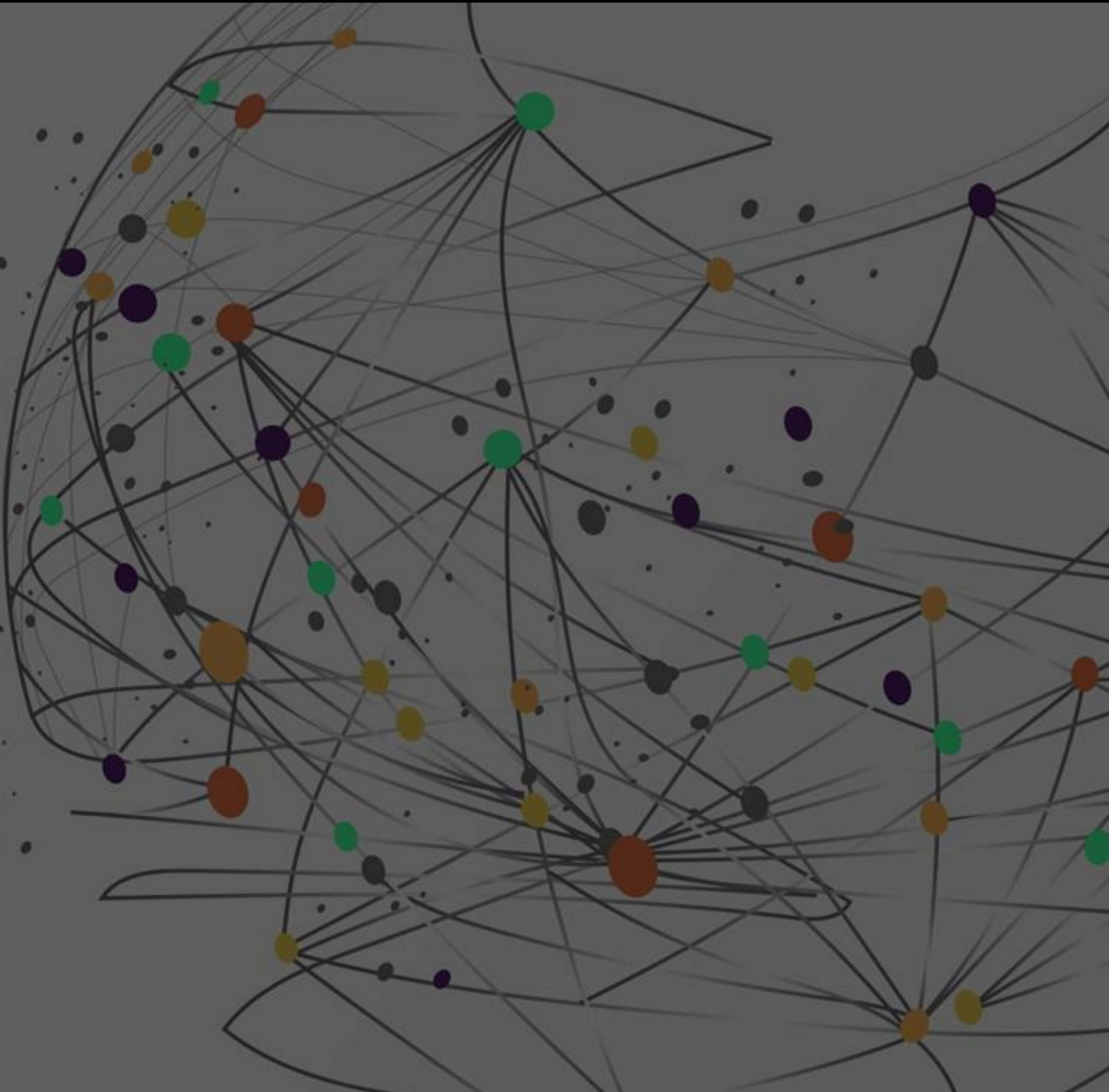
SAC ONLINE vs OFFLINE LEARNING

AN INVESTIGATION INTO SAMPLE EFFICIENCY



VISWAK RB – 124104338

M.SC. DATA SCIENCE & ANALYTICS - UCC

Reinforcement Learning (RL)

Reinforcement Learning (RL) is a way to train an agent to make decisions by trial and error. The agent interacts with an environment by:

- Observing a **state**
- Taking an **action**
- Receiving a **reward**
- Moving to a **new state**

Over time, the agent learns a policy that maximizes total reward.

Two main training styles:

- **Online RL**: The agent learns by actively interacting with the environment.
- **Offline RL**: The agent is trained on a pre-recorded dataset — no new environment interaction is allowed.

SAC (Soft Actor-Critic)

- SAC is a modern RL algorithm designed for **continuous action spaces**.
- It uses both **actor–critic architecture** and **entropy regularization**
- The actor learns a **stochastic policy** (samples actions from a Gaussian)
- Two critics estimate how good actions are (Q-values)
- The entropy term in the objective encourages exploration and avoids premature convergence

Why SAC?

- Stable
- Sample-efficient
- Works well even in complex, continuous environments

SAC vs DQN

Feature	DQN (Deep Q-Network)	SAC (Soft Actor-Critic)
Action type	Discrete	Continuous
Policy	Deterministic (pick best Q-value)	Stochastic (sample from a learned dist.)
Exploration	ε-greedy	Built-in via entropy
Training style	Off-policy	Off-policy
Target networks	One Q-network	Two Q-networks + soft updates

Key difference:

- DQN works well for simple, discrete problems like Atari games.
- SAC is designed for more complex tasks like robotic control, where actions are continuous and noisy.

Task Summary

We wanted to explore how SAC performs when trained:

1. Online — the agent interacts with the environment while learning
2. Offline — the agent is trained only on a fixed dataset collected earlier

We ran this experiment on two environments:

- LunarLanderContinuous-v2 : A simulated lunar module must land softly on a designated pad using thrusters.
  Action space: 2 continuous controls (main engine + side thrusters)
  Reward depends on landing speed, position, angle, and fuel efficiency.
- Pendulum-v1 : A simple inverted pendulum must be balanced upright by applying torque.
  Action space: 1 continuous torque value
  Reward penalizes angle deviation and high velocity.

For each:

- We trained online for 1,000 episodes
- Collected the replay buffer
- Used it to train a new agent offline
- Compared convergence and final performance

Code Structure (.py Files)

sac_torch.py

The heart of the SAC agent. Defines:

- Actor and critic networks
- Replay buffer
- Training loop (learn())

networks.py

Contains the neural network models for actor and critic.

utils.py

Simple utility for plotting learning curves

main_sac.py

- Handles online training:
- Interacts with the environment
- Stores data
- Trains and saves models and dataset

main_sac_offline.py

- Handles offline training:
- Loads dataset
- Trains without any new interaction.

Online and Offline Code Flow

Online Training Flow (main_sac.py)

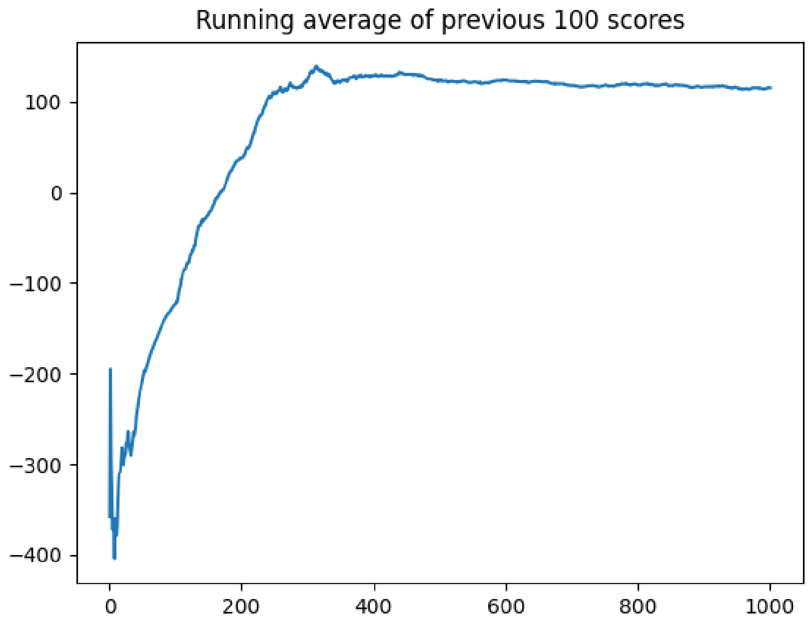
1. Agent runs episodes in the environment
2. After every step:
    1. Saves transition to replay buffer
    2. Learns from sampled batches
3. All transitions are saved as a .pkl file for offline use
4. Plots learning curve during training

Offline Training Flow (main_sac_offline.py)

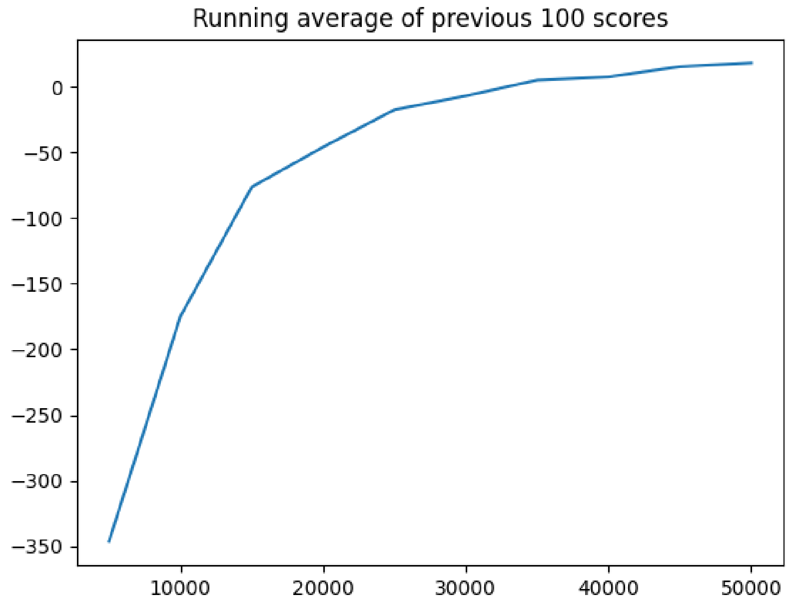
1. Loads the .pkl dataset into replay buffer
2. Trains the agent entirely from this fixed data (no new steps in the env)
3. Every few thousand steps:
    1. Evaluates policy in the environment
    2. Plots performance

By reusing the same agent class across both flows, we minimized code duplication while testing both training setups.

Results Overview - LunarLanderContinuousv2

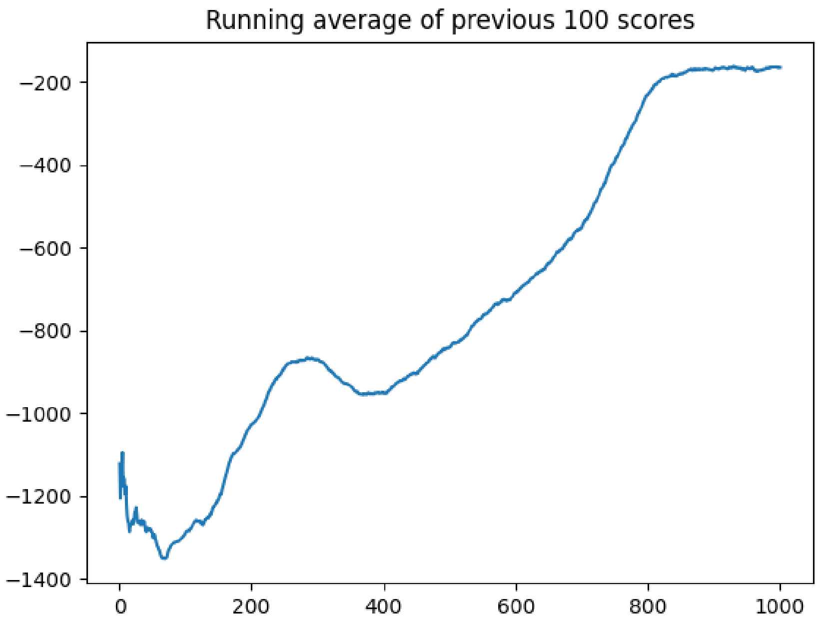


ONLINE

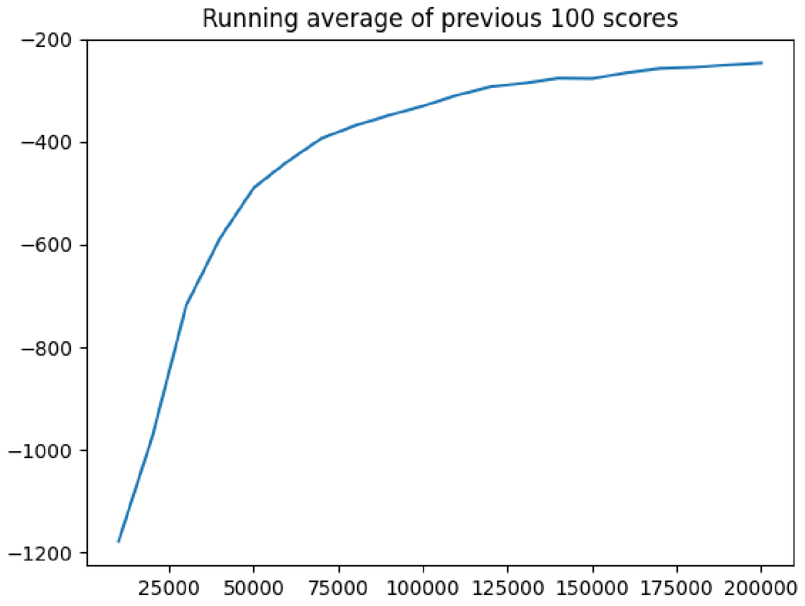


OFFLINE

Results Overview - Pendulum V1



ONLINE



OFFLINE

Key Challenges

1. NaNs During Training (Pendulum)

- When we ran SAC on Pendulum, training exploded — the policy outputs and losses became NaN early on.
  This happened because Pendulum gives small, consistent negative rewards (–16 to 0), making it easy for unstable gradients to blow up.

Fixed:

- Added a **random warm-up phase** (5,000–10,000 steps) before learning started
- **Clamped** log standard deviation (log σ) to stay in a safe range [–20, +2]
- Limited σ values themselves to avoid zero or extreme variance
- Added **gradient clipping** to stop huge updates from destabilizing learning
- These tweaks made the training smooth and reproducible.

Key Challenges (Cont.)

2. Gym API Changes

- The latest version of Gym changed how env.reset() works — it started returning a tuple (obs, info) instead of just obs. If not handled, this broke the input to the agent and caused crashes or silent bugs.
- We unpacked the tuple properly before passing observations to the network.

3. Missing Folders During Model Saving

- Our model checkpoints were failing to save because folders like models/ or tmp/ didn't exist.
- We fixed this by calling os.makedirs(..., exist_ok=True) before each torch.save().

Interpretation

- SAC performs well online: both tasks reached strong rewards after 1000 episodes.
- In the offline setting, the performance dropped — especially on Pendulum.
- This is expected: SAC relies on fresh samples to stay stable.
- Pendulum required more tricks to train properly (due to sparse reward scale)
- The **gap** between online and offline SAC suggest that:
  Offline SAC doesn't handle out-of-distribution actions well.
- Next Steps ? – Hybrid (**combine offline pretraining with online fine-tuning?). -** Exploring offline-specific methods like CQL or AWAC?

References

- https://github.com/rail-berkeley/d4rl
- https://huggingface.co/blog/offline-rl
- https://github.com/vwxyzjn/cleanrl/blob/master/cleanrl/sac_continuous_action.py
- https://spinningup.openai.com/en/latest/algorithms/sac.html