Reinforcement Learning Briefing Document

1. Introduction to Reinforcement Learning (RL)

Reinforcement Learning is a paradigm within artificial intelligence where an autonomous agent learns to make optimal decisions by interacting with an environment. It differs from supervised and unsupervised learning by focusing on an agent learning through trial and error, guided by a "reward signal" rather than explicit labels or inherent data structures. The core objective of an RL agent is to discover a "policy" that maximizes the "expected cumulative long-term rewards."

1.1 Markov Decision Processes (MDPs)

Reinforcement Learning problems are formally modeled using Markov Decision Processes (MDPs), a mathematical framework for sequential decision-making in uncertain environments. An MDP is defined by a five-tuple: (S, A, P, R, γ).

- States (S): "The set of all possible situations or conditions the agent can be in." This defines the agent's perception of the world. For example, in Lunar Lander, the state is an "8-dimensional vector: lander's x/y position, x/y velocity, angle, angular velocity, leg contact booleans." In CartPole, it's a "4-dimensional continuous vector comprising the cart's position, cart's velocity, pole's angle, and pole's angular velocity."
- Actions (A): "The set of all possible moves or decisions the agent can make from any given state." These are the agent's means of influencing the environment. Actions can be discrete (e.g., "applying a fixed force of either +1 or -1 to a cart" in CartPole, or "do nothing, fire main engine, fire left engine, fire right engine" in Lunar Lander) or continuous (e.g., a "4-dimensional continuous vector, with each dimension representing a motor speed value in the range [-1, 1]" for BipedalWalker).
- Transition Model (P): "$P(s' | s, a)$ defines the probability of transitioning from the current state s to a new state s′ after taking action a." This component models the uncertainty or stochasticity of the environment. In Lunar Lander, this is "The physics of the Lunar Lander environment (how thruster fires change position, velocity, angle)."
- Reward Function (R): "Provides the immediate scalar reward received after taking action a in state s (and potentially transitioning to s′)." Positive rewards indicate good outcomes, negative indicate penalties. Examples include "+1 for reaching a goal, -1 for stepping into fire, -0.1 for each step to encourage shorter paths" in GridWorld, "+100 for successful landing, -100 for crashing, -0.3 for main engine fire, +10 for leg contact" in Lunar Lander, and "+1 for every timestep the pole remains upright" in CartPole.
- Discount Factor (γ): "A value between 0 and 1 that determines the present value of future rewards." It "balances immediate and future rewards, preventing infinite reward sums in continuous tasks and encouraging agents to prioritize near-term gains without ignoring long-term outcomes." A higher γ values future rewards more, encouraging long-term planning, while a lower γ prioritizes immediate gains, leading to myopic behavior.

The "Markov Property" states that "the future depends only on the current state and action, not on the sequence of events that preceded them."

1.2 Policy vs. Plan

-

Policy (π): "The agent's strategy, mapping states to actions or probability distributions over actions." It "defines an agent's behavior in any state." The goal in an MDP is to find an optimal policy that maximizes the total expected reward over time.

◦

Deterministic Policy: Maps each state directly to a single, specific action. No randomness in action selection.

◦

Stochastic Policy: Maps states to probability distributions over actions.

•

Plan: A sequence of actions derived from a policy to achieve a goal.

1.3 Exploration vs. Exploitation

A key challenge in RL is balancing "Exploration" (trying out new actions to discover more about the environment and potentially better rewards) and "Exploitation" (leveraging current knowledge to choose actions that are known to yield high rewards). Strategies like Epsilon-Greedy and Softmax (Boltzmann selection) are used to manage this trade-off. Epsilon-greedy involves selecting a random action with a small probability (epsilon) and the best-known action otherwise.

2. Value-Based vs. Policy-Based Learning

Reinforcement Learning algorithms can be broadly categorized into Value-Based and Policy-Based methods.

2.1 Value-Based Learning

Value-based methods aim to learn a "value function" that estimates the "goodness" of states or state-action pairs. The agent then derives its policy from these learned values.

•

State-Value Function (Vπ(s)): "Quantifies the expected cumulative discounted return an agent can achieve if it starts in a particular state s and subsequently follows a given policy π indefinitely." It reflects "the long-term desirability or potential of a state under a specific behavioral strategy."

•

Action-Value Function (Qπ(s,a)): "Estimates the expected cumulative discounted return if the agent starts in state s, takes a specific action a, and then adheres to policy π for all subsequent steps." It is "useful for deciding what to do."

2.1.1 Bellman Equations

The Bellman Equation is fundamental to Q-learning and dynamic programming in RL. It recursively defines the value of a state or state-action pair in terms of the immediate reward and the discounted value of future states/actions.

•

Bellman Expectation Equation: For a given policy π, it describes the expected value of a state or state-action pair. For state-value: "$V^\pi(s)=\sum_a\pi(a|s)\sum_{s',r}P(s',r|s,a)[r+\gamma V^\pi(s')]$".

•

Bellman Optimality Equation: Describes the optimal value of a state or state-action pair, assuming the agent always acts to maximize long-term rewards. For optimal state-value: "$V^*(s) = \max_a \sum_{s'} P(s'|s,a) [R(s,a,s') + \gamma V^*(s')]$". For optimal action-value (the basis of Q-learning): "$Q^*(s, a) = \sum_{s'} P(s'|s,a) [R(s,a,s') + \gamma \max_{a'} Q^*(s', a')]$".

2.1.2 Q-Learning

Q-Learning is a model-free, off-policy, value-based algorithm that "learns the optimal Q-function" directly. "The Q-table is the agent's 'memory' or 'knowledge base' about the value of different actions in different states."

- Temporal Difference (TD) Learning: Q-learning uses TD learning, updating its Q-values based on "the 'prediction error' or 'surprise' experienced at time t." The TD error (δ_t) is defined as: "$\delta t=Rt+1+γV(St+1)−V(St)$." This error measures "how much the critic's current value estimate for S_t deviates from a 'better' or more informed estimate, which is the TD target ($R_{t+1} + \gamma V(S_{t+1})$)."
- Update Rule: "$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$". This means Q-values are updated "between the received reward plus the discounted best Q-value of the next state, and the current Q-value." It is an *off-policy* method because it learns the optimal Q-function regardless of the policy being followed (often an epsilon-greedy policy for exploration).
- Experience Replay: A technique where past transitions (state, action, reward, next_state, done) are stored in a buffer and sampled randomly to train the neural network. This helps "to break correlations in the data and improves learning stability."
- Deep Q-Networks (DQN): Extends Q-learning by using neural networks to approximate the Q-function, enabling it to handle environments with continuous or high-dimensional state spaces (like pixel images from Atari games).

2.1.3 SARSA

SARSA (State-Action-Reward-State-Action) is another model-free, on-policy, value-based algorithm. Its update rule is similar to Q-learning but crucially uses the *actual* action taken in the next state, rather than the maximum possible action: "$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \cdot Q(s', a') - Q(s, a)]$". Because it follows the same policy for both behavior and learning, "SARSA tends to learn safer strategies, especially in environments with negative outcomes (e.g., cliffs)."

2.2 Policy-Based Learning

Policy-based methods directly learn a policy function that maps states to actions without explicitly learning a value function.

- Policy Gradient Theorem: Allows for direct optimization of the policy, often using gradient ascent to maximize the expected return. The gradient is "estimated by collecting trajectories (episodes) under the current policy."
- Actor-Critic Methods: Combine elements of both value-based and policy-based approaches.
  - Actor Network: The "policy network" that outputs actions. Its architecture depends on the action space (discrete or continuous). For continuous action spaces, it often outputs parameters for a probability distribution (e.g., mean and standard deviation for a Gaussian policy), usually with a Tanh activation for actions in [-1, 1]. For discrete action spaces, it outputs logits for action probabilities.
  - Critic Network: The "value network" that estimates the state-value V(s) or action-value Q(s,a). It "consistently outputs a single scalar value, representing the estimated state-value V(s)." Its final activation is typically linear.

◦
Advantage (A(s,a)): The core concept linking actor and critic, defined as "$A(s, a) = Q(s, a) - V(s)$". It indicates how much better or worse a taken action was compared to the average expected return from that state. If A(s,a) > 0, the action was better than expected.
◦
Loss Functions: Actor loss typically involves advantage-weighted policy gradient and entropy regularization (to encourage exploration). Critic loss is typically a Mean Squared Error (MSE) comparing predicted values to target values.
◦
Temporal Difference (TD) Error: In actor-critic methods, the TD error "is an estimate of the advantage. It essentially tells you 'how surprised' the critic was by the reward and next state."

2.2.1 A2C (Advantage Actor-Critic)

A2C is a synchronous variant of the Actor-Critic algorithm. It uses a single agent or multiple agents running synchronously to collect experiences and update a single network.

2.2.2 A3C (Asynchronous Advantage Actor-Critic)

A3C is an asynchronous variant that "utilizes multiple workers running in parallel environments to collect experience and update a global neural network." This parallelization speeds up training and decorrelates data. Key aspects include:

•
Global and Local Networks: Each worker has a "local network" initialized with the "global network's state."
•
Push and Pull Mechanism: Workers "push" their computed gradients to the global network and "pull" updated global parameters to their local networks periodically.
•
Shared Optimizer: A crucial component is a "Shared Adam optimizer" which ensures that "the optimizer's internal states (like moving averages of gradients) are also shared across processes, allowing for correct global updates."
•
Hyperparameters: Key hyperparameters include `GAMMA` (discount factor), `LR` (learning rate), `UPDATE_GLOBAL_ITER` (rollout size), `ENTROPY_COEF` (entropy regularization), and `VALUE_LOSS_COEF` (value function loss).

2.2.3 PPO (Proximal Policy Optimization)

PPO is a policy gradient method that "encourages policy exploration by preventing the agent from converging to deterministic (and possibly suboptimal) policies too quickly." It aims to keep policy updates within a "proximal" region to ensure stability. It utilizes Generalized Advantage Estimation (GAE), which "combines n-step returns with exponentially weighted temporal difference errors" to stabilize training by reducing variance.

2.3 Model-Based vs. Model-Free RL

•
Model-Based RL: The agent learns or is given a model of the environment (Transition Model P and Reward Function R). With a model, the agent can "plan" by simulating future states and rewards. Algorithms like Value Iteration and Policy Iteration are model-based.
•

Model-Free RL: The agent learns directly from interactions with the environment without explicitly building a model of its dynamics. Q-learning, SARSA, and Actor-Critic methods (DDPG, TD3, SAC, A2C, A3C, PPO) are model-free.

3. Advanced RL Concepts and Algorithms for Continuous Control

While Q-Learning and SARSA are typically applied to discrete action spaces, more advanced algorithms are needed for continuous action spaces, often employing deep neural networks.

3.1 DDPG (Deep Deterministic Policy Gradient)

DDPG is a model-free, off-policy algorithm designed for environments with continuous action spaces. It uses an Actor-Critic architecture:

- Actor: A "deterministic policy network" that directly outputs a specific continuous action for a given state.
- Critic: A Q-function approximator that estimates the expected return of taking an action in a state.
- Experience Replay: Used to "break temporal correlations and improve learning stability."
- Target Networks: DDPG utilizes target networks for both the actor and critic to stabilize training, slowly updating them towards the online networks.
- Noise: To encourage exploration, noise (e.g., Ornstein-Uhlenbeck noise or Gaussian noise) is added to the actor's output actions during training.

3.2 TD3 (Twin Delayed DDPG)

TD3 is an extension of DDPG designed to address common issues like overestimation bias in Q-values and sensitivity to hyperparameters.

- Twin Critics: Uses "twin Q-networks" (two critic networks) and takes the minimum of their predictions to reduce overestimation bias.
- Delayed Policy Updates: Updates the actor policy and target networks less frequently than the critic networks, allowing the Q-value estimates to stabilize.
- Target Policy Smoothing: Adds noise to the target action when computing the target Q-value, "encourag[ing] smoother value landscapes and more robust policy learning."

3.3 SAC (Soft Actor-Critic)

SAC is an off-policy actor-critic algorithm that aims to maximize a "maximum entropy objective." This means it tries to maximize both the expected return and the entropy of the policy, encouraging more exploration and potentially more robust policies.

4. Gymnasium Environments and Their Characteristics

Gymnasium provides a standardized interface for various RL environments, serving as benchmarks for algorithms.

- CartPole-v1:
  -

Observation Space: 4-dimensional continuous vector (cart position, velocity, pole angle, angular velocity).
◦
Action Space: Discrete (2 actions: push left/right).
◦
Reward: +1 per timestep upright.
◦
Suitability for TD3: "Unsuitable (Discrete Action Space)."
•
LunarLander-v3:
◦
Observation Space: 8-dimensional vector (x, y coordinates, x, y linear velocities, angle, angular velocity, leg contact booleans).
◦
Action Space: Discrete (4 actions: do nothing, fire left engine, fire main engine, fire right engine).
◦
Reward: "~100-140 for landing, +100 for rest, +10 per leg contact. Penalties: -100 for crash, -0.3 for main engine, -0.03 for side engine per frame." Solved at 200 points.
◦
Episode Termination: Lander crashes, moves outside viewport, or comes to rest.
•
BipedalWalker-v3:
◦
Observation Space: 24-dimensional continuous vector (hull angle speed, angular velocity, horizontal speed, vertical speed, joint positions/speeds, 10 lidar readings). "Notably, the state vector does not contain coordinates of the robot, focusing on relative information."
◦
Action Space: 4-dimensional continuous vector, each dimension representing a motor speed value in [-1, 1] for the four leg joints.
◦
Rewards: "300+ points achievable at the far end of the terrain. Falling results in a -100 penalty, and applying motor torque incurs a small cost, encouraging efficient movement."
◦
Episode Termination: Hull contact with ground (falls) or walker exceeds terrain length.
•
CliffWalking-v0:
◦
Observation Space: Discrete(48), representing the agent's position on a 4x12 grid.
◦
Action Space: Discrete(4) (Up, Right, Down, Left).
◦
Start State: (3,0) / State 36.
◦
Goal State: (3,11) / State 47.
◦
Cliff: Locations (3,1) through (3,10). Stepping here incurs -100 reward and resets to start.

◦
Reward Structure: -1 per timestep, -100 for cliff.
◦
Determinism: By default, deterministic, but can be made "slippery."
•
MountainCar-v0 / MountainCarContinuous-v0:
◦
Observation Space: 2 continuous values (car's x-position clipped [-1.2, 0.6] and velocity clipped [-0.07, 0.07]).
◦
Action Space: * `MountainCar-v0`: Discrete (3 actions: accelerate left, right, or no acceleration). * `MountainCarContinuous-v0`: Continuous (value representing directional force).
◦
Starting State: Position random between -0.6 and -0.4, velocity 0.
•
Taxi-v3:
◦
Observation Space: Discrete (500 possible states representing taxi position, passenger location, destination).
◦
Action Space: Discrete (6 actions: move directions, pickup/drop-off).
◦
Reward: -1 per step, +20 for delivery, -10 for illegal actions.
◦
Termination: Successful passenger drop-off or max steps (typically 200).
◦
Characteristics: Deterministic.

5. Neural Network Architectures and Components

Deep Reinforcement Learning algorithms leverage neural networks to approximate complex functions (policies, value functions).
•
Actor-Critic Network Structure (Example from A3C):
◦
A shared initial layer (`self.fc1`) for "feature extraction."
◦
An "Actor head" (`self.actor_head`) for the policy, outputting logits for discrete action probabilities or parameters for continuous action distributions.
◦
A "Critic head" (`self.critic_head`) for the value function, outputting a single scalar value.
◦
Activation Functions: ReLU is commonly used in hidden layers. Actor output activations depend on action space (e.g., Softmax for discrete probabilities, Tanh for continuous actions in [-1, 1]). Critic output is typically linear.
•

Weight Initialization: `nn.init.xavier_uniform_` for weights and `nn.init.constant_` for biases are common.

•

Gradient Clipping: `torch.nn.utils.clip_grad_norm_` is used to prevent "exploding gradients" during training.

•

Replay Buffer (`ReplayBuffer`): A data structure (`collections.deque` is often used) that stores experiences (`s, a, r, s2, done`) and allows for random sampling of mini-batches. This is crucial for off-policy algorithms like DQN, DDPG, and TD3 to "break temporal correlations."

•

Environment Model (`EnvModel`): In model-based RL, a neural network can be trained to predict the `next_state` and `reward` given a `state` and `action`. This model can then be used for planning.

6. Training and Evaluation

•

Hyperparameters: Key parameters like `GAMMA` (discount factor), `LR` (learning rate), `MAX_EPISODES`, `MAX_EP_STEPS`, `ENTROPY_COEF`, `VALUE_LOSS_COEF`, `MAX_GRAD_NORM` are tuned to optimize learning.

•

Episode Termination: Episodes end when a goal is reached, a penalty condition is met (e.g., falling), or a maximum number of steps is exceeded.

•

Reward Shaping: Modifying the reward function to guide the agent towards desired behaviors (e.g., stronger penalties for cliffs to encourage safer paths).

•

Rendering and Video Recording: Environments can be rendered (`env.render()`) for visualization, and training progress can be recorded into videos (`VecVideoRecorder`).

•

Logging and Monitoring: Training statistics (episode reward, loss, value estimates, etc.) are logged to track progress and debug.