1. REINFORCE (On-Policy)

Summary:

- Innovation: policy gradient method. Allows direct optimization of policies by calculating gradients of expected rewards
- Shift from value-based methods, simplifying learning in environments where VFA is challenging
- Good for episodic tasks; continuous or discrete action spaces.
- Uses Monte Carlo approach; update policy after each episode completes based on cumulative return/objective function. (Calculate gradient based on 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), Discount Factor (γ)

2. DQN, DDQN, + Prioritized (Off-Policy)

DQN Summary:

- Combined Q-learning with deep neural networks to approximate the Q-value function for complex environments; scaling Q-learning to high-dimensional state spaces like Atari games
- Introduced experience replay and a target network to stabilize training and prevent the divergence of Q-values
- Appropriate for discrete action spaces with large state spaces
- Approximates Q-values using a neural network, updating the network parameters via backpropagation using a loss function based on the Bellman equation

DDQN Summary:

- Fixes overestimation bias in DQN by decoupling the action selection and value evaluation steps. Improves stability and accuracy of Q-value estimates
  - Innovation: Using two separate networks for selecting actions and another for evaluating Q-values
  - More suitable for environments where Q-value estimates have high variance

DQN Loss:

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

"For the next state, we select the action with the max q-value from the target (offline) network, and evaluate it against the same target (offline) network"

DDQN Loss:

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

"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 Loss:

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

Networks Used:

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

Hyperparameters: Learning Rate (α), Discount Factor (γ), Batch Size, Target Network Update Frequency, Replay Buffer Size, Priority Exponent (β), Priority Scaling Factor (ϵ)

3. A2C/A3C (On-Policy)

A2C Summary:

- Innovation: Combine actor-critic architecture with advantage function to reduce variance in policy gradient updates
  - Provides more stable learning signal; improves the efficiency and stability of policy learning
- Appropriate for environments requiring stable policy optimization
- Uses value function (critic) to estimate the advantage, which is then used to update the policy (actor)

Asynchronous Advantage Actor-Critic (A3C) Summary:

- Extends A2C: runs multiple agents in parallel, each exploring different parts of the environment
- Improvement: Inefficiencies of single-threaded A2C, so good for complex environments where parallel exploration speeds up the learning process. Agents update a global model asynchronously

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 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), Critic Learning Rate (β), Discount Factor (γ), Number of Workers (A3C)

4. DPG, DDPG (Off-Policy Actor-Critic Methods)

DPG Summary:

- Innovation: Use deterministic policies instead of stochastic ones; allows algorithm to directly optimize the expected return with respect to deterministic actions
- Useful in continuous action spaces where stochastic policies may be inefficient
- Operates with one deterministic policy and a critic that evaluates the actions chosen by this policy.

DDPG Summary:

- Extends DPG: Uses deep neural networks for function approximation, experience replay, and target networks
- Scales to high-dimensional state and action spaces

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), Critic Learning Rate (β), Discount Factor (γ), Replay Buffer Size, Batch Size, Soft Update Rate (τ), Target Policy Noise

5. TRPO, PPO (On-Policy)

TRPO (Trust Region Policy Optimization) Summary:

- Innovation: Hard constraint on the policy update to ensure it stays within a trust region; prevents large, destabilizing updates
- Adresses maintaining stability in policy learning while still making meaningful progress
- Good for complex environments where policy stability is critical
- Ensures updates do not deviate too far from the previous policy

PPO (Proximal Policy Optimization) Summary:

- Improves TRPO's trust region approach by using a clipped objective function instead of a hard constraint
  - Makes the algorithm easier to implement and tune; more practical and scalable
  - Instead of ignoring updates outside trust region, clips them to be within a certain range (max or min, not ignored)
- Good for discrete and continuous action spaces
- Balances exploration and exploitation with the 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), Discount Factor (γ), KL Divergence Constraint (δ) (TRPO only), Clip Range (ϵ) (PPO only), Number of Epochs, Batch Size

6. MCTS (Off-Policy)

Summary:

- Innovation: Combine tree search methods with Monte Carlo simulations to explore future states and make decisions based on empirical outcomes
- Addresses decision-making in complex environments by systematically exploring and evaluating potential outcomes

- Suitable for games and decision-making problems with complex state spaces

- Constructs a search tree and follows: Selection, Expansion, Simulation, Backpropagation

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, Tree Depth: Problem-specific

7. AlphaGo, AlphaZero (Off-Policy)

AlphaGo Summary:

- Innovation: Uses MCTS paired with deep neural networks to enhance MCTS

- Effective for board games with very large state spaces

AlphaZero Summary:

- Innovation: Extends AlphaGo by combining MCTS with deep neural networks trained entirely through self-play (generate its own data to train itself)

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), Discount Factor (γ), MCTS Simulations, Temperature: Annealed over time, Regularization Term (λ)

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 (VFA)

Summary: 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.

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, Regularization, Overfitting, Pooling)

Convolutional Neural Network (CNN) are commonly used and use convolutions instead of matrix multiplications to exploit spatial structure in data. They are not fully connected like a standard neural network to save on computation.

Recurrent Neural Networks (RNN) are designed to handle sequential data by maintaining a hidden state that captures temporal dependencies.

Regularization techniques like L1 and L2 regularization prevent overfitting by adding a penalty term to the loss function. Overfitting occurs when a model learns the training data too well, leading to poor generalization.

Usage

- L1: Add $\lambda|w|$ to each weight error function (encourages all weights to zero)
- L2 (Weight Decay): Augment the error function with $\frac{1}{2}\lambda w^2$ encourages small weights/to use all inputs instead of relying on strongest signal

Pooling is used in CNNs to reduce the spatial dimensions of the input and improve invariance, for example we combine/pool nearby neurons to reduce the size of the representation, like taking the max or average of a group of neurons.

12. 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.

13. 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.

14. 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

15. 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. $\theta^- \leftarrow \theta$

16. 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.

17. Importance Sampling

A technique used to estimate the properties of a particular distribution, while only having samples generated from a different distribution than the one of interest. This approach can greatly improve the efficiency of Monte Carlo simulations by reducing variance, especially when dealing with high-dimensional data.

18. Bootstrapping

Bootstrapping methods mean that it updates the value estimates based on other estimated values, rather than waiting for the final outcome (trajectory) of the episode.

19. Bellman Equation

$$v_\pi(s) = \sum_{a\in A} \pi(a|s)\left[\sum_{s'\in S} p(s'|s,a)[r + \gamma v_\pi(s')]\right]$$

Probability we take specific action a in state s (policy) for all actions. Probability of we actually end up in state s' if we take action a in state s (deterministic or not).

20. Policy Improvement Theorem

We can always greedify to obtain a better policy π', where ”better” is $q_{\pi'}(s,a) \geq q_\pi(s,a)$ for all s, a. Basically keep greedifying until π is the same as π':

$$\pi'(s) = \arg\max_a q_\pi(s,a)$$

21. Eligibility Traces

Summary: Eligibility traces are a mechanism used to bridge the gap between Monte Carlo methods (which use complete returns) and Temporal Difference (TD) methods (which use one-step returns). They allow for a mix between these two extremes, enabling more efficient learning by considering a broader range of rewards over multiple steps.

- In Monte Carlo, every state-action pair that occurs in an episode is updated based on the complete return.
  - Every state gets $\frac{r}{t}$ reward (equal) for each time step t in the episode. Equivalent to TD($\lambda = 1$)
- In TD($\lambda = 0$), when terminal state reached, only the immediate state-action pair that led to the terminal state is updated with the reward.
- In TD($\lambda = 0.9$), the immediate state-action pair that led to the terminal state gets the full reward, but previous state-action pairs get decayed reward (scaled by λ).
- Also has a threshold for the minimum eligibility trace value, below which the trace (and reward) is set to zero.

22. N-Step Prediction

Summary: Used in TD learning methods, to update the value of a state by looking ahead multiple steps in the future.

- 1-step equivalent to TD(0), need to only look ahead one step to update the value of a state.
- ∞-step is equivalent to Monte Carlo, need to look ahead to the end of the episode to update the value of a state.

23. SARSA: On-Policy TD Control

- Initialize $Q(s,a)$ for all $s \in S$, $a \in A(s)$ arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$
- Repeat (for each episode):
  - Initialize S
  - Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  - Repeat (for each step of episode):
    - Take action A, observe R, S'
    - Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    - $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    - $S \leftarrow S'$; $A \leftarrow A'$
  - until S is terminal

24. Expected SARSA: On-Policy TD Control

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A\right.$$

- $\pi(a|s) = 0.9$ for max a, $\frac{0.1}{|A|-1}$ for others

25. Q-Learning: Off-Policy TD Control

- Initialize $Q(s,a)$ for all $s \in S$, $a \in A(s)$ arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$
- Repeat (for each episode):
  - Initialize S
  - Repeat (for each step of episode):
    - Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    - Take action A, observe R, S'
    - $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    - $S \leftarrow S'$
  - until S is terminal

26. Double Q-Learning

- Initialize $Q_1(s,a)$ and $Q_2(s,a)$ for all $s \in S$, $a \in A(s)$ arbitrarily
- Initialize $Q_1(\text{terminal-state}, \cdot) = Q_2(\text{terminal-state}, \cdot) = 0$
- Repeat (for each episode):
  - Initialize S
  - Repeat (for each step of episode):
    - Choose A from S using policy derived from Q_1 and Q_2 (e.g., ϵ-greedy in $Q_1 + Q_2$)
    - Take action A, observe R, S'
    - With 0.5 probability:
      - $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha[R + \gamma Q_2(S', \arg\max_a Q_1(S',a)) - Q_1(S,A)]$
    - else:
      - $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha[R + \gamma Q_1(S', \arg\max_a Q_2(S',a)) - Q_2(S,A)]$
    - $S \leftarrow S'$
  - until S is terminal