1. REINFORCE

Summary: REINFORCE introduces the fundamental innovation of the policy gradient method, allowing direct optimization of policies by calculating gradients of expected rewards. This approach shifts from value-based methods, simplifying learning in environments where value function approximation is challenging. REINFORCE is well-suited for episodic tasks, particularly those with continuous or discrete action spaces. The algorithm uses a Monte Carlo approach, updating the policy after each episode based on cumulative return, with the gradient computed via the log-likelihood of actions. A baseline can be introduced to reduce variance and stabilize learning.

Objective:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\right]$$

Policy Gradient:

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

where $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ is the return from time t.

Policy Update:

$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta J(\theta)$$

Variance Reduction (Optional Baseline):

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

Networks Used:

- **Policy Network** (Online)

Hyperparameters:

- Learning Rate (α): $1e^{-3}$ to $1e^{-4}$
- Discount Factor (γ): 0.99

2. DQN, DDQN, Prioritized Replay DDQN

DQN Summary: Deep Q-Network (DQN) was a fundamental innovation that combined Q-learning with deep neural networks to approximate the Q-value function for complex environments. It introduced experience replay and a target network to stabilize training and prevent the divergence of Q-values. DQN was developed to address the challenge of scaling Q-learning to high-dimensional state spaces, such as those found in Atari games. It is appropriate for discrete action spaces with large state spaces. DQN deals with states and actions by approximating the Q-values using a neural network, updating the network parameters via backpropagation using a loss function based on the Bellman equation.

DDQN Summary: Double DQN (DDQN) addresses the overestimation bias present in DQN by decoupling the action selection and value evaluation steps, thereby improving the stability and accuracy of Q-value estimates. The fundamental innovation in DDQN was to reduce the overestimation of Q-values by using two separate networks for selecting actions and another for evaluating the Q-values. This modification makes DDQN more suitable for environments where Q-value estimates have high variance. The update rule in DDQN: For the next state, we select the action with the max q-value from the Q-Network (online) network and evaluate it against the other target (offline) network.

Prioritized DDQN Summary: Prioritized Experience Replay DDQN enhances DDQN by focusing the learning process on more important experiences (i.e., experiences with high temporal difference (TD) error). This innovation makes learning more efficient by prioritizing the replay of transitions that are more informative. It is particularly effective in environments where some experiences contribute more significantly to learning than others. Prioritized replay selects experiences based on their TD error, leading to faster convergence and better performance.

DQN Loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a)\right)^2\right]$$

DDQN Loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma Q_{\theta^-}(s', \text{argmax}_{a'} Q_\theta(s', a')) - Q_\theta(s, a)\right)^2\right]$$

Prioritized Replay DDQN Loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\frac{p_i}{\text{Priority\_Sum}}(\text{Same as DDQN Loss Error})\right]$$

Networks Used:

- **Q-Network** (θ, Online)
- **Target Q-Network** (θ^-, Offline)

Hyperparameters:

- Learning Rate (α): $1e^{-4}$ to $1e^{-3}$
- Discount Factor (γ): 0.99
- Batch Size: 32 to 64
- Target Network Update Frequency: 1000
- Replay Buffer Size: $1e^5$ to $1e^6$
- Priority Exponent (β): 0.4 to 1.0
- Priority Scaling Factor (ϵ): $1e^{-5}$

3. A2C/A3C

A2C Summary: Advantage Actor-Critic (A2C) introduces the idea of combining the actor-critic architecture with the advantage function to reduce variance in policy gradient updates. The advantage function provides a more stable learning signal, which improves the efficiency and stability of policy learning. A2C is appropriate for environments requiring stable policy optimization. It manages states and actions by using a value function (critic) to estimate the advantage, which is then used to update the policy (actor).

A3C Summary: Asynchronous Advantage Actor-Critic (A3C) extends A2C by running multiple agents in parallel, each exploring different parts of the environment. The agents update a global model asynchronously, which leads to faster learning and greater robustness. A3C was developed to address the inefficiencies of single-threaded learning, making it particularly suitable for complex environments where parallel exploration can speed up the learning process. The update rule for A3C is similar to A2C but involves asynchronous updates across multiple agents.

Actor Objective:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_t, a_t}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)\right]$$

Actor is updated using stochastic gradient ascent using the objective.

Critic Loss:

$$L(\phi) = \mathbb{E}_{s_t}\left[\left(R_t - V_\phi(s_t)\right)^2\right]$$

Critic is updated using stochastic gradient descent using the loss.

$$\phi = \phi - \alpha \nabla_\phi L(\phi)^2$$

Networks Used:

- **Actor Network** (θ, Online)
- **Critic Network** (ϕ, Online)
- **Multiple Workers (A3C)**

Hyperparameters:

- Actor Learning Rate (α): $1e^{-4}$
- Critic Learning Rate (β): $1e^{-3}$
- Discount Factor (γ): 0.99
- Number of Workers (A3C): 16 to 32

4. DPG, DDPG

DPG Summary: Deterministic Policy Gradient (DPG) introduces the innovation of using deterministic policies instead of stochastic ones, allowing the algorithm to directly optimize the expected return with respect to deterministic actions. This is particularly useful in continuous action spaces where stochastic policies may be inefficient. DPG is suitable for problems involving continuous action spaces. It deals with states, actions, and rewards by optimizing a deterministic policy, using the deterministic policy gradient to update the policy.

DDPG Summary: Deep Deterministic Policy Gradient (DDPG) extends DPG by incorporating deep neural networks for function approximation, experience replay, and target networks. This allows DDPG to scale to high-dimensional state and action spaces, making it well-suited for tasks like robotic control. DDPG handles states, actions, and rewards by using an actor-critic architecture, where the actor outputs deterministic actions, and the critic evaluates them using the Q-function.

DPG/DDPG Actor Update:

$$\nabla_\phi J(\phi) = \mathbb{E}_s\left[\nabla_a Q_\theta(s, a)\big|_{a=\mu_\phi(s)} \nabla_\phi \mu_\phi(s)\right]$$

DDPG Critic Loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma Q_{\theta^-}(s', \mu_{\phi^-}(s')) - Q_\theta(s, a)\right)^2\right]$$

Networks Used:

- **Actor Network** (ϕ, Online)
- **Critic Network** (θ, Online)

- **Target Actor Network** (ϕ^-, Offline)
- **Target Critic Network** (θ^-, Offline)

Hyperparameters:

- Actor Learning Rate (α): $1e^{-4}$ to $1e^{-3}$
- Critic Learning Rate (β): $1e^{-3}$
- Discount Factor (γ): 0.99
- Replay Buffer Size: $1e^5$ to $1e^6$
- Batch Size: 100
- Soft Update Rate (τ): 0.005
- Target Policy Noise: 0.2

5. TRPO, PPO

TRPO Summary: Trust Region Policy Optimization (TRPO) innovates by introducing a constraint on the policy update to ensure it stays within a trust region, thereby preventing large, destabilizing updates. This approach addresses the problem of maintaining stability in policy learning while still making meaningful progress. TRPO is suitable for complex environments where policy stability is critical. It handles states, actions, and rewards through a constrained optimization process, ensuring that updates do not deviate too far from the previous policy.
PPO Summary: Proximal Policy Optimization (PPO) simplifies TRPO's trust region approach by using a clipped objective function instead of a hard constraint, which makes the algorithm easier to implement and tune. PPO solves the problem of maintaining stability in policy updates with a more practical and scalable approach. It is appropriate for both discrete and continuous action spaces. PPO deals with states, actions, and rewards by balancing exploration and exploitation through a clipped objective function.
TRPO Objective:

$$\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A^{\pi_{\theta_{\text{old}}}}(s,a) \right]$$

PPO Objective:

$$\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}} \left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t \right) \right]$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

Networks Used:

- **Policy Network** (Online)
- **Value Network** (Online)

Hyperparameters:

- Learning Rate (α): $1e^{-4}$ to $3e^{-4}$
- Discount Factor (γ): 0.99
- KL Divergence Constraint (δ): 0.01 to 0.02 (TRPO only)
- Clip Range (ϵ): 0.1 to 0.2 (PPO only)
- Number of Epochs: 3 to 10
- Batch Size: 64 to 2048

6. MCTS

Summary: Monte Carlo Tree Search (MCTS) innovates by combining tree search methods with Monte Carlo simulations to explore future states and make decisions based on empirical outcomes. This method addresses decision-making in complex environments by systematically exploring and evaluating potential outcomes. MCTS is suitable for games and decision-making problems with large, complex state spaces. It handles states and actions by constructing a search tree and evaluating potential actions using simulations, balancing exploration and exploitation through the Upper Confidence Bound (UCB1) formula.
UCT (Upper Confident Trees) Formula:

$$Q(s,a) = \frac{w(s,a)}{n(s,a)} + c\sqrt{\frac{\ln N(s)}{n(s,a)}}$$

Networks Used:

- **Search Tree** (Online)

Hyperparameters:

- Exploration Constant (c): Empirical
- Number of Simulations: 1000 to 10000
- Tree Depth: Problem-specific

7. AlphaGo, AlphaZero

AlphaGo Summary: AlphaGo combines Monte Carlo Tree Search (MCTS) with deep neural networks to evaluate states and select moves in the game of Go, tackling the immense complexity of the game. The innovation of using deep learning to guide and improve MCTS allowed AlphaGo to master Go, a game previously considered too complex for traditional AI approaches. It is suitable for board games with large state spaces. AlphaGo manages states and actions by using neural networks to evaluate board positions and MCTS to decide the best moves.
AlphaZero Summary: AlphaZero generalizes the AlphaGo approach by combining MCTS with deep learning trained entirely through self-play, making it applicable to multiple board games without any domain-specific knowledge. This innovation created a versatile system capable of mastering various games by learning from scratch. AlphaZero is appropriate for a broad range of board games and other decision-making problems. It handles states, actions, and rewards by using self-play to generate training data, which is then used to train neural networks combined with MCTS for decision-making.
Loss Function:

$$L(\theta) = \mathbb{E}_{(s,\pi,z)\sim D}\left[(z - V(s;\theta))^2 - \pi \cdot \log \pi(a|s;\theta) + \lambda \|\theta\|^2 \right]$$

Networks Used:

- **Policy Network** (Online)
- **Value Network** (Online)

Hyperparameters:

- Learning Rate (α): $1e^{-3}$ to $1e^{-2}$
- Discount Factor (γ): 1.0
- MCTS Simulations: Several thousand
- Temperature: Annealed over time
- Regularization Term (λ): $1e^{-4}$ to $1e^{-3}$

8. Advantage Function

Summary: The Advantage function $A(s,a)$ estimates how much better it is to take a specific action a in state s, compared to the average action in that state. It can be defined as:

$$A(s,a) = Q(s,a) - V(s)$$

The advantage function helps in reducing variance and is often used in Actor-Critic methods to stabilize learning.

9. Stochastic Gradient Descent

Summary: SGD is a method used to minimize an error function (like loss or TD-error) $\nabla_w Error^2$ through backpropagation. The update rule for SGD is:

$$w \leftarrow w - \alpha \nabla_w Error^2$$

Where α is the learning rate. The objective is to find the local minima of the error function by adjusting the weights in the direction of the steepest descent. **Ascent Variation:** Gradient ascent is used to maximize a function, such as an objective or reward function. Instead of minimizing the error, gradient ascent increases the return or reward by updating the weights in the direction of the steepest ascent:

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

This technique is commonly used in reinforcement learning to optimize policy parameters.

10. Value Function Approximation

Summary: Value Function Approximation (VFA) is a technique used to represent value functions in environments with large state and action spaces, where tabular methods become impractical. By using function approximation, we can generalize across states or state-action pairs; $\hat{v}(s,w)$ or $\hat{q}(s,a,w)$, where the parameter w is updated using learning methods. This reduces memory and computational requirements.
Function Approximation Methods:

- **Linear Function Approximation:** The value function is represented as a linear combination (dot product) of features:

  $$\hat{v}(s,w) = w^T x(s) = \sum_{i=1}^{n} w_i \cdot x_i(s)$$

  where w is the weight vector, and $x(s)$ is the feature vector (some property or aspect of state s).

- **Common Approximators:** Linear functions, Neural Networks, Decision Trees, Nearest Neighbors, and Fourier/Wavelet Bases.

Control with Function Approximation: VFA can be used in both Monte Carlo and Temporal Difference (TD) methods. The function approximator helps in learning the value function incrementally, making the learning process more efficient in large state-action spaces.

11. Neural Networks (CNN, RNN, L1/L2 Regularization, Overfitting, Optimizers, Pooling)

Summary:

12. General Notes on Value-based, Policy-based, and Actor-Critic Methods

Summary:

13. Objective Function

Summary: The objective function $J(\theta)$ represents the expected cumulative return. The goal is to find parameters θ that maximize it. It is often defined as:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \sum_\tau P(\tau;\theta)R(\tau)$$

Where τ represents the trajectory, which is the sequence of states and actions. The cumulative return $R(\tau)$ is taken over all possible trajectories, and $P(\tau;\theta)$ represents the probability of the possible trajectory under policy π_θ.

14. Trajectories

Summary:

15. Experience Replay & Prioritized Experience Replay

Experience Replay: This technique stores the agent's experiences in a replay buffer, allowing the agent to learn from a (uniformly) randomly sampled subset of past experiences to break correlations between consecutive samples.

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

Prioritized Experience Replay: This variant assigns higher sampling probability to experiences with larger Temporal Difference (TD) errors, making learning more efficient by focusing on more informative experiences. The idea is that not all experiences are equally valuable. Some experiences (like those involving significant errors in prediction) can provide more informative updates to the agent's policy.

16. REINFORCE Baseline

Summary: The baseline $b(s_t)$ is subtracted from the return $R(\tau)$ to reduce the variance of the gradient estimates. This makes the learning process more stable and efficient. The policy gradient with baseline is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)(R(\tau) - b(s_t))\right]$$

Adding a baseline does not introduce any bias into the estimates of the policy gradient since it does not affect the expectation of the gradient estimates because it is usually chosen to be independent of the action taken

17. Soft Update DQN (τ)

Summary: In DQN, a soft update is used to gradually update the target network parameters θ^- using the online network parameters θ:

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$$

Where τ is a small number, typically $\tau = 0.005$, ensuring smooth updates and preventing instability during training. This differs from a hard update, where the target network is updated directly with the online network parameters. $\theta^- \leftarrow \theta$

18. Policy Gradient Theorem

Summary: The Policy Gradient Theorem provides a way to maximize the objective function $J(\theta)$ by **estimating the gradient** $\nabla_\theta J(\theta)$ as $\hat{g}$

$$\nabla_\theta J(\theta) \approx \hat{g} = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right]$$

Key Points:

- The gradient $\nabla_\theta \log \pi_\theta(a_t|s_t)$ represents the direction of the steepest increase in the log probability of selecting action a_t at state s_t.
- This gradient tells us how to adjust the policy weights to increase or decrease the log probability of choosing action a_t at state s_t.
- The reward $R(\tau)$ influences the update direction—high rewards increase the log probability of the (state, action) combination, while low rewards decrease it.

Policy Update:

$$\theta \leftarrow \theta + \alpha\hat{g}$$

$\hat{g}$ serves as an approximation of the true policy gradient $\nabla_\theta J(\theta)$, which is computationally expensive because of calculating the probability of each possible trajectory. $\hat{g}$ is computed using sampled trajectories.