SOC Assignment 1

Name of student: Pratyaksh Bhardwaj
Roll number: 23B2401
Project: RL in self driving car
Project ID: 121

Task 1. Theory: Write a short explanation of the mathematical foundations of all 5 algorithms:

- **Q-Learning**
- **Monte Carlo Control**
- **PPO (Proximal Policy Optimization)**
- **DDPG (Deep Deterministic Policy Gradient)**
- **SAC (Soft Actor Critic)**

❖ Q-Learning

Think of it like: A robot learning the best way through a maze by trial and error.

Simple Idea:
Imagine you're in a grid world. Every cell is a "state" and you can move up, down, left, or right (your "actions"). You get a reward when you reach the goal.

Q-Learning learns how good each move is from each place, and writes it in a big table (called the Q-table).

Optimal policy and Q function depends on expected value according to bellman eqn.

Q-Learning is based on the **Bellman equation**:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

- Q(s,a): expected return from state sss taking action aaa

- α: learning rate
- γ: discount factor
- Learns the **optimal action-value function** Q*(s,a) by bootstrapping from future Q-values.
- r: reward

You move right and get a reward of +5. Your old Q-value was 2. If α=0.5 and γ=0.9:

$Q \leftarrow 2+0.5[5+0.9 \cdot \max(Q')-2]$

Example:

You're at cell A. You move right to cell B and get a reward of 10.

The Q-value for (A, right) will get updated to reflect this good move.

Over time, you learn which actions give you the most reward.

How does Q-Learning Works?

The environment provides the agent with a starting state which describes the current situation or condition.

Based on the current state and the agent chooses an action using its policy. This decision is guided by a Q-table which estimates the potential rewards for different state-action pairs.

The agent typically uses an ε-greedy strategy:

It sometimes explores new actions (random choice).
It mostly exploits known good actions (based on current Q-values).

The agent performs the selected action. The environment then provides:

A new state (S') — the result of the action.
A reward (R) — feedback on the action's effectiveness.

The agent updates the Q-table using the new experience:

It adjusts the value for the state-action pair based on the received reward and the new state.
This helps the agent better estimate which actions are more beneficial over time.

With updated Q-values the agent:

Improves its policy to make better future decisions.
Continues this loop — observing states, taking actions, receiving rewards and updating Q-values across many episodes.
Over time the agent learns the optimal policy that consistently yields the highest possible reward in the environment.

❖ Monte Carlo Control

Think of it like: Playing a full game of chess and learning at the end how good each move was.

Simple Idea:
You don't update your knowledge after every move. Instead, you wait until the game finishes, then go back and update how good your actions were based on the whole result.

Example:

You play an episode: Start → A → B → C → Goal

At the end, you got 100 points.

You go back and say, "Hey, the action I took at B helped me get 100, so let me make that move more likely in the future."

Monte Carlo methods estimate the value function based on **complete episodes**:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[G_t - Q(s,a)\right]$$

- Gt=∑γ^k * r: return from time ttt
- No bootstrapping; uses **sampled returns** to update Q-values
- Improves policy via **greedy policy improvement** over Q(s,a)

How does Monte Carlo works ?

The method works by running simulations or episodes where an agent interacts with the environment until it reaches a terminal state. At the end of each episode, the algorithm looks back at the states visited and the rewards received to calculate what's known as the "return" — the cumulative reward starting from a specific state until the end of the episode. Monte Carlo policy evaluation repeatedly simulates episodes, tracking the total rewards that follow each state and then calculating the average. These averages give an estimate of the state value under the policy being followed.

By aggregating the results over many episodes, the method converges to the true value of each state when following the policy. These values are useful because they help us understand which states are more valuable and thus guide the agent toward better decision-making in the future. Over time, as the agent learns the value of different states, it can refine its policy, favouring actions that lead to higher rewards.

❖ PPO (Proximal Policy Optimization)

Think of it like: A personal trainer giving small, safe improvements to your workout rather than radical changes.

Simple Idea:
You're learning a policy (a strategy) for what to do in each state. But if you change the policy too fast, it might break things. So PPO makes small, stable updates.

Clipping?
It's like saying: "Only update your strategy if it doesn't change too much at once."

Example:

Let's say you're a robot choosing how fast to run.

PPO will say: "Let's test a slightly faster speed and keep it if it works well — but don't jump to full speed instantly."

PPO maximizes a **clipped surrogate objective**:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$: probability ratio
- A^t: advantage estimate
- Prevents large policy updates via the **clipping mechanism**

The **clip** keeps the update from changing too much: clip(r,1−ϵ,1+ϵ) ϵ: small constant (e.g., 0.2) to control update size

<u>The main goal of PPO is to find a balance between two things:</u>

Maximizing the objective: This is the core of policy optimization where the agent's policy is adjusted to maximize expected rewards.

Keeping updates small: Big changes can mess up learning so PPO makes sure each update is limited to a safe amount to avoid problems.

The clip function ensures that the ratio between the new and old policy probabilities remains within a certain range (typically [1−ϵ,1+ϵ]) thus preventing large, destabilizing changes. If the policy update leads to a large deviation the clip function effectively limits the contribution of that update to the objective function.

❖ DDPG (Deep Deterministic Policy Gradient)

Think of it like: A GPS that tells you the exact steering angle to drive safely, and learns from feedback.

Simple Idea:
You have a robot arm (or car) that needs to choose precise continuous actions — like moving exactly 0.5 meters. DDPG combines two things:

A critic that judges how good an action is.

An actor that chooses the best action using the critic's advice.

Example:

The critic says, "Turning the steering wheel 15 degrees is better than 20 degrees."

The actor adjusts and learns to pick 15 degrees next time.

Combines **deterministic policy gradient theorem** with deep function approximation:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_a Q(s,a)\big|_{a=\mu(s)} \nabla_\theta \mu_\theta(s)\right]$$

- Uses an **actor** and **critic** (Q-function)
- Learns with **experience replay** and **target networks**
- Optimizes continuous action spaces
- D: experience replay buffer
- Actor improves by following the gradient of the critic's evaluation

DDPG interleaves learning an approximator to Q^*(s,a) with learning an approximator to a^*(s), and it does so in a way which is specifically adapted for environments with continuous action spaces.

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

Because the action space is continuous, the function Q^*(s,a) is presumed to be differentiable with respect to the action argument.

The term

$$r + \gamma(1-d) \max_{a'} Q_\phi(s', a')$$

is called the target, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: \phi. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to \phi, but with a time delay—that is to say, a second network, called the target network, which lags the first.

DDPG Detail: Calculating the Max Over Actions in the Target. Computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a target policy network to compute an action which approximately maximizes Q_{\phi_{\text{targ}}}. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) \right) \right)^2 \right],$$

where \mu_{\theta_}} is the target policy.

Policy learning in DDPG is fairly simple. We want to learn a deterministic policy which gives the action that maximizes Q_{\phi}(s,a). Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) .

- **SAC (Soft Actor-Critic)**
  Think of it like: A curious kid trying everything before settling on the best way.

  Simple Idea:
  SAC says: “I want to get good rewards, but also keep exploring different actions.”

It adds an entropy bonus — which means it likes uncertainty (trying new things). That's helpful in complex environments.

Example:

You're playing a new video game.

SAC encourages you to try all weapons and paths, not just the one that gave you a reward once.

Over time, you learn a balanced and smart strategy, not a greedy one.

Maximum entropy RL

Maximizes **expected reward plus entropy** for exploration:

$$J(\pi) = \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right]$$

- Entropy term: $\mathcal{H}(\pi(\cdot|s)) = -\mathbb{E}_{a\sim\pi}[\log \pi(a|s)]$
- Balances exploitation and exploration using a temperature α\alphaα
- Learns a **stochastic policy** in continuous spaces

Entropy-Regularized Reinforcement Learning

Entropy is a quantity which, roughly speaking, says how random a random variable is. If a coin is weighted so that it almost always comes up heads, it has low entropy; if it's evenly weighted and has a half chance of either outcome, it has high entropy.

In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep.

SAC sets up the MSBE loss for each Q-function.SAC uses the clipped double-Q trick, and takes the minimum Q-value between the two Q approximators.

The way we optimize the policy makes use of the reparameterization trick, in which a sample is drawn by computing a deterministic function of state, policy parameters, and independent noise. This policy has two key differences from the policies we use in the other policy optimization algorithms:

1. The squashing function. The \tanh in the SAC policy ensures that actions are bounded to a finite range.
2. The way standard deviations are parameterized. In VPG, TRPO, and PPO, we represent the log std devs with state-independent parameter vectors. In SAC, we represent the log std devs as outputs from the neural network, meaning that they depend on state in a complex way.

SAC uses:

- **Two Q-networks (clipped double Q-learning)**
- **Reparameterization trick** for backprop through stochastic sampling
- **Squashed Gaussian policy**

Algorithm	Policy Type	Explanation
Q-Learning	Off-policy	Learns optimal policy using a different behavior policy (e.g., ε-greedy), relying on max Q-values.
Monte Carlo	On-policy	Typically on-policy, learning from episodes generated by the current policy; off-policy versions exist with importance sampling.
PPO	On-policy	Uses data from the current policy to update itself, with clipping to ensure stable learning.
DDPG	Off-policy	Uses a replay buffer and target networks, allowing learning from old data not generated by the current policy.
SAC	Off-policy	Learns from stored past experiences via replay buffer and trains a stochastic policy, promoting exploration.