Reinforcement Learning

Per l’Ottimizzazione delle Prestazioni Energetiche negli Smart Building

Relatore Prof. Meli Daniele
Correlatore Prof. Farinelli Alessandro

Candidato Jacopo Parretti, matricola VR473936
Laurea Triennale in Bioinformatica, Università degli Studi di Verona

UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

A comprehensive framework for evaluation and comparing state-of-the-art Reinforcement Learning algorithms on urban energy optimization tasks using the CityLearn environment.

About the Project

CityLearn is an **opensource** OpenAI gym environment for research in demand response and smart grid energy management.

It simulates building energy dynamics, enabling the **development** and **testing** of control strategies.

By comparing modern RL algorithms (*SAC, PPO, TD3*), the focus is to highlight their effectiveness for efficiently managing building energy resources-*cutting costs, reducing emissions, enhancing grid stability*-especially in dynamic environment where traditional methods may be insufficient.

Objectives

- **Evaluate RL algorithms (SAC, PPO, TD3)** for urban energy optimization using the CityLearn environment

- **Reduce energy consumption, costs, and emissions** through smart building control strategies

- **Develop an extensible and replicable framework** to support research, experimentation, and education in energy-aware reinforcement learning

My Contributions

- **Designed and implemented CityLearnRL**, a modular framework for benchmarking RL algorithms in realistic multi-building energy environments
- **Customized the simulation environment** with a Gym wrapper exposing detailed state/action spaces and discrete control over six buildings
- **Engineered multi-objective reward functions**, balancing comfort, emissions, grid performance, and resilience with configurable weights
- **Built a full training and evaluation pipeline** using SB3: vectorized training, custom logging (TensorBoard), and animated result analysis
- **Performed algorithms comparing** evaluating reward trajectories, convergence behaviors, and KPI trade-offs

Soft Actor-Critic (SAC)

Type: off-policy actor-critic; learns both from a policy and a value function at the same time

*Soft: it adds an **entropy bonus;** it rewards the policy for staying uncertain or explanatory, so it doesn't get stuck doing the same thing*

How does it work?

1. **Collect experiences** (state -> action -> reward -> next state) in a replay buffer
2. **Critic update**: learn by minimizing a Bellman error that includes an entropy term
3. **Actor update**: adjust the policy to both **maximise** Q(s, a) and keep entropy high

Key benefits:

- Learns efficiently from past data (off-policy)
- Naturally balances exploration vs exploitation
- Strong, stable performance on continuous tasks

Proximal Policy Optimization (PPO)

Type: an **on-policy** method that limits how much the policy can change at each step

Proximal: to avoid big leaps in policy space that could collapse performance

Core idea - “clipped" update: compute the probability ration and clip it into [1 - ϵ, 1 + ϵ] so that the update stays within a safe range

How does it work?

1. **Run the current policy** in the env, collecting a batch of trajectories
2. **Compute advantages** (how much better an action was than expected)
3. **Optimize** the policy loss with multiple small gradient steps, using the clipped ration to prevent over-updating

Key benefits:

- Easy to implement and tune
- Stable training
- Works well for both discrete and continuous controls

Twin Delayed DDPG (TD3)

Type: an improved version of DDPG, an **off-policy** actor-critic for deterministic control

*“Twin trick”: use **two** critic networks Q1 Q2; take the **minimum** of their estimates to reduce over-optimistic value estimates*

Delayed updates: only updates the actor and the target networks every few critic updates - this gives the critics time to stabilize first

Target policy smoothing: when computing the target Q-value, add small clipped **noise** to the action. This stops the critic from learning to exploit narrow peaks in Q.

How does it work?

1. **Store** transitions in replay buffer
2. **Critic step:** update both critic toward a common target that uses the noisy, delayed actor
3. **Actor step** (less often): maximize Q using gradients from the first critic.

Key Benefits:

- Reduces overestimation bias
- More stable and reliable than vanilla DDPG
- Excellent for high-dimensional, continuous action problems

Reward Design

This project uses the CityLearn Challenge reward function, which is **multi-objective** and designed to **minimize a normalized cost**:

$$\text{Reward} = -C_{\text{RL}}$$

Where CRL < 1 indicates performance **better than** the Rule-Based-Controller.

The **Cost CRL** is the average of **five key metrics**, each with equal weight:

1. **Peak electricity demand**
2. **Average daily peak reward**
3. **Ramping** (variation between timesteps)
4. **1-Load Factor** (inefficiency in usage)
5. **Net electricity consumption**

Each metric is **normalized** with respect to the corresponding RBC baseline value.

$$C_{\text{RL}} = \frac{1}{5} \sum_{i=1}^{5} \frac{\text{metric}_i}{\text{baseline}_i} \quad \Rightarrow \quad \text{Reward} = -C_{\text{RL}}$$

Reward Design

Parameters used in code:

Parameter	Value/Description
weights[l] = 1/5	Each metric contributes equally to the total cost
Baseline	Rule-Based-Controller RBC metrics
Normalization	Each metric divided by its RBC equivalent
Final Reward	Negative of the normalized average cost

Hands on experiments

Environment Setup & Dependency Alignment

- Python 3.8.20 in a dedicated conda environment
- Install with pip install -r requirements.txt in repo root

Core Libraries & Versions:

```
setuptools>=65.5.0 # For building and distributing Python packages
wheel<0.40.0 # For faster installation of Python packages
gym==0.21.0 # Toolkit for reinforcement learning
stable-baselines3==1.8.0 # Implementations of reinforcement learning algorithms
CityLearn==2.1.2 # OpenAI Gym environment for building energy coordination
numpy>=1.24.0 # For scientific computing and multi-dimensional arrays
torch>=2.0.0 # Machine learning framework
cloudpickle>=2.0.0 # For serializing Python objects
matplotlib>=3.7.0 # For creating visualizations
pandas>=1.5.0 # For data analysis and manipulation
```

feat: compatibility guide

The CityLearn tutorial.ipynb and examples were originally written against older *Gym/SB3* releases. In particular, **CityLearn v2.0**+ targeted *Python 3.7+, Gym 0.21.x and SB3 1.x*

Later CityLearn versions switched to the Gymnasium API, but **the tutorial assumes the pre-Gymnasium Gym interface.**

- **Python 3.8.20** – Install Python 3.8 (specifically 3.8.20 if possible) to meet CityLearn's requirements (≥3.7) and SB3's (1.x supports 3.7+).
- **Gym 0.21.0** – Pin Gym to exactly version 0.21.0. This restores the legacy Gym API.
- **CityLearn** – Use a CityLearn release that supports Gym 0.21. CityLearn 2.2.0 (Nov 2024) is appropriate (it requires gym>=0.21.0). Later versions (2.3.x) assume Gymnasium.
- **Stable-Baselines3** – Install SB3 1.8.0 (or any 1.x ≥1.5). This version supports Gym 0.21 as its minimum dependency.

feat: compatibility guide

Code Changes

Gym 0.21 differs from newer Gym/Gymnasium in the reset()/step() signatures

env.reset()	In Gym 0.21 this returns only initial observation. Gymnasium's env.reset() returns (obs, info)
env.step()	In Gym 0.21 this returns (obs, reward, done, info)

```
# Gymnasium style (newer): returns (obs, info)
observations, _ = env.reset()
# Gym 0.21 style: returns obs
observations = env.reset()
```

```
observations, _, _, _ = env.step(actions)
```

Main tutorial extension highlights

Soft Actor-Critic (SAC)

🔧 Implementation Overview

- **Algorithm**: Soft Actor-Critic (SAC)
- **Environment**: CityLearn with custom reward function
- **Training Episodes**: 1000 (optimized for convergence)
- **Seeds**: 5 different random seeds for robust evaluation
- **Wrappers**:
  - `NormalizedObservationWrapper`: Normalizes observations for stable training
  - `StableBaselines3Wrapper`: Ensures compatibility with SB3

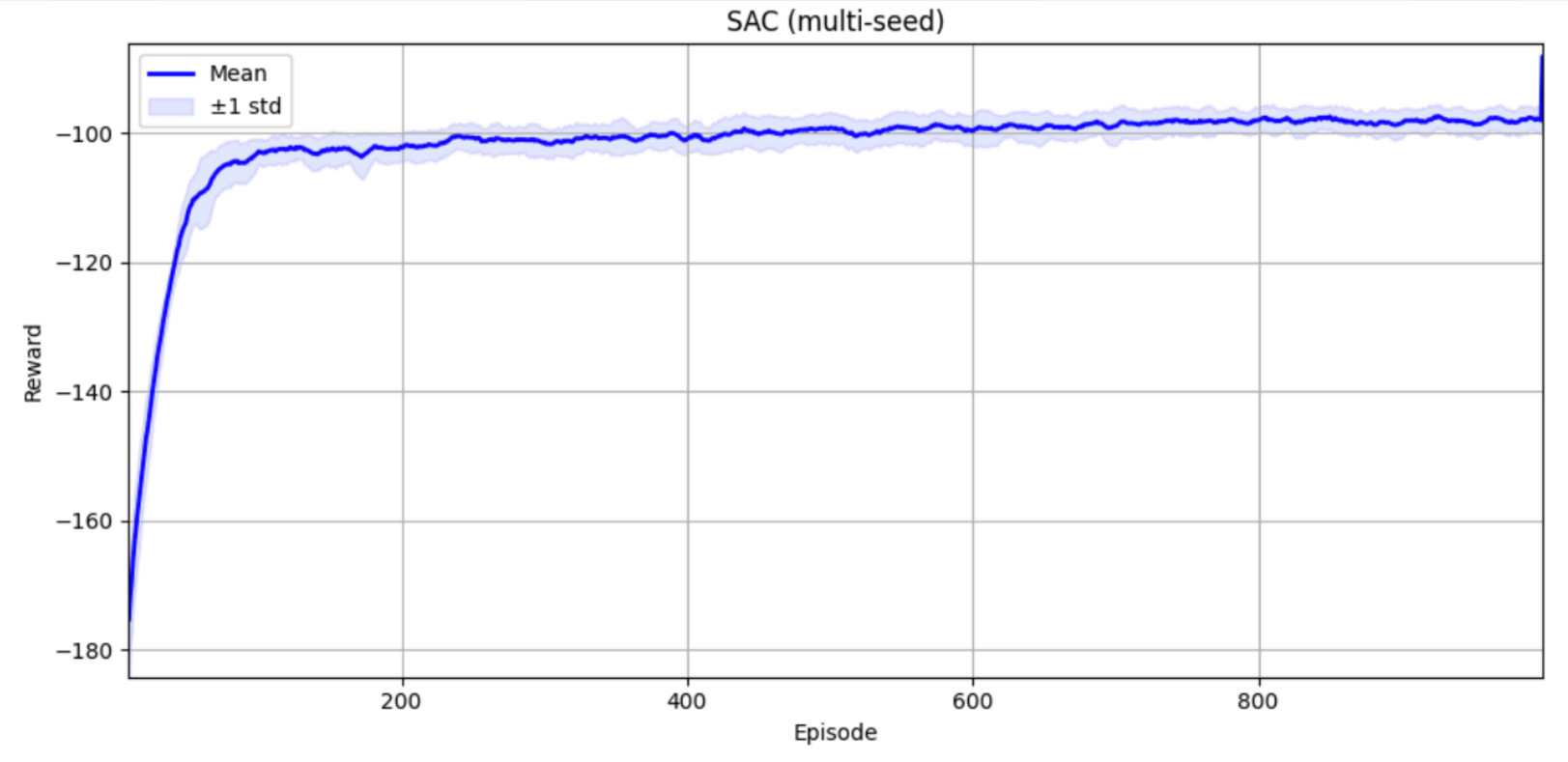
SAC
Tuning

Parameter	Value	Description
Policy	MlpPolicy	Multi-Layer Perceptron policy
Learning Rate	3e-4	Optimizer step size
Batch Size	512	Size of minibatches for training
Gamma (γ)	0.99	Discount factor for future rewards
Tau (τ)	0.005	Soft update coefficient for target networks
Entropy Coefficient	Auto-tuned	Balances exploration/exploitation

This SAC setup uses an MLP policy trained with a 3e-4 learning rate on mini batches of 512, applying a discount factor of 0.99 and soft-target updates with tau=0.005 for stable value learning. The entropy coefficient is auto-tuned to dynamically balance exploration and exploitation.

SAC

Results



The SAC plot shows the multi-seed mean reward rapidly rising from around -180 to about -100 within the first -50 episodes, then plateauing with minimal variance (shaded band) for the remainder of training. This indicates fast convergence and stable performance.

Proximal Policy Optimization (PPO)

🔧 Implementation Overview

- **Algorithm**: Proximal Policy Optimization (PPO)
- **Environment**: CityLearn with custom reward function
- **Training Episodes**: 1000
- **Seeds**: 5 different random seeds for robust evaluation
- **Wrappers**:
  - `NormalizedObservationWrapper` : Normalizes observations for stable training
  - `StableBaselines3Wrapper` : Ensures compatibility with SB3

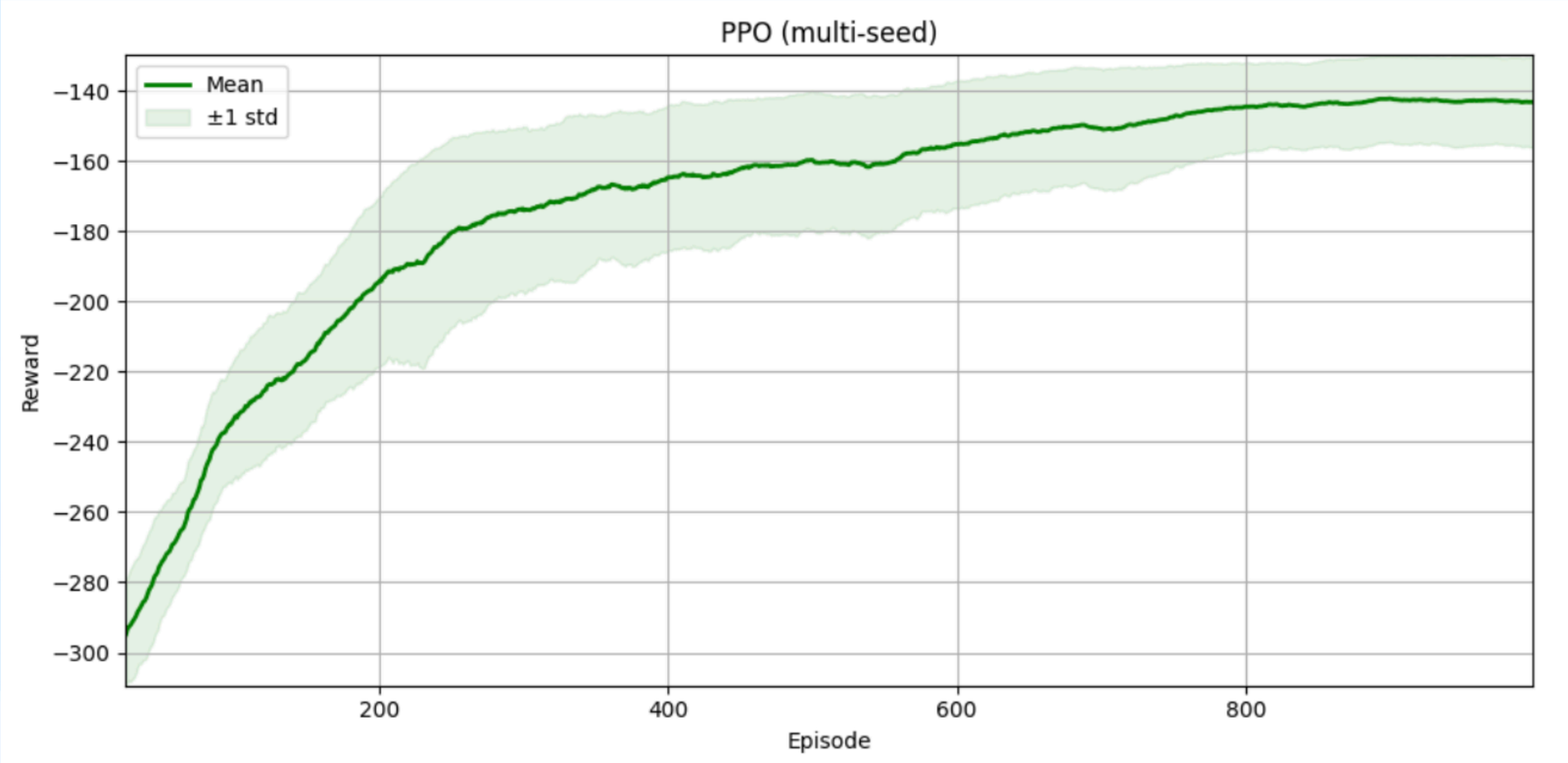
PPO
Tuning

Parameter	Value	Description
Policy	MlpPolicy	Multi-Layer Perceptron policy
Learning Rate	3e-4	Optimizer step size
Batch Size	64	Size of minibatches for training
N Steps	1024	Number of steps per update
N Epochs	10	Number of epochs for optimization
Gamma (γ)	0.99	Discount factor for future rewards
Clip Range	0.2	Clipping parameter for policy updates
Entropy Coefficient	0.01	Encourages exploration
Target KL	0.05	Maximum KL divergence between updates

This PPO setup uses a multilayer-perceptron policy trained on 1024-step rollouts divided into mini batches of 64, optimized over 10 epochs with a learning rate of 3e-4 and discount factor of 0.99. The clip range (0.2), entropy bonus (0.01), and target KL (0.05) strike a balance between stable updates and sufficient exploration.

PPO

Results



The plot shows the average episodic reward across multiple seeds steadily climbing from around -300 to about -150 over 1000 episodes, demonstrating consistent policy improvement. The shaded ±1 std band narrows over time, indicating that training converges and becomes more stable as learning progresses.

Twin Delayed DDPG (TD3)

🔧 Implementation Overview

- **Algorithm**: Twin Delayed DDPG (TD3)
- **Environment**: CityLearn with custom reward function
- **Training Episodes**: 1000
- **Seeds**: 5 different random seeds for robust evaluation
- **Device**: Automatically uses GPU if available, falls back to CPU
- **Wrappers**:
  - `NormalizedObservationWrapper` : Normalizes observations for stable training
  - `StableBaselines3Wrapper` : Ensures compatibility with SB3

TD3

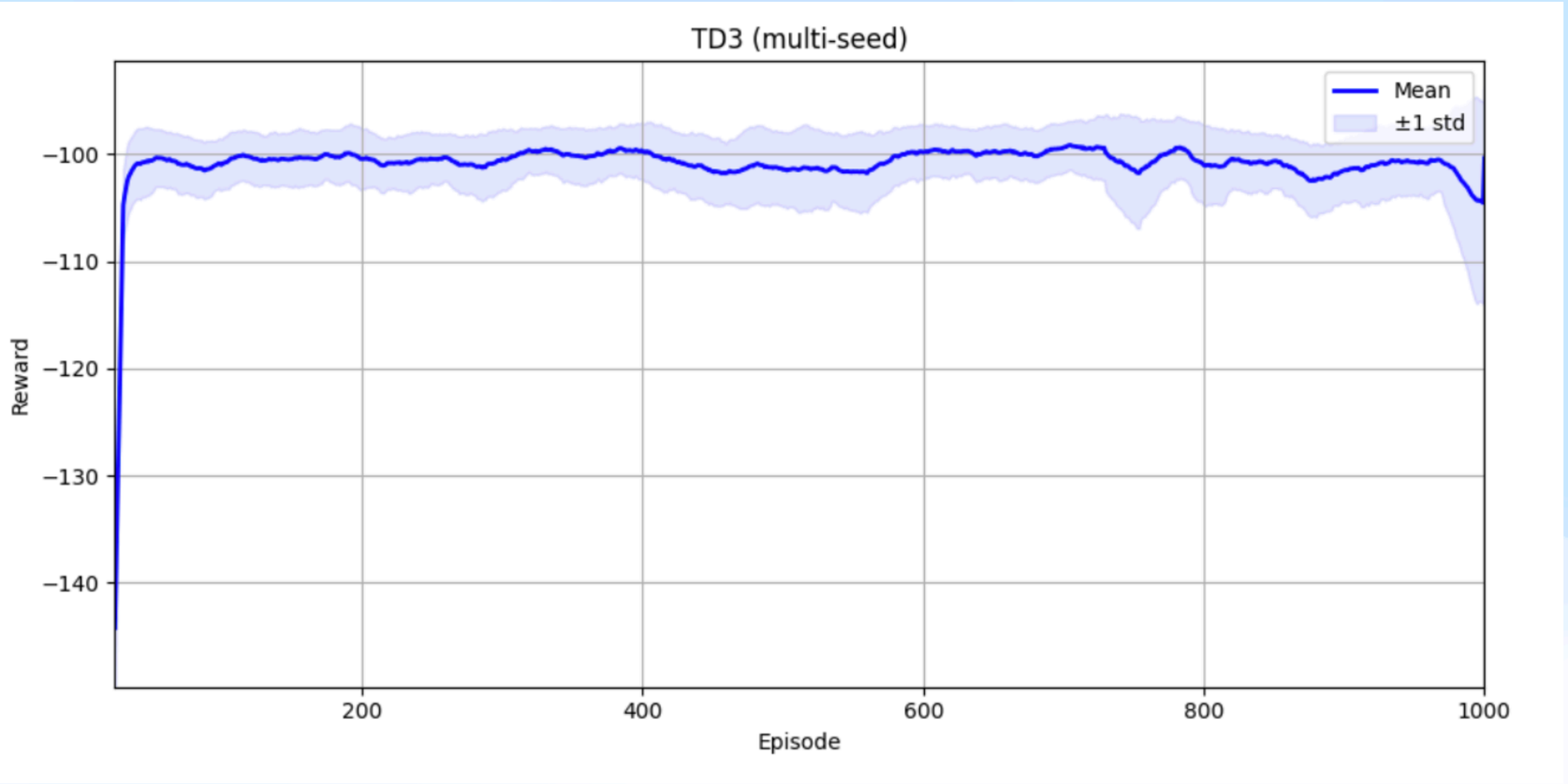
Tuning

Parameter	Value	Description
Policy	MlpPolicy	Multi-Layer Perceptron policy
Learning Rate	1e-3	Optimizer step size
Batch Size	128	Size of minibatches for training
Buffer Size	50,000	Size of experience replay buffer
Tau (τ)	0.001	Soft update coefficient for target networks
Gamma (γ)	0.99	Discount factor for future rewards
Policy Delay	2	Frequency of policy updates relative to Q-functions
Target Noise	0.2	Stddev of noise added to target actions
Noise Clip	0.3	Range for clipping target noise

This TD3 setup uses an MLP actor-critic trained with a 1e-3 learning rate on mini batches of 128 sampled from a 50000-transition replay buffer, with soft target updates (0.001) and discount factor 0.99. Policy updates occur every two Q-function steps, and target actions are perturbed with clipped Gaussian noise (0.2, clip ±0.3) for smoother value estimation.

TD3

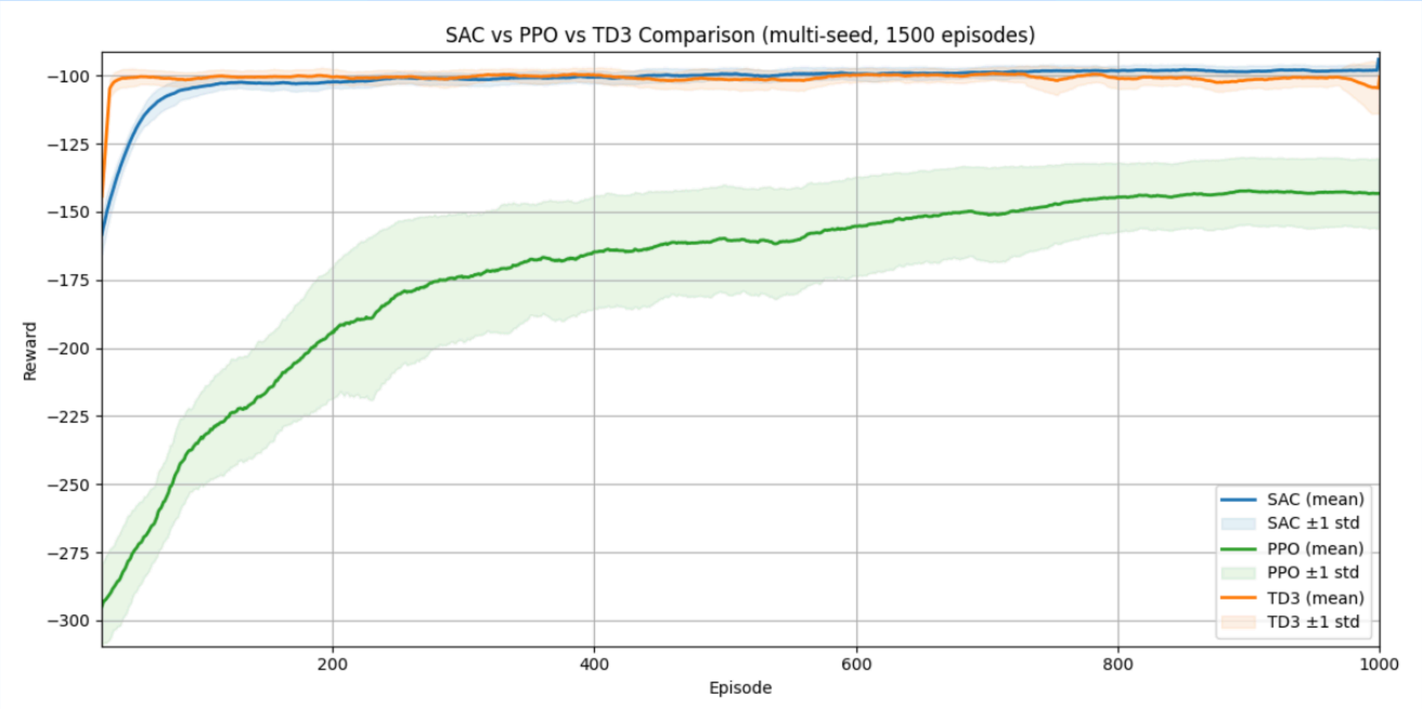
Results



The TD3 agent rapidly improves from an initial mean reward of around -275 to about -00 within the first ~30 episodes, then plateau with stable performance near -100 for the remainder of the training. The narrow ±1 std band demonstrates low variance across seeds.

Algorithm Comparison

Performance Comparison: SAC vs PPO vs TD3



Algorithm	Final Reward	Convergence Speed
SAC	High (~ -100)	Fast (~100 ep)
PPO	Lower (~ -150)	Slowest (~1000 ep)
TD3	High (~ -100)	Fastest (~20 ep)

NeurIPS 2023 CityLearn Challenge Control Track

Control Track

Objective: Optimize control of batteries, hot-water storage and heat pumps to minimize carbon emissions, mantain comfort, smooth grid load and withstand outages.

Environment: synthetic neighborhood of six buildings equipped with PV, storage (electrical & DHW) and heat pumps

Approach: Develop RL policy to dispatch resources under normal and outage conditions

Scoring (ScoreControl): weighted sum of Comfort, Emissions, Grid-performance and Resilience metrics

My approach

Environment & Reward Design

- **Custom Gym wrapper** (src/citylearn_rl/env.py)
    - Wraps CityLearn's six-building simulation
    - Exposes observations (temperatures, SOCs, PV generation, price signals) and discrete dispatch actions
- **Modular KPI rewards** (src/citylearn_rl/reward.py)
    - Comfort deviation: penalize indoor-temp drift outside bounds
    - Emissions: CO2 footprint of heat-pump & grid imports
    - Grid smoothing: reward lowering ramp & peak loads
    - Resilience: bonus for meeting demand during stochastic outages
- **Configurable weighting**
    - **Phase I:** emphasize grid (w = 0.6), comfort (w = 0.3)
    - **Phase II:** add resilience (w = 0.3) and rebalance others

Reward Design

Reward Function

Overall objective: maximise a **weighted combination** of four KPI-based scores:

$$\text{ScoreControl} = w_1 \times \text{ScoreControlComfort} + w_2 \times \text{ScoreControlEmissions} + w_3 \times \text{ScoreControlGrid} + w_4 \times \text{ScoreControlResilience}$$

Control Track KPIs

- **Carbon Emissions (G)** - Emissions from imported electricity
- **Discomfort (U)** - Occupied hours where indoor temperature deviates outside comfort band
- **Grid KPIs** (averaged): Ramping R, 1-Load Factor L, Daily Peak, All-time Peak
- **Resilience KPIs** (Phase II only, averaged): 1-Thermal Resilience M, Normalized Unserved Energy S

All KPIs are **normalized** by the corresponding baseline.

My approach

Algorithm & Training Pipeline

- **Stable-Baselines3 PPO**
    - scripts/train.py sets up vectorized ends (n_envs), policy (MlpPolicy), hyperparams (timesteps, clip)
    - Two-stage learning: first training on **2-KPI** case (thermal comfort & carbon emissions), and then **full multi-KPI** training to observe differences
- **Logging & Monitoring**
    - TensorBoard outputs in results/ppo_tensorboard_logs_*
    - Custom callback logs per-KPI scalars each rollout
- **Automation scripts**
    - scripts/plot_training.py: aggregate and plot learning curves
    - scripts/analyze_tb.py: export TB scalars -> CSV for downstream analysis

My approach

Evaluations

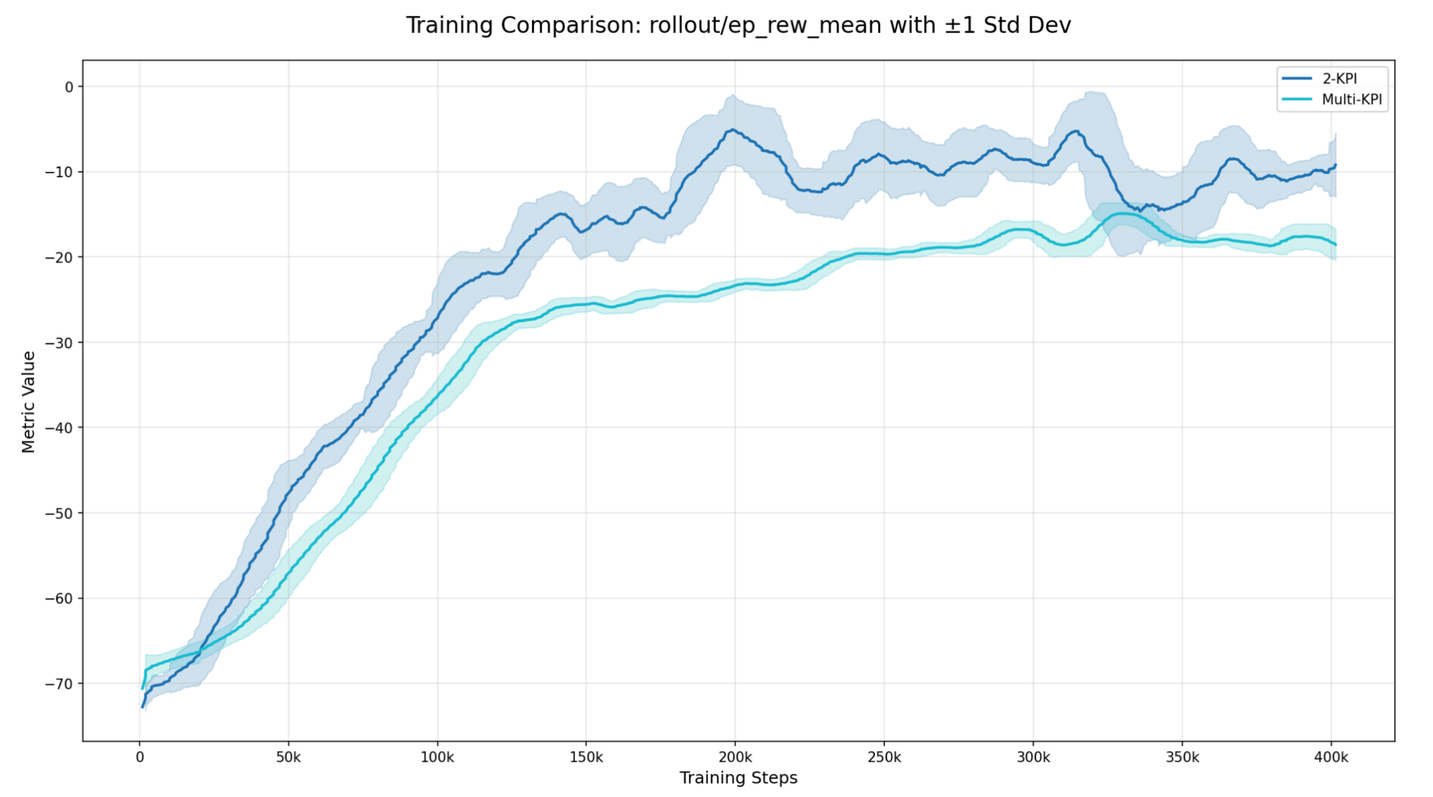
***PPO** is chosen over SAC/TD3 due to its **training stability** with multiple objectives and **robustness to environmental noise**. Its simplicity in hyper parameter tuning makes ideal for balancing both my 2KPI and multi-KPI setup.*

Performance analysis

- Compare comfort vs. emissions vs. grid smoothing trade-offs
- Sensivity to weight configuration

PPO rapidly improves mean episodic reward, climbing from -72 to around -10 over ~200k time steps in the 2-KPI setup, while the full multi-KPI variant converges more slowly around -20.

Reduced variance and plateauing after ~250k time steps indicate stable policy convergence, with the 2-KPI agent consistently achieving higher peak rewards and tighter performance bands.



Conclusions & Limitations

- **Suboptimal performance in complex multi-KPI setups:** PPO shows slower convergence and lower peak rewards when optimizing for comfort, emissions, grid, and resilience simultaneously

- **Reward sensitivity:** performance heavily depends on KPI weight tuning, making generalization across environments and use-cases difficult

- **No temporal foresight:** agents act purely reactively based on current observations, without modeling or predicting future demand, prices, or weather patterns

- **Simplified action space:** actions are discrete and manually designed, limiting the flexibility and realism of resource dispatch strategies

Future Work

- **Integrate predictive control:** combine reinforcement learning with time-series forecasting (e.g., weather, occupancy, demand) to improve performance under uncertainty

—> see: *Liu et al., "Forecasting-Aware Control for Urban Energy Systems", BuildSys 2023*

https://dl.acm.org/doi/10.1145/3583780.3614653

- **Expand to hybrid action spaces:** enable continuous + discrete actions for more fine-grained, realistic control of HVAC, battery, and DHW systems

- **Multi-agent coordination:** extend to decentralized RL policies for each building with communication or shared goals to reflect real-world grid scenarios

- **Transfer learning & generalization:** study how well trained agents perform on unseen building layouts or weather profiles, and explore domain adaptation techniques

Links & References

Official CityLearn repo: https://github.com/intelligent-environments-lab/CityLearn

My “CityLearnRL" framework: https://github.com/djacoo/CityLearnRL-Bioinformatica/tree/main

My “AICrowdTesting” framework: https://github.com/djacoo/AICrowdTesting