Intelligent Control:

*Implementing Soft Actor-Critic (SAC) from Scratch*

*Description:*

*Implement the SAC algorithm from scratch using PyTorch or JAX and apply it to robotics tasks. Analyze the impact of entropy regularization, target networks, and Q function stability.*

ΜΑΡΚΑΝΤΩΝΑΤΟΥ ΕΙΡΗΝΗ | up1095489

ΣΟΛΩΜΟΝΙΔΗ ΕΛΕΥΘΕΡΙΑ | up1092837

**ΕΤΟΣ:** 4ο

ΕΥΦΥΗΣ ΕΛΕΓΧΟΣ| 23/6/2025

# **Introduction**

This project involves implementing the Soft Actor-Critic (SAC) algorithm from scratch in PyTorch and applying it to a maze navigation task using the PointMaze environment. SAC is a popular off-policy Reinforcement Learning (RL) algorithm designed for continuous action spaces. The goal of this project is to develop a working SAC agent capable of learning to reach a moving target in a 2D maze with sparse rewards. Key components of the implementation include twin Q-networks, soft target updates, and entropy regularization to maintain a balance between exploration and exploitation. Special focus is given to analyzing the effect of entropy scaling (α), target entropy tuning, and critic stability during training.

# **Theoretical foundations of SAC**

2.1 Limitations of Standard Actor-Critic Methods

The Actor-Critic algorithm is a type of RL method that combines two key components: the **Actor**, which selects actions, and the **Critic**, which evaluates those actions. While this approach performs well in many environments, it can struggle in scenarios with sparse rewards, such as ours. In such cases, the agent may prematurely converge to a suboptimal policy due to insufficient exploration of alternative strategies. This limited exploration increases the risk of getting stuck in local optima and failing to discover more effective solutions.

Another significant drawback of Actor-Critic methods is their tendency to generalize poorly. The actor often learns a single strategy that performs well during training but lacks robustness when faced with new or slightly different situations.

These limitations highlight the value of more advanced approaches like **SAC**, which improve exploration by introducing entropy and randomness into the learning process. SAC helps address the issues of local optima and poor generalization, making it a more reliable choice in complex or unpredictable environments.

2.2 Resolving Issues with SAC

Instead of only asking the Actor to maximize his reward when training, the Critic will ask the Actor to do so while acting as randomly as possible. That is maximizing the entropy of its current policy. “Soft” refers to “entropy regularization”. The entropy of the policy being maximized, the actor is incentivized to explore alternative strategies and to recover from unexpected situations. Consequently, it introduces a new value function, the soft action – value function:

(1)

This includes the entropy term:

(2)

The **log function** is able to measure uncertainty or randomness in a probability distribution. The term is the probability that the policy chooses action in .

* When is close to 1 is near 0 (low surprise)
* When is close to 0 is a large negative number (high surprise)
* So is small for confident choices, large for unlikely ones.

Εικόνα που περιέχει κείμενο, γραμμή, γράφημα, διάγραμμα

Το περιεχόμενο που δημιουργείται από AI ενδέχεται να είναι εσφαλμένο.

Figure 1: Log and negative Log on action probabilities in SAC

Figure 1 illustrates how SAC penalizes overconfident decisions to encourage exploration.

SAC builds on this idea using a **policy iteration framework**, alternating between **policy evaluation** (updating the Q-functions) and **policy improvement** (updating the policy via a stochastic off-policy gradient step). This structure not only helps balance exploitation and exploration but also enables efficient learning from past experience using a replay buffer.

2.3 Policy Evaluation

Policy Evaluation is the initial step in RL, involving the determination of the value function for a given policy. In standard RL, this is done through the Bellman Optimality Equation using only rewards and assuming a greedy policy:

*Bellman Optimality Equation*:

, (3)

where:

(4)

But in SAC, we defined a different value function in Equation (1), which includes an entropy term to encourage exploration. This leads to a modified Bellman update known as the **Soft Bellman Equation:**

*Soft Bellman Equation:*

(5)

Instead of picking just the **best** action, it considers **all possible actions** based on their likelihood under the current policy. I want to find the Bellman operator that will allow me to optimize my Q- values.

2.4 Policy Improvement

The goal is to find the optimal policy. Policy Evaluation is the initial step in reinforcement learning, involving the determination of the value function for a given policy. Once the value function is evaluated, Policy Improvement follows.

In most RL algorithms, the optimal policy is found by:

(6)

The maximum entropy objective generalizes the standard objective shown in Equation (6) by augmenting it with an entropy term, such that the optimal policy additionally aims to maximize its entropy at each visited state:

*,* (7)

where α is the temperature term that determines the relative importance of the entropy term versus the reward, and thus controls the stochasticity of the optimal policy.

Specifically:

* A higher α encourages more exploration, since the policy is rewarded for being more random.
* A lower α shifts the focus toward exploitation, where the policy sticks to high-reward actions.

2.5 Iterative Soft Q- Value Optimization and Policy Update

To optimize action-value estimates, the Bellman operator is employed, which iteratively updates the soft Q-values for a fixed policy. Starting from any function and repeatedly applying Equation (5).

In the policy improvement step, we update the policy towards the exponential of the new soft Q function.

, (8)

where π′ denotes the current policy before the update. The Kullback –Leibler Divergence tells us how much information is lost when you approximate one distribution with an another one estimated. In our case, it makes our policy look like a normalized exponential of Q- values without moving too far from our current policy. We make actions that have higher soft Q-values more probable (softmax like update).

The partition function Z normalizes the distribution, and while it is intractable in general, it does not contribute to the gradient with respect to the new policy and can thus be ignored.

2.6 Gaussians and the Reparameterization Trick

In RL environments, actions can typically be either discrete or continuous. For discrete action spaces, action probabilities can be modeled directly using techniques like the softmax function. In contrast, continuous action spaces require modeling a probability density function, such as a Gaussian distribution.

A Gaussian Distribution is a common way to model continuous stochastic policies:

, (9)

where s is the current state.

Why are Gaussians tractable?

If you define a Gaussian as shown in Equation 9 then sampling an action is:

*(Reparameterisation Trick)* (10)

Since in practice we prefer policies that are tractable we will restrict the policy in Equation (8) to Gaussians. In the SAC, Gaussian distributions are used to represent stochastic policies in continuous action spaces.

# Implementation

3.1 Network Architecture

We will consider a parameterized soft Q-function, and a tractable policy). The Q neural network is trained to approximate the **soft Bellman operator** while the policy network can be modeled as a Gaussian with mean and covariance given by neural networks:

(11)

(12)

Here, μ(s) represents the most likely action in a given state, and captures the uncertainty or stochasticity around that action. We are expecting to get smaller as time progresses and the policy becomes more deterministic.

Why output instead of ?

* σ must be positive (it is a standard deviation)
* If you output it directly, your network might output negative or zero values, which would break the Gaussian:

(13)

Τhe third neural network will automate tuning of temperature coefficient α.

The idea behind the third neural network is that we are trying to maximize rewards but not let the policy become too deterministic. So instead of forcing the policy to have entropy H, we do this:

* Let the policy do whatever it wants
* But penalize it when it becomes too confident (when entropy gets too low)
* The strength of the penalty is controlled by a variable

3 objective functions:

The parameters of the above networks are and .

The soft Q- function parameters can be trained to minimize the soft Bellman residual

where D is a replay pool with experiences and the θ’ refers to the target network

The policy parameters can be directly learned by minimizing the expected KL- divergence:

The final objective function:

# **Environment Setup and Agent Dynamics**

4.1 PointMaze Simulation Setup

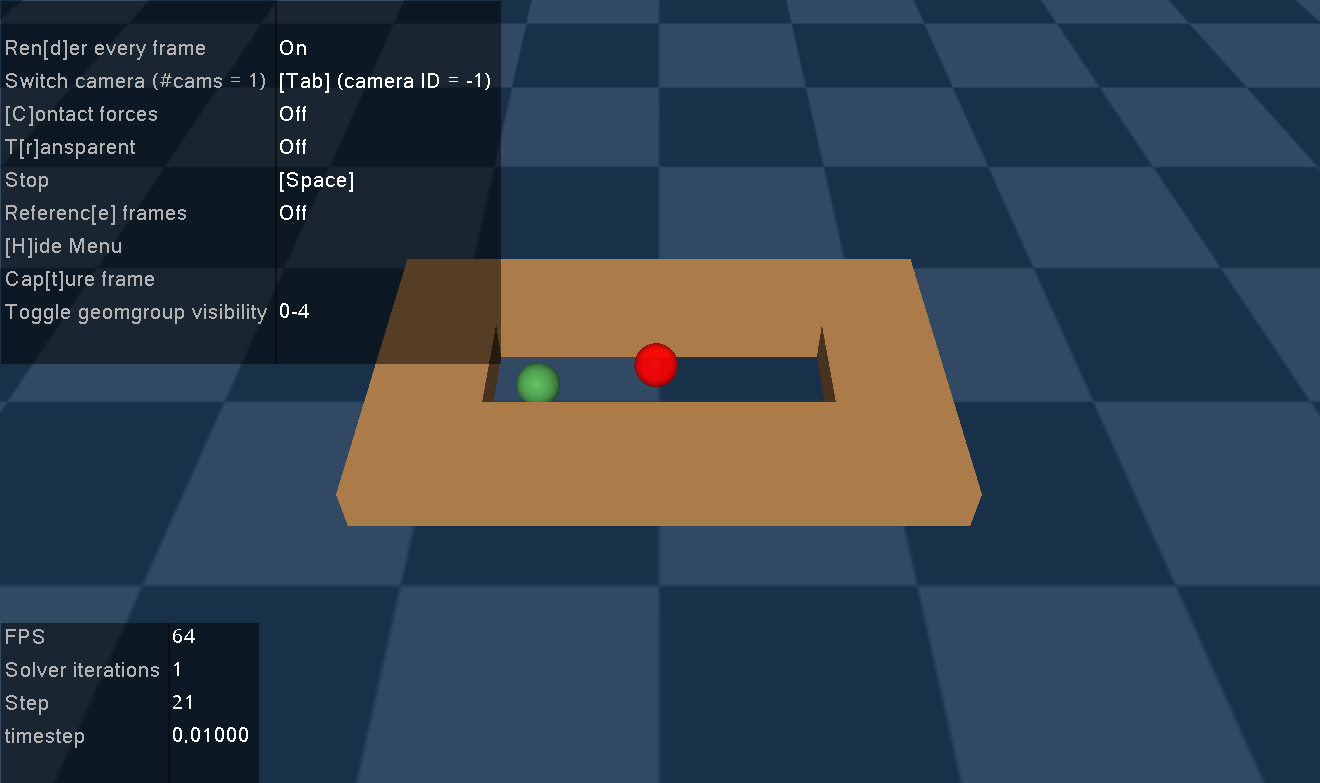
**

Figure 2: PointMaze-UMaze-v3 simulation environment

In this project, the **Gymnasium** library was used to provide simulation environments for reinforcement learning experiments. Specifically, the PointMaze environment was selected, simulating a maze where an agent must navigate from a randomly initialized starting position to a predetermined goal. The chosen environment is a two-dimensional horizontal maze (Figure 2), where the agent is modeled as a point mass that can move freely along the x and y axes. At each time step, the agent observes its current **state** and selects an **action**, which influences its **next state** and the **reward** it will receive. These concepts are described below.

4.1.1 State

The agent's state includes its current position (achieved\_goal) and the position of the desired goal (desired\_goal). The agent needs to know the desired goal at every step (a fixed and specific vector for each episode) because this determines where it should move and what it is trying to achieve. Especially in environments where the goal changes each episode, incorporating the desired goal is necessary for learning a policy that can generalize to different targets.

4.1.2 Action

The agent’s action represents the linear force applied to the point mass in the x and y directions, moving it to the next state. At each time step, the agent selects an action corresponding to a continuous change in its position within the maze. Once an action is selected, the agent transitions to the next state, receives a reward, and a signal indicating whether the episode is done.

4.1.3 Reward

The reward function used in this environment is based on **sparse rewards**. Specifically, the agent receives a reward if and only if it reaches the goal (the ball is considered to have reached the goal when the Euclidean distance is less than 0.5 m), while all intermediate steps yield zero reward. This makes the learning process more challenging, as the agent does not receive direct feedback on whether it is moving in the right direction or not. Learning is primarily based on a few but crucial experiences where the agent achieves the goal. Despite the use of sparse rewards, the SAC algorithm manages to learn effectively, mainly thanks to the use of **replay buffer**.

4.1.4 Replay Buffer

The replay buffer stores all the agent’s experiences (state, action, reward, next state, done)at each step of the episode, allowing successful experiences to be reused. During training, the agent samples randomly from the buffer, selecting mini-batches of experiences according to a predefined batch size. This way, the information about "success" is not lost but can reappear multiple times, gradually strengthening the agent’s policy. When an experience includes reaching the goal (i.e., a high reward), the Q-network is updated.

Specifically, the Q-value increases because the target equation:

includes a high reward r. The Q-network (the Critic) gradually learns that certain state-action pairs lead to high rewards. At the same time, the policy network (the Actor) is trained to select actions that maximize the Q-value—those most likely to result in successful transitions. As a result, the probability that the agent will reproduce success increases, even in environments with sparse rewards.

4.1.5 Neural Networks

This process is supported by the two neural networks mentioned earlier: the Actor network and the Critic network. The neural network for the Critic tries to estimate the Q-values, that is, to calculate the expected returns after selecting a specific action from a given state following the policy. Two Q-functions are used to prevent overestimation of the Q-value, meaning to avoid the network mistakenly thinking an action is much better than it really is, which would cause the Actor to prefer it without justification. The critic\_loss measures how well the Critic learns to predict how good or bad a chosen action is; in other words, it evaluates the actions. However, the selection of actions is based on the Actor’s neural network.

The Actor essentially tries to learn a stochastic policy that maximizes expected returns while maintaining high entropy (H), aiming to select good actions. A policy is stochastic when the actions taken at each step are sampled from a probability distribution so that the same action is not always chosen for a given state. Stochasticity thus introduces randomness in action selection. The actor\_loss indicates how well the agent’s policy is being trained. When the actor\_loss is high, it means the Actor proposes actions that, according to the Critic, do not lead to high rewards, signaling that further training is needed to improve the policy.

Having analyzed the roles of the actor and critic networks, we now turn to the target Q-networks, a crucial component introduced to enhance training stability and reduce value estimation bias in the Soft Actor-Critic algorithm.

4.1.6 Target Q-Networks

In SAC, target Q-networks are delayed copies of the main Q-networks used to compute the target Q-values for the next state during training. This means that the target networks are updated only occasionally and are fixed values for some time.

The target for updating the Q-function is based on Equation (5). However, if the same Q-network is used for both:

* Predicting the current Q-value and
* Estimating the next-state Q-value as the target,

then the network ends up learning from its own unstable predictions.

This feedback loop can lead to:

* Overestimation bias: The Q-network tends to overestimate values due to the use of max over noisy predictions.
* Oscillations and divergence: Since the targets keep changing as the network trains, updates become unstable and learning may diverge.

To solve this, target Q-networks are introduced. Target Q-networks are delayed copies of the main Q-networks used to compute the target Q-values for the next state during training. This means that the target networks are updated only occasionally and are fixed values for some time. They are initialized with the same weights as the original Q-network:

In SAC, two such target Q-networks are used (to support the *Double Q-learning* technique), and the minimum of their predictions is used when computing the backup target. This helps further reduce overestimation bias.

4.1.7 Entropy Regularisation

*MORE*

As mentioned earlier in section 2 , besides maximizing the expected reward, the SAC algorithm incorporates another important objective: maintaining high entropy in the policy. **Entropy regularization** is a technique that improves **exploration** by encouraging the policy to maintain a balanced distribution over actions, preventing it from becoming too deterministic too quickly. Without considering entropy regularization, the policy may quickly converge to a small set of actions that yield high rewards but potentially ignore better alternative actions. By introducing the entropy term, we define how stochastic, or how "random," we want the agent’s policy to be when selecting actions. This way, the agent learns to explore the environment by trying different actions instead of getting stuck repeating the same ones.

Specifically, in this work, the entropy is dynamically adjusted during training through the temperature parameter (denoted by α), which controls how much weight entropy has in the training process. If the policy becomes too "confident" (low entropy), α increases to promote more randomness. Conversely, with high entropy, α decreases to make the system more deterministic.

**5. Agent Training**

Everything analyzed so far refers to the data used to train the agent. Training is divided into episodes, and at the beginning of each episode, the environment is reset to an initial state. Each episode ends after a maximum number of steps. In the first few episodes (the warm-up phase), actions at each step are chosen randomly to allow the agent to explore the environment and fill the replay buffer with some initial experiences. During this phase, the Actor neural network is being trained but is not yet used to select actions.

When training a neural network, we don’t update the weights for just one sample at a time. Instead we use a batch of data. For each state in the batch:

* The actor picks an action
* The critic evaluates it
* We get one q value per state

After enough data has been collected from these episodes, the actual training begins. From this point on, at each step, actions are selected by the Actor network. More specifically, the agent samples past experiences by taking random batches from the buffer and performs updates on the Actor and Critic neural networks.

5.1 Critic Update

In the Soft Actor-Critic (SAC) algorithm, two Q-networks and two corresponding target Q-networks are maintained. To train the Critic, the network learns to estimate how good an action is in a given state by computing Q-values. These are compared to the target Q-values computed using the Bellman equation (I), based on the target networks. The objective is to minimize the loss between the prediction and the target. The process works as follows:

From a stored transition (s, a, r, s′) collected by the agent:

The target Q-value is computed as an estimate for the (s, a) pair:

**(3.2)**

This Q\_target(s, a) is the value we want the Critic to learn.

Then, the loss of the Q-networks is computed (let y = Q\_target(s, a)):

**(3.3)**

Each Q-network tries to bring its output as close as possible to *y*. Then, backpropagation is performed and the weights of the Q-networks are updated.

5.2 Actor Update

To train the Actor network, once an action is selected for a given state s, both Q1 and Q2 values are computed for that action, and the smaller of the two is used. This serves as a conservative estimate to prevent overestimating the action value.

Then the actor loss is computed using the equation:

**(3.4)**

The goal is to **minimize** this term.

The actor is trying to choose actions that:

* Yield a high Q(s, a) value (i.e., good rewards)
* Have high entropy (low confidence), encouraging exploration

Back propagation is then performed and the weights of the policy network are updated.

5.3 Temperature Update

In addition to updating these two neural networks, the entropy coefficient α is also updated automatically to control how random or confident the policy should be.

For this reason, the temperature loss is computed:

**(3.5)**

This loss measures how far the current entropy is from the target.

We perform gradient descent on this loss to adjust α:

* If the policy is too random → decrease α to make it more focused
* If the policy is too predictable → increase α to add more randomness

5.4 Soft Q-Network Update Target

The Target Q-Networks are soft-updated at each step (Polyak averaging) to ensure they slowly track the values of the current Q-networks. This helps stabilize training and avoid oscillations. The update rule is:

**(3.6)**

* θ are the weights of the current Q-network
* θ\_target are the weights of the target Q-network
* **τ** is a small constant (typically around 0.005) that determines how slowly the target network is updated

This soft update ensures the target network evolves gradually, preventing rapid changes that could destabilize learning