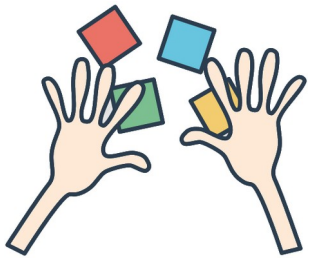


Hands-on: #1

Policy Gradient (REINFORCE) for simple problems

Activity #1:

01B_Rfrc_frozenlakeV2.py:
This is the Policy gradient method for the frozen lake.

- Run the code.
- Try and find a solution.
- Explain your results. Does the agent reach its goal?

Adjust hyperparameters:

- Learning rate
- Discount factor
- Numbre of iteration
- is_slippery
- Test your results.

Activity #2:

02B_Rfrc_cartpoleV2.py:
This is the Policy gradient method for the frozen lake.

- Run the code.
- Try and find a solution.
- Explain your results. Does the agent reach its goal?

Adjust hyperparameters:

- Learning rate
- Number of trajectories
- Discount factor
- Numbre of iteration
- Test your results.
- Network: hidden layer size
- Rewards Shaping
- Episode length

Suggested Exercises for REINFORCE on CartPole

Hyperparameter Tuning

- Learning Rate: Increase or decrease the (learning rate) in the Adam optimizer. Observe whether the agent balances the pole more quickly or gets stuck.
- Number of Trajectories (N): Adjust how many episodes you collect per iteration. For instance, run more episodes per update (e.g., N=10 vs. N=1) and see how it affects stability and convergence.
- Discount Factor (γ): Try different discount factors (like 0.95, 0.99, 0.999) to see if the agent focuses more on immediate stability vs. long-term control.

Network Architecture

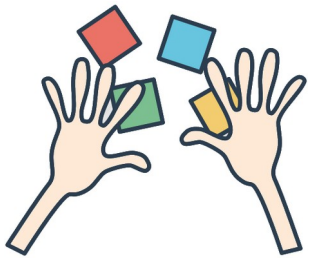
- Hidden Layer Size: The default code might have one hidden layer with a certain width (e.g., 128 neurons). Decrease or increase it (64 or 256) and note any impact on learning speed and final reward.
- Additional Hidden Layers: If the code only has one hidden layer, add a second layer. Does it improve performance or slow down training?

Reward Scaling

- Reward Shaping: Introduce a small negative reward for large angle deviations or for each step spent far from the pole's vertical alignment. Observe if the agent learns faster or overfits to partial goals.
- Episode Length: Track how many time steps the cart balances the pole. If the environment allows a maximum of 500 steps, see if the agent can hit that maximum regularly.

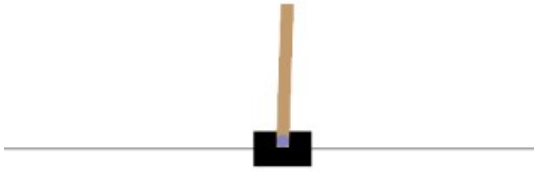
Performance Logging

- Episode Length vs. Reward: Plot the average episode length alongside total reward. Does the agent increase both in parallel?
- Policy Entropy: (Optional advanced) Keep track of the entropy of the policy over time to see if exploration declines as it converges.



Hands-on: #2

Stable Baseline3 for Cart pole



03_SB3cartpoleSimple.py

Run the Codes

- Execute the provided script to train the model with default hyperparameters and watch the agent balance the pole.

Adjust the Number of Training Steps

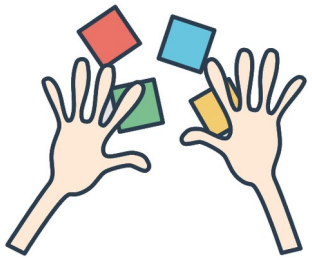
- Change total_timesteps=10000 to something smaller (e.g., 5000) or larger (e.g., 20000) and rerun.
- Question: Does increasing training steps improve performance or speed up convergence?

Hyperparameter Tweaks

- Experiment with the learning_rate parameter inside the PPO constructor (PPO("MlpPolicy", env, learning_rate=..., ...)).
- Question: Does a higher or lower learning rate affect stability?

Record Observations

- After each tweak, note whether the agent balances longer or if training is slower/faster.
- Question: Which settings yield the most reliable performance?



Hands-on: #3

Stable Baseline3 with callback

04_SB3cartpoleCallback.py

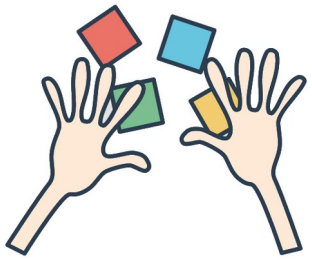
In this version, we use a callback to track rewards or plot performance during training. Your task is to modify parameters, observe how the metrics change, and discuss your findings.

1. Run the Code
   - Execute the script
   - Observe any live plots or logged outputs from the callback.

2. Experiment with Hyperparameters
   - Change total_timesteps to see if more training leads to faster convergence or better final rewards.
   - Adjust n_steps in PPO("MlpPolicy", env, n_steps=2048, ...) if available. How does it affect your reward curve?

3. Examine Callback Outputs
   - Look at the reward plots or logging from the callback:
   - Does it plateau at some point or fluctuate?

4. Discussion Questions
   - Which parameter change had the biggest impact on the final performance?
   - How stable is the training curve over time (do you see large spikes or dips in reward)?
   - Compare any differences in the reward curve for a smaller n_steps vs. a larger one.



Hands-on: #4

Custom Rewards Shaping and Action Shaping

05A_CustReward.py
05B_ActionBased.py

In this lab, you will work with two different DQN-based code examples for modifying rewards in LunarLander:

Reward Wrapper

- The default LunarLander-v3 environment is wrapped with a custom class that injects fuel penalties into the reward after each step.

Subclassing the Environment

- A new environment class (CustomLunarLanderEnv) inherits from the original and overrides the step() method to apply custom reward logic internally.
- You will run both versions, observe and compare their performance, and discuss which approach might be preferable under various circumstances.

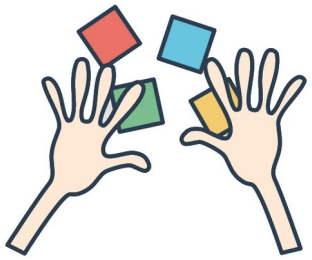
Hands-on: #4

Using a Reward Wrapper (First Approach)

- You take the existing LunarLander environment (e.g., "LunarLander-v3") and wrap it in a custom ActionBasedRewardWrapper that overrides the step() method after the environment returns its reward. The wrapper then modifies or penalizes that reward based on the action taken (fuel usage, etc.).
- This approach leaves the base environment code unchanged and simply uses the wrapper to inject custom reward logic. It’s often cleaner if you only need to adjust rewards and not touch the core environment implementation.

Inheriting from LunarLander and Overriding (CustomLunarLanderEnv)

- You create a subclass of LunarLander (CustomLunarLanderEnv) and override its step() method directly. Inside this new class, you do all the environment logic including your custom reward logic in a single place.
- The result is effectively the same—your final reward is penalized for fuel usage—but the difference is that you’re extending or replacing the environment’s code rather than wrapping it.
- This approach can be more flexible if you want to override other environment methods (e.g., reset(), custom observations, done criteria, etc.). However, it’s more “intrusive” since you rely on details of the base environment code.
- In both approaches, the final effect is that you run a DQN on a variant of Lunar Lander with custom reward modifications. The difference is simply whether you’re injecting the reward change from outside (via a wrapper) or within (by subclassing and overriding the step() method).



Hands-on: #5

Refactored code: DQN and REINFORCE

06_refactored

1. CustomLunarLanderEnv
   - Subclass of LunarLander that overrides step() to modify the reward with a fuel penalty.

2 Two Different Approaches
   - DQNTrainer: Uses Stable-Baselines3 to create and train a DQN model.
   - REINFORCETrainer: A self-implemented policy gradient approach that collects episodes, computes returns, and updates the policy network accordingly.

3 Switching Methods
   - In the main block, set method = "dqn" or method = "pg" to select your training method.
   - Each approach has its own train() and evaluate() logic, but both use the same custom environment with the fuel penalty.

4 Reward Logging and Plots
   - DQN logs rewards via a RewardTrackingCallback that updates a Matplotlib figure and saves it to results/reward_plot.png.
   - REINFORCE logs average return in a second Matplotlib figure, saving it to results/reinforce_reward_plot.png.

5 Saving and Loading
   - DQN: The final model is saved to dqn_custom_lunar_lander.zip.
   - REINFORCE: The final policy network is saved to reinforce_lunar_lander.pth.

6 Evaluation
   - For DQN: calls agent.evaluate() and loads dqn_custom_lunar_lander.zip if not already loaded.
   - For REINFORCE: calls pg_agent.evaluate() which loads reinforce_lunar_lander.pth if needed, then runs a few episodes in human mode.