectation and Variance

- **Expectation (Mean)**:
    - Weighted average of outcomes, where weights are probabilities.
    - Formula (Discrete Case): E[X]=∑xP(x)⋅xE[X] = \sum_{x} P(x) \cdot xE[X]=x∑P(x)⋅x

Python Example:
Python Code
```
outcomes = [18, 21, 17, 17, 21]
probabilities = [0.6, 0.1, 0.1, 0.1, 0.1]
expected_value = sum(p * x for p, x in zip(probabilities,
outcomes))
print(expected_value)  # Output: 18.4
```

-
- **Variance**:
  - Measures the spread of the distribution.
  - Formula: Var(X)=σ2=E[(X−μ)2]
  - Relation: Variance is the square of the standard deviation (σ\sigmaσ).

Random Variables

- **Definition**: A variable whose values depend on a probability distribution.
- Example: Tomorrow's temperature modeled as T=t0+eT = t_0 + eT=t0+e, where eee is Gaussian noise.

Discrete vs. Continuous Distributions

- **Discrete Distributions**:
  - Finite sample space (e.g., coin flips).
  - Example: Probability values stored in arrays.
- **Continuous Distributions**:
  - Infinite outcomes, represented using probability densities.
  - Example: Temperature ranges modeled using continuous functions.

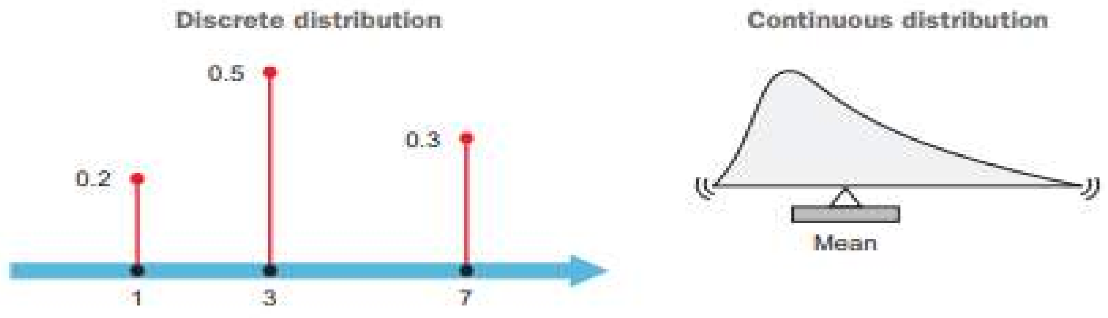


Figure: Left: A discrete distribution is like a numpy array of probabilities associated with another numpy array of outcome values. There is a finite set of probabilities and outcomes. Right: A continuous distribution represents an infinite number of possible outcomes, and the y axis is the probability density (which is the probability that the outcome takes on a value within a small interval).

Applications in Reinforcement Learning

- These concepts are pivotal for understanding the **Distributional Bellman Equation** in reinforcement learning.

- By modeling rewards as distributions rather than deterministic values, algorithms can better handle uncertainty and randomness.

Understanding probability and statistics at this deeper level provides essential tools for advanced machine learning and reinforcement learning tasks, enabling models to incorporate uncertainty effectively. The concepts reviewed here are foundational for implementing sophisticated approaches like Dist-DQN.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : THE BELLMAN EQUATION

The Bellman equation is a fundamental concept in reinforcement learning, providing the mathematical foundation for understanding and updating value functions. It describes how the value of a state-action pair can be determined recursively based on the observed rewards and the values of subsequent states.

The Bellman equation can be expressed as follows:

$$Q_\pi(s_t, a_t) \leftarrow r_t + \gamma \cdot V_\pi(s_{t+1})$$

Where:

- $Q_\pi(s_t, a_t)$: The value of the state-action pair under policy π.
- r_t: The immediate reward observed after taking action a_t in state s_t.
- γ: The discount factor (determines the importance of future rewards).
- $V_\pi(s_{t+1})$: The value of the next state, defined as:

The equation suggests that the value of a current state-action pair is the observed reward plus the discounted maximum value of all possible actions in the subsequent state. This recursive definition enables iterative updates to approximate the optimal Q*, the perfect state-action value function.

Neural Network Approximation:

When neural networks approximate Qπ, the Bellman equation's left-hand side represents predicted values. The network parameters are updated to minimize the error between the predicted Qπ and the computed values from the equation.

The Distributional Bellman Equation

Motivation:

The standard Bellman equation assumes a deterministic environment, where observed rewards for a given state-action pair are consistent. However, real-world scenarios often involve randomness in state transitions and rewards, requiring a probabilistic approach.

$$Z(s_t, a_t) \leftarrow R(s_t, a_t) + \gamma \cdot Z(s_{t+1}, A_{t+1})$$

Where:

- $Z(s_t, a_t)$: The distributional value function, representing the probability distribution of rewards for a state-action pair.
- $R(s_t, a_t)$: The reward distribution for the state-action pair.
- γ: Discount factor.

Key Differences:

- The deterministic Qπ is replaced by Z, capturing the stochastic nature of rewards and state transitions.
- This approach models a distribution of rewards rather than just their expected values, enabling better handling of uncertainty and variance.

Applications:

- Enables reinforcement learning algorithms to learn full reward distributions.
- Facilitates risk-sensitive policies that account for variance and multimodality in the value distribution.

Significance of the Bellman Equation in Reinforcement Learning

- Recursive Updates: The equation provides a framework for iteratively refining value estimates, crucial for methods like Q-learning and Deep Q-Networks (DQN).
- Stochastic Adaptation: The distributional variant allows algorithms to adapt to environments with inherent randomness.

- Policy Optimization: Helps identify optimal policies by maximizing expected returns or considering reward distributions for risk-aware decisions.

Applications in Distributional Q-Learning

- Value Distribution Estimation: Distributional Q-learning uses neural networks to approximate value distributions instead of expected values.
- Practical Implementation: Algorithms such as Distributional Deep Q-Networks (Dist-DQN) leverage the distributional Bellman equation for better performance in environments with uncertainty.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : DISTRIBUTIONAL Q-LEARNING

In the context of Distributional Q-learning, we represent a probability distribution over rewards using two numpy arrays. This approach allows us to model the uncertainty in the rewards associated with state-action pairs, providing a more comprehensive understanding of the expected outcomes.

Representing A Probability Distribution in Python

Components of the Representation

1. **Support Array**:

    - This array contains the possible outcomes (i.e., the support of the distribution). It represents the range of values that the rewards can take.
    - For example, if we want to represent rewards ranging from -10 to 10, we can create a support array with evenly spaced values within this range.
2. **Probability Array**:

    - This array holds the probabilities associated with each outcome in the support array. The probabilities must sum to 1, ensuring that they represent a valid probability distribution.
    - Initially, we can start with a uniform distribution where each outcome has an equal probability.

Implementation Steps

1. **Setting Up the Support**:

    - Define the minimum and maximum values for the support.
    - Specify the number of elements in the support array.

```
import numpy as np

vmin, vmax = -10., 10.  # Minimum and maximum values of the support
nsup = 51  # Number of elements in the support
support = np.linspace(vmin, vmax, nsup)  # Create the support array
```

Creating the Probability Distribution:

- Initialize a probability array with equal probabilities for each support element.
- Normalize the probabilities to ensure they sum to 1.

```
probs = np.ones(nsup)  # Start with equal probabilities
probs /= probs.sum()  # Normalize to sum to 1
```

Visualizing the Distribution:

- Use a plotting library (e.g., Matplotlib) to visualize the probability distribution.

```
import matplotlib.pyplot as plt

plt.bar(support, probs)  # Create a bar plot of the distribution
plt.xlabel('Support Values')
plt.ylabel('Probability')
plt.title('Probability Distribution')
plt.show()  # Display the plot
```

Example Code

Here is a complete example that sets up and visualizes a discrete probability distribution:

```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Define the support
vmin, vmax = -10., 10.  # Minimum and maximum values of the support
nsup = 51  # Number of elements in the support
support = np.linspace(vmin, vmax, nsup)  # Create the support array

# Step 2: Create the probability distribution
probs = np.ones(nsup)  # Start with equal probabilities
probs /= probs.sum()  # Normalize to sum to 1

# Step 3: Visualize the distribution
plt.bar(support, probs)  # Create a bar plot of the distribution
plt.xlabel('Support Values')
plt.ylabel('Probability')
plt.title('Probability Distribution')
plt.show()  # Display the plot
```

By representing a probability distribution using numpy arrays, we can effectively model the uncertainty in rewards within the framework of Distributional Q-learning. This representation allows for more nuanced decision-making and learning in reinforcement learning environments.

IMPLEMENTING THE DIST-DQN

The Distributional Deep Q-Network (Dist-DQN) is an extension of the traditional Deep Q-Network (DQN) that models the distribution of returns rather than just the expected value. This approach allows the agent to capture the uncertainty in the rewards, leading to improved performance in reinforcement learning tasks.

Architecture

1. **Input Layer**:
   - The Dist-DQN accepts a state representation as input. In this implementation, a 128-element state vector is used, which can represent various features of the environment (e.g., positions and velocities of game characters).
2. **Hidden Layers**:
   - The state vector is passed through a series of fully connected (dense) layers. These layers learn to extract relevant features from the input state.
3. **Output Layer**:
   - The final layer produces three separate distribution vectors, each corresponding to a different action. Each distribution vector represents the probability distribution over possible returns for that action.
   - The output is structured as a 3×51 matrix, where 3 represents the number of actions and 51 represents the number of discrete support values in the distribution.
4. **Softmax Function**:
   - After obtaining the raw output from the neural network, the softmax function is applied to ensure that the output distributions are valid probability distributions (i.e., they sum to 1).

Implementation Steps

1. **Define the Neural Network**:
    - Create a neural network architecture using a deep learning framework (e.g., PyTorch or TensorFlow). The network should consist of input, hidden, and output layers as described above.
2. **Forward Pass**:
    - Implement the forward pass of the network, which takes the state vector as input and produces the output distribution matrix.
3. **Loss Function**:
    - Define a loss function that measures the difference between the predicted distribution and the target distribution. Common choices include the Kullback-Leibler divergence or the Wasserstein distance.
4. **Training Loop**:
    - Implement the training loop where the Dist-DQN is updated based on the observed rewards and the corresponding target distributions.

Example Code

Here is a simplified example of how to implement the Dist-DQN architecture using PyTorch:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class DistDQN(nn.Module):
    def __init__(self, state_size, action_size, num_support):
        super(DistDQN, self).__init__()
        self.fc1 = nn.Linear(state_size, 128)  # First hidden layer
        self.fc2 = nn.Linear(128, 128)          # Second hidden layer
        self.fc3 = nn.Linear(128, action_size * num_support)  # Output layer

        self.action_size = action_size
        self.num_support = num_support

    def forward(self, x):
        x = F.relu(self.fc1(x))  # Activation for first layer
        x = F.relu(self.fc2(x))  # Activation for second layer
        x = self.fc3(x)          # Output layer
        x = x.view(-1, self.action_size, self.num_support)  # Reshape output
        return F.softmax(x, dim=2)  # Apply softmax to get probabilities

# Example usage
state_size = 128  # Size of the state vector
action_size = 3   # Number of actions
num_support = 51  # Number of discrete support values

model = DistDQN(state_size, action_size, num_support)
```

Training the Dist-DQN

1. **Collect Experience**:
    - Gather experience tuples (state, action, reward, next state) from the environment.
2. **Compute Target Distribution**:
    - For each experience, compute the target distribution based on the observed reward and the next state.
3. **Update the Model**:
    - Use the collected experiences to update the Dist-DQN model by minimizing the loss between the predicted and target distributions.

The implementation of the Dist-DQN allows for a more nuanced approach to reinforcement learning by capturing the distribution of returns. This leads to improved decision-making and performance in complex environments. By following the outlined steps and utilizing the provided code, practitioners can effectively implement and train a Dist-DQN for various reinforcement learning tasks.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : COMPARING PROBABILITY DISTRIBUTIONS

Comparing probability distributions is essential in machine learning tasks where models must generate, predict, or match data distributions to empirical observations. This process underpins generative modeling, reinforcement learning, and training models to reduce prediction errors by aligning generated and real-world distributions.

Probability Distributions:
A probability distribution describes how the probabilities are distributed over the values of a random variable. In machine learning, understanding these distributions is crucial for tasks such as generative modeling, where the goal is to generate data that resembles a given empirical dataset.

Generative Models:
Generative models aim to learn the underlying distribution of a dataset so that they can generate new samples that are similar to the training data. For instance, a generative model trained on celebrity images would learn to produce new images that resemble those in the training set.

Empirical Data vs. Generated Data:
Empirical data refers to real-world data collected from observations, while generated data is produced by the generative model. The objective of training a generative model is to maximize the likelihood that the generated data closely matches the empirical data.

Target Distribution Generation:
A function like get_target_dist generates a target distribution for reinforcement learning tasks based on observed rewards and prior distributions.

Terminal states with specific rewards produce degenerate distributions, while non-terminal states use Bayesian updates for distributions.

Loss Functions for Distribution Comparison:
Standard loss functions like Mean Squared Error (MSE) are unsuitable for probability distributions because they do not measure probabilistic distance.
Loss functions for distributions must minimize differences in probabilistic characteristics, often using metrics like Kullback-Leibler (KL) divergence or cross-entropy loss.

Kullback-Leibler Divergence (KL Divergence):
KL divergence is a measure of how one probability distribution diverges from a second, expected probability distribution. It quantifies the difference between two distributions, denoted as DKL(Q∣∣P), where Q is the true distribution and P is the model distribution.

The KL divergence is defined mathematically as: DKL(Q∣∣P)=∑xQ(x)log(P(x)Q(x))

A lower KL divergence indicates that the two distributions are more similar, while a higher value indicates greater divergence.

Cross-Entropy Loss:
Defined as $H(Q,P)=-\sum Q(x)\log(P(x))$, where $Q(x)$ is the empirical distribution. Optimizes the alignment of the model's predictions with the true data distribution.

Implementation in Neural Networks:
Models, such as Distributional DQN, use cross-entropy to update action-value distributions. Distributional approaches allow capturing uncertainty and variability in predictions.

Code Examples

Generating Target Distribution (get_target_dist):

Python Code

```
def get_target_dist(dist_batch, action_batch, reward_batch, support, lim=(-10, 10), gamma=0.8):

    nsup = support.shape[0]
    vmin, vmax = lim
    dz = (vmax - vmin) / (nsup - 1)
    target_dist_batch = dist_batch.clone()
    for i in range(dist_batch.shape[0]):
        dist_full = dist_batch[i]
        action = int(action_batch[i].item())
        dist = dist_full[action]
        r = reward_batch[i]
        if r != -1:  # Terminal state
            target_dist = torch.zeros(nsup)
            bj = int(np.clip(np.round((r - vmin) / dz), 0, nsup - 1))
            target_dist[bj] = 1.0
        else:  # Non-terminal state
            target_dist = update_dist(r, support, dist, lim=lim, gamma=gamma)
        target_dist_batch[i, action, :] = target_dist
    return target_dist_batch
```

KL Divergence in Python:

```
import numpy as np

def kl_divergence(p, q):
   return np.sum(p * np.log(p / q))
```

Cross-Entropy Loss (lossfn):

```
def lossfn(x, y):
   loss = torch.Tensor([0.])
   loss.requires_grad = True
   for i in range(x.shape[0]):
      loss_ = -1 * torch.log(x[i].flatten(start_dim=0)) @ y[i].flatten(start_dim=0)
      loss = loss + loss_
   return loss
```

Practical Applications

1. Reinforcement Learning:
   - In methods like Distributional DQN, target distributions guide the agent to refine predictions based on observed rewards.
2. Generative Modeling:
   - Models such as GANs or VAEs aim to produce samples that match the empirical distribution, evaluated through metrics like KL divergence or cross-entropy.
3. Simulated Testing:
   - Using synthetic data and predefined parameters, models validate their ability to align predicted and target distributions.

Challenges and Considerations

- Numerical Stability:
   - Direct multiplication of probabilities risks numerical underflow. Using log-probabilities avoids this issue.
- Weighting Samples:
   - Real-world samples may have different probabilities. Weighted loss functions can address this variability.
- Symmetry:
   - Loss functions like KL divergence are directional; choosing $DKL(Q||P)$or $DKL(P||Q)$ depends on the use case.

Comparing probability distributions is a fundamental aspect of machine learning that enables the development of models capable of generating realistic data. By leveraging concepts like KL divergence, practitioners can effectively measure and minimize the differences between distributions, leading to improved model performance and more accurate data generation.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : DIST-DQN ON SIMULATED DATA

The Distributional Deep Q-Network (Dist-DQN) extends standard Q-learning by modeling the full distribution of future rewards instead of just the expected value. This approach improves the robustness and expressiveness of reinforcement learning models. The simulated data experiment demonstrates the functionality and stability of the Dist-DQN algorithm under synthetic conditions.

Cross-Entropy Loss Function

A specialized loss function is implemented to compute the loss between the predicted and target distributions:

- Input Dimensions: The prediction (x) and target (y) distributions are of size $B \times 3 \times 51$, where B is the batch size.
- Flattening: Distributions are flattened along the action dimension to form a $B \times 153$ matrix.
- Loss Calculation: The cross-entropy loss is computed for all action-value distributions to stabilize training by ensuring that distributions of actions not taken remain unchanged.

Cross-Entropy Loss Function

```
def lossfn(x, y):
    loss = torch.Tensor([0.])
    loss.requires_grad = True
    for i in range(x.shape[0]):
        loss_ = -1 * torch.log(x[i].flatten(start_dim=0)) @ y[i].flatten(start_dim=0)
        loss = loss + loss_
    return loss
```

Simulated Data Initialization

- Action Space: Set to 3 actions (e.g., NO-OP, UP, DOWN).
- Parameters:
    - theta: Randomly initialized parameters for the Dist-DQN network.
    - gamma: Discount factor (0.9).
    - lr: Learning rate ($1e{-5}$).
    - update_rate: Synchronization rate for the target network (75 steps).
- State and Support:

- State initialized with two random vectors.
- Support set between [−10,10] with 51 evenly spaced values.

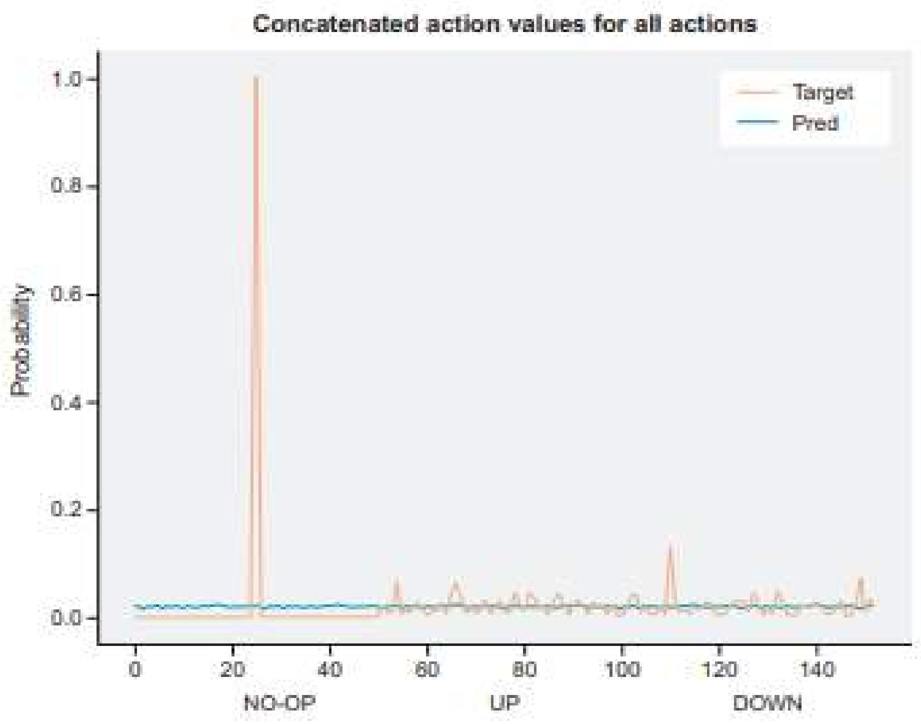


Figure: This shows the predicted action-value distributions produced by an untrained Dist-DQN and the target distribution after observing a reward. There are three separate action-value distributions of length 51 elements, but here they've been concatenated into one long vector to illustrate the overall fit between the prediction and target.

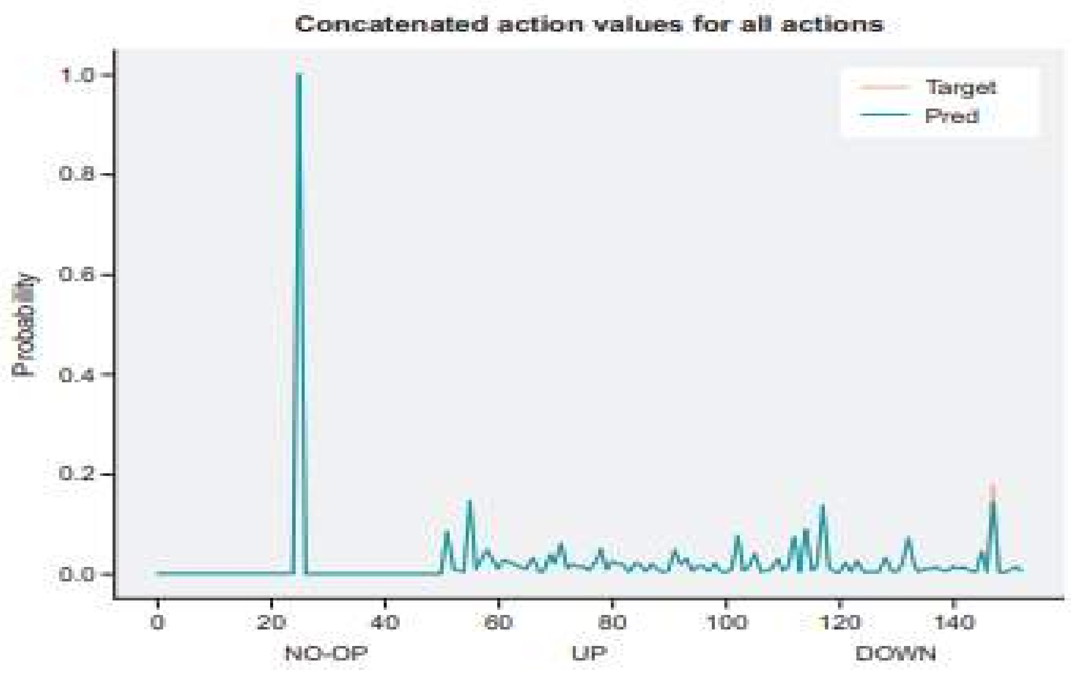
Training Process

The training involves updating the network parameters to minimize the cross-entropy loss between the predicted and target distributions.

- Target Network: A lagged copy of the main network to stabilize training.
- Random Noise: Small noise added to rewards to reduce overfitting.

```
for i in range(1000):
    reward_batch = torch.Tensor([0, 8]) + torch.randn(2)/10.0
    pred_batch = dist_dqn(state, theta, aspace=aspace)
    pred_batch2 = dist_dqn(state, theta_2, aspace=aspace)
      target_dist = get_target_dist(pred_batch2, action_batch, reward_batch, support, lim=(vmin,
vmax), gamma=gamma)
```

```
loss = lossfn(pred_batch, target_dist.detach())
losses.append(loss.item())
loss.backward()
# Gradient Descent
with torch.no_grad():
    theta -= lr * theta.grad
theta.requires_grad = True
if i % update_rate == 0:
    theta_2 = theta.detach().clone()
```



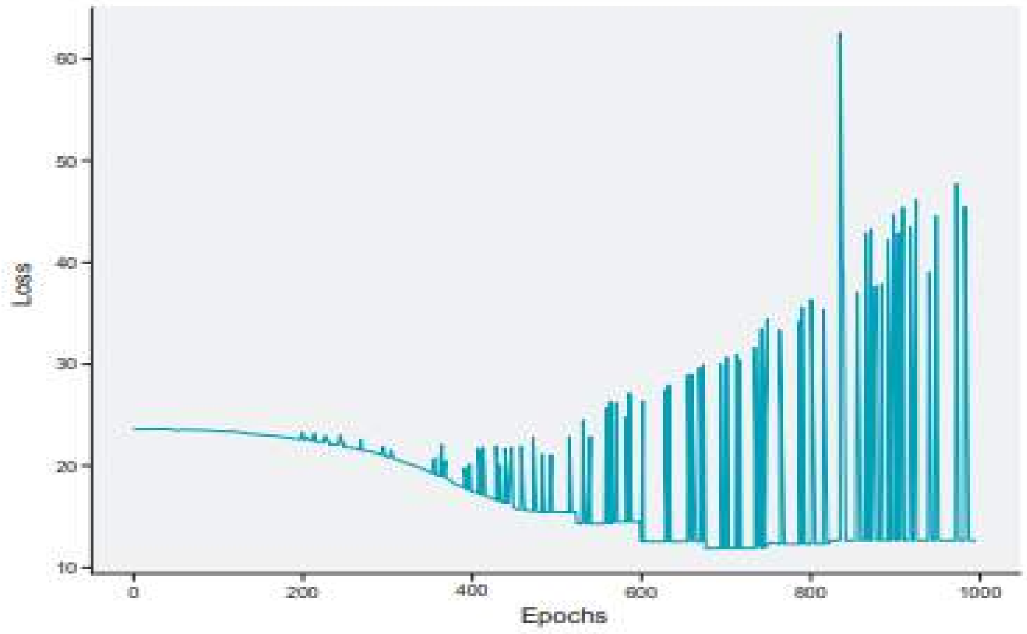


Figure: Top: The concatenated action-value distributions for all three actions after training. Bottom: Loss plot over training time. The baseline loss is decreasing, but we see ever-increasing spikes.

Performance Evaluation

The experiment visualizes the predicted and target distributions and monitors the loss over time:

- Matching Distributions: The predicted distributions align closely with the target after sufficient training.
- Loss Spikes: Periodic spikes in the loss occur due to target network synchronization.

Results

- Initial State: Predictions are uniform distributions, reflecting no knowledge of reward distributions.
- Trained State: Predictions align well with the target distributions, with distinct peaks for corresponding actions.
- Stability: The target network significantly stabilizes training by decoupling parameter updates from target generation.

Visualization

1. Action-Value Distributions:
    - Concatenated distributions for all actions are visualized, demonstrating convergence to the target.

- Separate bar charts for each action-value distribution highlight the network's learning.

2. Loss Curve:
    - Loss decreases over epochs, with noticeable spikes during target synchronization.

```
# Plotting distributions
plt.plot((target_dist.flatten(start_dim=1)[0].data.numpy()), color='red', label='target')
plt.plot((pred_batch.flatten(start_dim=1)[0].data.numpy()), color='green', label='pred')
plt.legend()

# Loss Plot
plt.plot(losses)
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

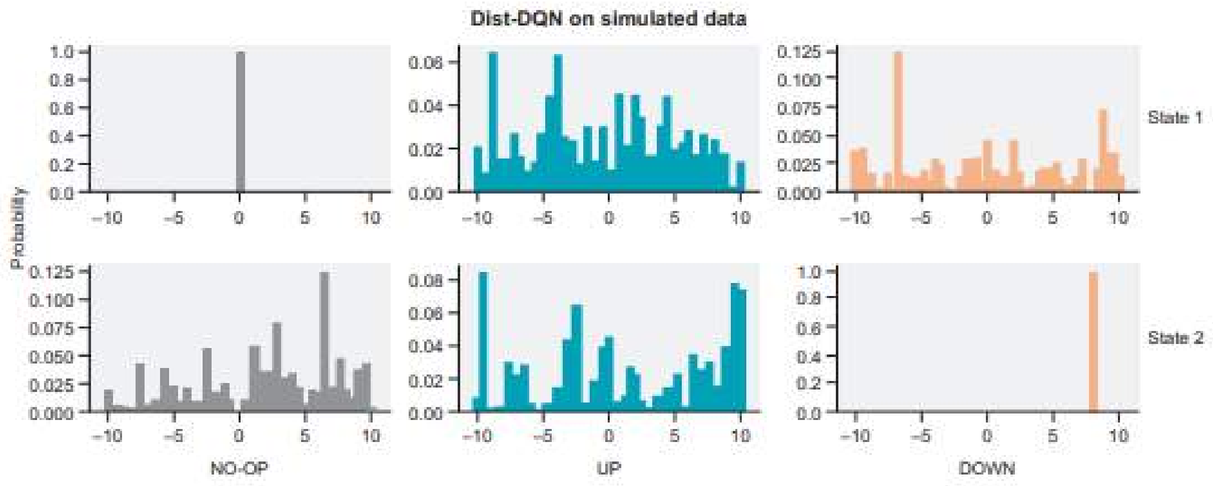


Figure : Each row contains the action-value distributions for an individual state, and each column in a row is the distribution for actions 0, 1, and 2 respectively

The Dist-DQN on simulated data experiment validates the algorithm's ability to:

1. Learn and predict full reward distributions.
2. Utilize target networks to stabilize training.
3. Match predictions to synthetic reward distributions effectively.

This foundational test sets the stage for deploying Dist-DQN in real-world environments, such as playing Atari games.

SUBJECT NAME: REINFORCEMENT LEARNING
UNIT-4
TOPIC : USING DISTRIBUTIONAL Q-LEARNING TO PLAY *FREEWAY*

This documentation outlines the implementation of distributional Q-learning (Dist-DQN) to play the game Freeway. Distributional Q-learning is an enhancement of traditional Q-learning that predicts the full distribution of action values rather than just the expected value. This approach has shown improved performance in various reinforcement learning tasks.

Distributional Q-Learning

- **Definition**: Distributional Q-learning involves learning a probability distribution over action values instead of a single expected value. This allows the agent to make more informed decisions based on the risk associated with different actions.
- **Advantages**: Improved performance, better generalization, and the ability to utilize risk-sensitive policies.

Environment Setup

- **OpenAI Gym**: The Freeway environment is accessed through the OpenAI Gym library, which provides a standardized interface for various reinforcement learning environments.
- **State Representation**: The state is preprocessed and converted into a PyTorch tensor, normalized to ensure that the gradient sizes remain manageable.

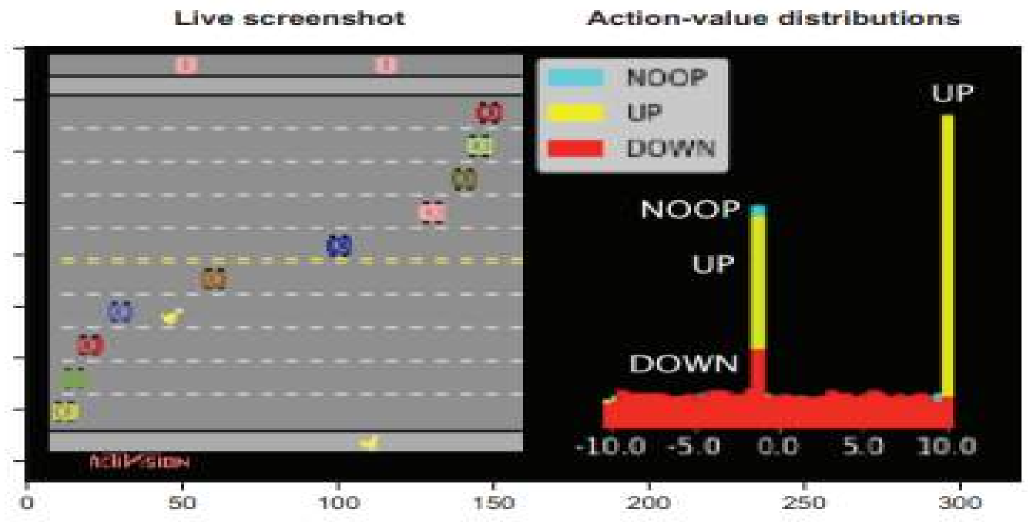


Figure: Left: Screenshot of live gameplay in Atari Freeway. Right: The corresponding action-value distributions of each of the each actions overlaid. The spike on the right corresponds to the UP action and the spike on the left corresponds mostly to the NO-OP action. Since the right spike is larger, the agent is more likely to take the UP action, which seems like the right thing to do in this case. It is difficult to see, but the UP action also has a spike on top of the NO-OP spike on the left, so the UP action-value distribution is bimodal, suggesting that taking the UP action might lead to either a –1 reward or a +10 reward, but the +10 reward is more likely since that spike is taller.

Implementation Steps

Initialization

- **Hyperparameters**: Set the necessary hyperparameters such as learning rate, discount factor, replay size, and epsilon values for the epsilon-greedy policy.
- **Action Space**: Define the action space for the Freeway environment.

```
import gym
from collections import deque
import torch

env = gym.make('Freeway-ram-v0')
aspace = 3  # Number of actions
vmin, vmax = -10, 10  # Value range for action-value distributions
replay_size = 200  # Size of the experience replay buffer
batch_size = 50  # Batch size for training
nsup = 51  # Number of supports for the distribution
dz = (vmax - vmin) / (nsup - 1)  # Step size for supports
support = torch.linspace(vmin, vmax, nsup)  # Supports
replay = deque(maxlen=replay_size)  # Experience replay buffer
lr = 0.0001  # Learning rate
gamma = 0.1  # Discount factor
epochs = 1300  # Number of training epochs
eps = 0.20  # Initial epsilon for exploration
eps_min = 0.05  # Minimum epsilon
priority_level = 5  # Prioritized replay level
update_freq = 25  # Frequency of target network updates
```

State Preprocessing

States are normalized to have values between 0 and 1, which stabilizes gradients during training. This is implemented as:

```
def preproc_state(state):
    p_state = torch.from_numpy(state).unsqueeze(dim=0).float()
    p_state = torch.nn.functional.normalize(p_state, dim=1)
    return p_state
```

Action Selection

The policy selects actions based on the expected value of predicted action-value distributions. Expected values are computed as the inner product of the distribution support and its probabilities:

```
def get_action(dist, support):
    actions = []
    for b in range(dist.shape[0]):
        expectations = [support @ dist[b, a, :] for a in range(dist.shape[1])]
        action = int(np.argmax(expectations))
        actions.append(action)
    return torch.Tensor(actions).int()
```

Experience Replay with Prioritization

To accelerate learning, the replay buffer prioritizes rare but significant experiences (e.g., winning or losing states). This helps the agent focus on learning impactful game dynamics.

Hyperparameters and Setup

- Environment: Freeway-ram-v0 from OpenAI Gym.
- Support Range: Values between -10 and +10 with 51 discrete steps.
- Epsilon-Greedy Strategy: Decreasing epsilon over epochs to balance exploration and exploitation.
- Replay Buffer: Fixed-size deque with prioritized replay for rare events.

```
import gym
from collections import deque

env = gym.make('Freeway-ram-v0')
aspace = 3
vmin, vmax = -10, 10
nsup = 51
support = torch.linspace(vmin, vmax, nsup)

replay = deque(maxlen=200)
lr, gamma = 0.0001, 0.1
eps, eps_min = 0.20, 0.05
priority_level = 5
update_freq = 25
```

Main Training Loop

1. Action Selection: Choose either a random action or one based on the policy.
2. Environment Interaction: Take an action, receive the next state and reward.
3. Reward Adjustment: Encode game-specific rewards (+10 for crossing, -10 for losing, -1 for continuing).
4. Experience Replay: Store transitions in the replay buffer, duplicating priority experiences.
5. Gradient Descent: Update the Dist-DQN parameters using the Bellman loss.
6. Target Network Updates: Periodically synchronize the target network with the main network.
7. Epsilon Decay: Gradually reduce exploration.

```
import gym

from collections import deque

env = gym.make('Freeway-ram-v0')
aspace = 3
vmin, vmax = -10, 10
nsup = 51
support = torch.linspace(vmin, vmax, nsup)

replay = deque(maxlen=200)
lr, gamma = 0.0001, 0.1
eps, eps_min = 0.20, 0.05
priority_level = 5
update_freq = 25

theta = torch.randn(128*100 + 25*100 + aspace*25*51) / 10
theta.requires_grad = True
theta_2 = theta.detach().clone()
```

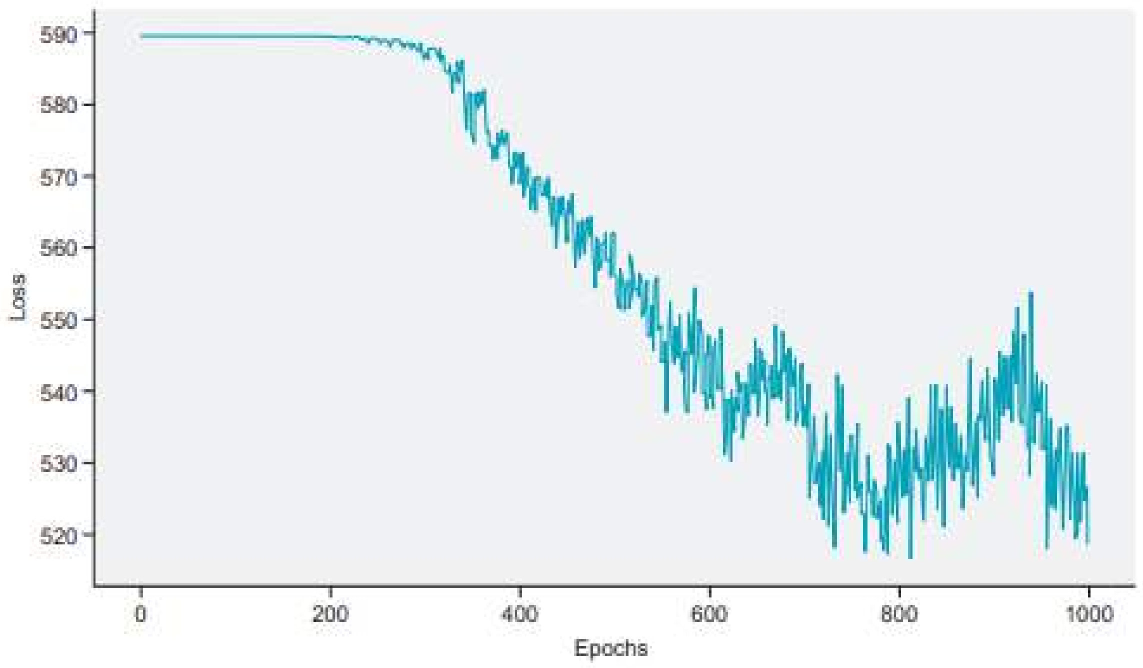


Figure :The loss plot for training Dist-DQN on the Atari game Freeway. The loss gradually declines but has significant “spikiness” due to the periodic target network updates.

Observations and Results

- Reward Distribution: The agent successfully learns to prioritize actions that lead to a +10 reward.
- Loss Dynamics: Loss declines over epochs but exhibits spikes due to target network updates.
- Action-Value Distributions: The UP action's bimodal distribution, with peaks at -1 and +10, guides the agent effectively.
- Performance: Achieving four or more successful crossings per episode indicates a well-trained model.