Reinforcement and Imitation learning in a Gym Environment

Orestis Konstnataropoulos
orestiskonsta@gmail.com
ECE NTUA
Neuro-Fuzzy Control

Abstract

In this project we analyze, develop and test the most prominent algorithms of reinforcement and imitation learning. We test the implementations on the Gym Lunar Lander simulation environment and provide a GitHub repository for users to train various agents and visualize them, adjust hyperparameters to their preferences or load pretrained models.

1 Introduction

1.1 Reinforcement and Imitation Learning

Nowadays the go-to method for getting machines to solve difficult learning problems is the supervised learning paradigm. One can collect vast ammounts of instances of a problem and a ground truth, a solution for each instance and assume that these instances are independent and identically distributed. However, infants do not require constant supervision in order to learn. Just by interacting with the environment they produce a wealth of information on which they manage to understand world patterns and create their own behaviours.

The fundamental idea underpinning reinforcement learning in the artificial intelligence community is the concept of learning through interaction. In this approach, an agent learns by trial and error, making decisions and adjusting its current state based on the often unknown dynamics of its environment. At the heart of most reinforcement learning problems lies the reward signal. The agent aims to maximize its expected reward while facing the challenge of balancing exploration and exploitation. However, crafting a reward function that accurately captures the desired behavior of the agent can be a challenging task. Additionally, reinforcement learning algorithms are typically highly sensitive to hyperparameters and are inefficient in terms of sample usage, requiring a large number of interactions to converge.

Bridging the two previously mentioned paradigms imitation learning aims to teach an agent to perform a task by observing and mimicking demonstrations provided by a teacher or expert. Unlike traditional reinforcement learning approaches leverages the knowledge encoded in expert demonstrations to accelerate the learning process. Now the design of a reward function is avoided as it is described implicitly through demonstrations. By learning directly from expert behavior, imitation learning enables agents to acquire complex skills and behaviors efficiently, making it particularly useful in domains where expert guidance or human expertise is available. This approach finds applications in various fields, including robotics, autonomous driving, natural language processing, and computer vision, where learning from human demonstrations can significantly enhance the performance of intelligent systems.

1.2 Project Objective

Within the scope of this project, we explore, develop, and evaluate several leading algorithms in reinforcement and imitation learning. In this GitHub repository[10], I have implemented these algorithms from scratch and offer a framework that enables users to train various agents, adjust hyperparameters to their preferences, and visualize outcomes within a Gym environment. The developed algorithms include (Double) Deep Q-Learning, Proximal Policy Optimization, Behavioral Cloning, Generative Adversarial Imitation Learning, and Goal-Conditioned Supervised Learning. In the following sections we will theoretically analyze the aforementioned algorithms.

1.3 The Lunar Lander simulation environment

In order to test the developed algorithms we use the Lunar Lander simulation environment provided by OpenAI Gym. The objective for the agent is to master the safe landing of a lunar module onto the lunar surface. The environment's state space comprises eight dimensions, encompassing the X and Y coordinates, X and Y velocity, angle, and angular velocity of the lander, as also two boolean variables indicating whether the left and right legs of the lander have made contact with the moon.

Upon successfully landing, the agent receives a reward of +100, while crashing results in a penalty of -100. Moreover, the agent receives "shaping" rewards throughout each step of the process. Positive rewards are granted for actions that bring the lander closer to [0,0], reduce velocity, align the lander upright, and make contact with the lunar surface using the lander's legs. Conversely, negative rewards are incurred for actions that move the lander away from the landing site, increase velocity, deviate from an upright orientation, raise the lander legs off the moon's surface, or consume fuel (via thruster firing). The highest attainable score for an agent in a single episode is approximately +250.

1.4 Notation

We consider our environment as an Markov Decision Process with model dynamics

$$P(s_{t+1}|s_t, a_t, ..., a_0) = P(s_{t+1}|s_t, a_t)$$

and a deterministic reward function $R(s)$. The agent aims to learn a policy π which is a mapping from the agent state to a distribution over actions $\pi(a_t|s_t)$. Given a policy π and discount factor $\gamma \in [0, 1]$, a value function V^π is an expected sum of discounted rewards,

$$V^\pi(s) = \mathbb{E}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots | s_t = s]$$

A q-function for a state s and an action a is defined as the expected return starting from the state $s_t = s$ at time t and taking the action $a_t = a$ and then subsequently following the policy π. It is written mathematically as

$$Q^\pi(s, a) = \mathbb{E}[R_t + \gamma V^\pi(s_{t+1})|s_t = s, a_t = a]$$

Let us note that Lunar Lander is a fully observable MDP.

2 Reinforcement Learning Fundamental Algorithms

2.1 Deep Q-Learning

Deep Q-learning, introduced by Mnih et al. [4], builds upon the foundational Q-learning algorithm, which has been known for decades. The traditional Q-learning algorithm is an off-policy temporal difference method that aims to learn the state-action tabular value function Q. It employs an epsilon-greedy policy based on Q to interact with the environment and updates Q based on the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

where α is the learning rate. An epsilon greedy policy is a policy in witch with probability ϵ agent selects a random action (explore), and with probability $1 - \epsilon$, selects the action with the highest estimated q value (exploit).

Mnih et al. extended this algorithms and manage to play Atari games in better than human levels. They model this q function as an Multilayer Perceptron that takes as input agents state and outputs a

value for each distinct possible action. The input state can even be the raw image of the environment at specific instance. This MLP learns by minimizing the following error:

$$J(w) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1})} \left[(y_t^{DQN} - \hat{q}(s_t, a_t, w))^2 \right]$$

where y_t^{DQN} is the one-step ahead learning target:

$$y_t^{DQN} = r_t + \gamma \max_{a'} \hat{q}(s_{t+1}, a', w^-)$$

where w^- represents the parameters of the target network, and the parameters w of the online network are updated by sampling gradients from minibatches of past transition tuples (s_t, a_t, r_t, s_{t+1}). The targets in the loss function are treated as fixed during gradient descent although dependent by a previous version of w. That is why Q-learning is considered as a semi-gradient algorithm.

Two more important major aspects of deep q-learning is the replay buffer and the target network:

- The agent stores state, action, next state and reward tuples to a limited capacity replay buffer after interacting with the environment. From this buffer the agent samples randomly a tuple and updates its weights. This way the data efficiency is increased as a sample can be used multiple times, while the sample correlation of consecutive samples breaks reducing update variance.
- Two networks are used while training. Except form the online network a target network is used to generate the target values. It is updated by softly copying parameters from the online network every several iterations. That results to enhancement of learning stability.

When computing the target values we simultaneously and using the same network choose the best action and compute its q value. In this scenario, there's a higher probability of selecting values that are overestimated, leading to an overly optimistic estimate of the target value. The Double DQN algorithm[5] proposes that the online network chooses greedily the best action, whereas the target network with parameters w^- estimates its q value. So the target becomes:

$$y_t^{DoubleDQN} = r_t + \gamma \hat{q} \left(s_{t+1}, \arg\max_{a'} \hat{q}\left(s_{t+1}, a', w\right), w^- \right)$$

The Q learning agent in our experiments justifies its fame as it performs great and obtains high reward with low variance.

2.2 Proximal Policy Optimization

2.2.1 Policy Gradient methods

Policy gradients methods are straight forward as they directly attempt to differentiate the reinforcement learning objective and then perform gradient ascent on the policy parameters. Let us first define the probability of a trajectory $\pi(\tau)$ as:

$$p(\tau) = p(s_1) \prod_{t=1}^{T-1} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

The objective of policy gradient is to find policy parameters θ so as to maximize $J(\theta)$:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \int \pi_\theta(s) R(\tau)\, d\tau \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \gamma^t R(s_{i,t}, a_{i,t})$$

So to get an estimation of $J(\theta)$ we just need to get the average sum of the rewards of trajectories generated with our policy. We now need to compute the gradient of the objective function:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(s) R(\tau)\, d\tau = \int \pi_\theta(s) \frac{\nabla_\theta \pi_\theta(s)}{\pi_\theta(s)} R(\tau)\, d\tau = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta log \pi_\theta(s) R(\tau)]$$

After some more operations:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau) r(\tau)\right]$$
$$= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \left[\log P(s_1) + \sum_{t=1}^{T}\left(\log \pi_\theta(a_t|s_t) + \log P(s_{t+1}|s_t, a_t)\right)\right] r(\tau)\right]$$
$$= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \left[\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t)\right] r(\tau)\right]$$
$$= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t=1}^{T}\gamma^t r(s_t, a_t)\right)\right]$$
$$\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\left(\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{t=1}^{T}\gamma^t r(s_{i,t}, a_{i,t})\right)\right)$$

With the last equality we have a way to approximate the gradient of the objective function, perform gradient ascent and improve our policy. However, this algorithm is on-policy, sample inefficient and has high variance as it largely depends on rewards.

2.2.2 Actor-Critic paradigm

Observing that previous rewards cannot be affected by future actions we modify $\nabla_\theta J(\theta)$ as:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\left(\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{t=t'}^{T}\gamma^{t'} r(s_{i,t}, a_{i,t})\right)\right)$$
$$= \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\left(\nabla_\theta \log \pi_\theta(a_{i,t}, s_{i,t})\hat{Q}_{i,t}\right)$$

where $\hat{Q}_{i,t}$ is an estimate of the reward to go. It can be proven that subtracting a baseline from the reward to go does not insert bias to the estimator. We choose for baseline an estimation of the value function of the state and introduce Advantages $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$. This means that we also have to learn a network, an MLP that estimates a value given a state. So we now have:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\left(\nabla_\theta \log \pi_\theta(a_{i,t}, s_{i,t}) A^{\pi_\theta}(s,a)\right)$$

Let us here denote that with the value function approximator MLP we can also approximate the advantages as $A^{\pi_\theta}(s_t, a_t) \approx r(s_t, a_t) + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)$. So we can have a critic MLP that approximates the advantages and learns the value function by minimizing them and an actor MLP that performs policy gradient on the last objective.

2.2.3 Clipping gradients

It can be proven that we can estimate the gradient $J(\theta)$ for an off-policy setting of the previous algorithm by maximizing:

$$L_\theta^{CPI} = \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] = \mathbb{E}_t\left[r_t(\theta)\hat{A}_t\right]$$

In [6], Schulman et al. propose a variation of objective function L^{CPI}, L^{CLIP}.

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

With this objective function the policy update is constrained to not overshoot if a trajectory is too good or too bad. The full algorithm is described bellow 19. Note that in this algorithm is important to be able to run on multiple environments simultaneously. A roll-out step hyperparameter is also crucial and defines how many step to take before bootstrapping for the estimation of target values.

PPO has recently gained popularity and it is used even in tasks related to LLMs. In our experiments PPO agent seem to perform FINE in Lunar Lander environment, although its obtained reward has high variance.

3 Imitation Learning Fundamental Algorithms

In the next two section the agents are provided with expert demonstrations. These demonstrations $\xi = \{(s_0, a_0), (s_1, a_1), \ldots\}$ are state-action pairs and can be provided by either human or a pretrained reinforcement learning agent.

Algorithm 1 Proximal Policy Optimization (PPO)

Initialize policy network π_θ with parameters θ
Initialize value network V_ϕ with parameters ϕ
for iteration = 1 to max_iterations **do**
  Collect trajectories using π_θ in the environment
  Compute advantages A_t using V_ϕ
  for epoch = 1 to num_epochs **do**
    Shuffle trajectories
    for mini-batch in trajectories **do**
      Compute ratio $r_t(\theta)$ using π_θ and $\pi_{\theta_{\text{old}}}$
      Compute clipped objective $\mathcal{L}_{\text{clip}}$ using $r_t(\theta)$ and A_t
      Compute value loss $\mathcal{L}_{\text{value}}$ using V_ϕ and A_t
      Compute entropy bonus $\mathcal{L}_{\text{entropy}}$ using π_θ
      Compute total loss $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{clip}} + \mathcal{L}_{\text{value}} - c \cdot \mathcal{L}_{\text{entropy}}$
      Update policy network parameters θ using gradient descent on $\mathcal{L}_{\text{total}}$
      Update value network parameters ϕ using gradient descent on $\mathcal{L}_{\text{value}}$
    end for
  end for
  Update old policy $\pi_{\theta_{\text{old}}}$ to match current policy π_θ
end for

3.1 Behavioural Cloning

Behavior cloning[7] methods leverage a collection of expert demonstrations represented by ξ to construct a policy π that mirrors the behavior of the expert. This is typically achieved using supervised learning methodologies, wherein the objective involves minimizing the disparity between the learned policy and the expert demonstrations according to a specific metric. More precisely, the aim is to address the optimization task:

$$\hat{\pi}^* = \arg\min_\pi \sum_{\xi \in \Xi} \sum_{x \in \xi} L(\pi(s), \pi^*(s))$$

However, this strategy might not result in optimal performance because the learning process relies solely on a limited set of examples provided by the expert and these demonstrations may not cover the entire range of states uniformly. Consequently, the learned policy is prone to underperforming when encountering states not adequately represented in the demonstration set ξ. This issue is especially prominent when the expert demonstrations follow a sequential trajectory of states and actions, shaping the distribution of sampled states x in the dataset according to the expert policy. Consequently, when applying an estimated policy $\hat{\pi}^*$ in practice, it generates its own distribution of states to be explored, which may differ from those encountered in the expert demonstrations. This disparity in distributions leads to compounded errors, presenting a significant challenge in imitation learning.

The above issues are encountered in our implementation of this simple behavioural cloning approach. Only after many epochs with different demonstrations will the agent manage to perform well on the Lunar Lander environment.

3.2 Generative Adversarial Imitation Learning

Generative Adversarial Imitation Learning[8] integrates adversarial networks with the imitation learning paradigm. Now, except from state action tuples provided by an expert, we also leverage a reinforcement learning agent based on a policy gradient method such as PPO. The PPO agent acts as a generator that provides synthetic state action tuples to a simple MLP classifier that acts as a discriminator. Discriminator's objective is to distinguish between tuples provided by the expert and the PPO agent. The PPO agent performs gradient ascent treating the logarithm of discriminator probabilities as its reward signal. Formally, we wish to find a saddle point (π, D) for the expression:

$$\mathbb{E}_\pi[\log(D(s,a))] + \mathbb{E}_\pi[\log(1 - D(s,a))]$$

The full algorithm of GAIL is provided below. This method seemed successful in our experiments, as it solves the Lunar Lander simulation.

Algorithm 2 Generative Adversarial Imitation Learning

Input: Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w
for $i = 0, 1, \ldots$ **do**
Sample trajectories $\tau_i \sim \pi_{\theta_i}$
Update the discriminator parameters from w_i to w_{i+1} with the gradient

$$\hat{E}_{\tau_i}[\nabla_w \log(D_w(s,a))] + \hat{E}_{\tau_E}[\nabla_w \log(1 - D_w(s,a))]$$

Take a policy step from θ_i to θ_{i+1}, using the PPO rule with cost function $\log(D_{w_{i+1}}(s,a))$.
end for

3.3 Goal Conditioned Supervised Learning

Goal Conditioned Supervised Learning[9] is an imitation learning paradigm that does not need either a reward signal nor expert demonstrations. GCSL leverages the principle of hindsight and treats data that agent selects under its current policy as optimal data selected under optimal policy. Unlike the previous mentioned agents GCSL agent selects an action depending on its current state, as well as a goal that is trying to reach and probably the number of steps to reach it.

Let's say the agent just completed an episode where it was instructed to reach the goal g^*, but ended up traversing through various states and taking corresponding actions:

$$s_0, a_0, s_1, a_1, s_2, \ldots, s_{50}, a_{50}$$

Despite the intended goal g^*, the agent might have struggled to reach it, perhaps due to the final state s_{50} being far from g^*. However, we can still leverage the fact that the agent successfully reached s_{50}, using the data collected during this trajectory to improve its approach to reaching s_{50}. Furthermore, the agent also reached states like $s_{49}, s_{48}, \ldots$, providing additional data for enhancing its navigation to these states.

To elaborate, for any two time steps t_1 and t_2 ($t_1 < t_2$), we observe that after taking action a_{t_1} in state s_{t_1}, the agent arrived at state s_{t_2} after $t_2 - t_1$ time steps. This implies that we can treat a_{t_1} as the optimal behavior at s_{t_1} when aiming to reach s_{t_2} within $t_2 - t_1$ time steps. The algorithm performs a simple three-step procedure:

- Collects a trajectory with the current agent
- Picks many choices of t_1, t_2 to create a dataset $\mathcal{D}$ of optimal data $\mathbf{x} = (s = s_{t_1}, g = s_{t_2}, h = t_2 - t_1)$ and $\mathbf{y} = a_{t_1}$.
- Performs supervised learning (classification or regression depending on discrete actions) to have the policy produce output $\mathbf{y}$ when provided input $\mathbf{x}$.

Formally, the policy is performing maximum-likelihood estimation (MLE) via supervised learning.

$$\arg\max_{\theta} \mathbb{E}_{((s,g,h),a)\sim\mathcal{D}}[\log \pi_\theta(a|s = s, g = g, h = h)]$$

Our implementation and experiment with the above algorithm is very informative. Our agent after training given as goal a state in which he lies on point (0, 0) with both rocket feet on the ground manages to reach that desired state. However, not having access to the reward signal does not have any reason to land smoothly on the ground and the reward of its trajectories is very low. It seems that this algorithm fails in this environment. Perhaps, adding the trajectory length to the MLP input would increase GCSL performance.

References

[1] Sutton RS, Barto AG. Reinforcement learning: An introduction. MIT press; 2018.

[2] Stanford CS234 Notes, Michael Painter, Emma Brunskill

[3] Berkley C285 online lectures, Sergey Levine [4] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

[5] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." AAAI. Vol. 16. 2016.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[7] Stefan Schaal. 1999. Is imitation learning the route to humanoid robots? Trends in Cognitive Sciences 3, 6 (1999), 233–242

[8] Jonathan Ho, Stefano Ermon Generative Adversarial Imitation Learning

[9] Dibya Ghosh, Abhishek Gupta, Ashwin Reddy, Justin Fu, Coline Manon Devin, Benjamin Eysen- bach, and Sergey Levine. Learning to reach goals via iterated supervised learning. In Interna- tional Conference on Learning Representations, 2021. URL https://openreview.net/ forum?id=rALA0Xo6yNJ.

[10] git@github.com:k-orestis/rl-il-for-lunar-lander.git