Analysis of a Training Pipeline for Behavioral Cloning

Koushani Chakrabarty
Department of Computer Science
University of Buffalo
Buffalo, NY
koushani@buffalo.edu

July 5, 2023

Abstract

In the general Reinforcement Learning Framework, an agent navigates an environment, collecting observations of the environment at regular instances, and taking actions based on such observations. The actions of the agent result in the agent getting rewards, which may be positive or negative, and in a change of the agent's state. In the long term, the agent's goal is to find an optimal policy of navigating an environment, that maximises its cumulative rewards. This general Reinforcement Learning Framework is a powerful tool, and algorithms are constantly under design, to help the agent in its search for the optimal policy. Some of the fastest reinforcement learning algorithms tackle both continuous and discrete control problems, while research continues to search for ways to mitigate the problems caused by incomplete reward functions in a reinforcement learning environment.

From the above discussion it is evident that the reward function is the predominant governing factor in an RL agent's performance. Therefore, any environment the agent wants to learn, must have a finite definable reward function. However, in most real-world scenarios, that may not be the case. Often the reward-function itself is sparse, for example in a game where the reward is received only at the end of the game. In many other cases, like in the case of self-driving cars, there is no specific reward function, which demands the construction of custom-made reward functions. However, in an environment with many actions and many states, it may be next to impossible to design a reward function which helps the agent learn exactly the way we want it to learn. Thus, manually designing reward functions is not a feasible solution to the problem at hand.

In order to explore some alternative methods to tackle the indefinite reward function problem in Reinforcement Learning, we go back to the basics, i.e: Supervised Learning. While Behavioural Cloning is a special case of Imitation Learning that uses supervised learning to imitate an expert's policy, a behaviorally cloned agent can also use reinforcement

learning to further optimize it's policy. The expert agent's behaviour is presented as a dataset which contains observations, actions and rewards of the agent while following an unknown policy.

The dataset used in this case is OpenAI Gym's 'Pendulum-v1' Environment. By using a DDQN Agent's experience as a dataset, a behavioral cloning agent learns to mimic the DDQN Agent's policy, which should theoretically improve further with Reinforcement Learning. This Project Aims to explore that Theory.

The rest of the report is organized as follows:

1 Introduction

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment. In RL, an agent takes actions based on the current state of the environment, receives feedback in the form of rewards or penalties, and uses this feedback to update its policy or strategy. The ultimate goal is to learn a policy that maximizes the cumulative reward.

Reinforcement learning (RL) has been successfully applied to a wide range of applications.Perhaps the most famous application of RL is in game playing, where systems like DeepMind's AlphaGo and OpenAI's DOTA 2 playing agents have achieved superhuman performance in complex games. These agents are capable of exploring and learning strategies far beyond human capabilities, showcasing the power of RL when applied to discrete, well-defined problems.RL can be used to teach robots to perform tasks that are difficult to program directly. For example, RL has been used to teach robotic arms to manipulate objects and to help quadcopters learn to fly. RL allows robots to learn from trial-and-error, improving over time as they interact with their environment.RL can also be applied to problems of resource management. For instance, in data centers, RL can be used to manage cooling systems, leading to significant energy savings. In networking, RL can be used for traffic signal control, improving traffic flow and reducing congestion.RL can be used to make personalized recommendations to users. Unlike traditional recommendation systems that simply suggest the most popular or highest-rated items, RL-based systems can take into account the sequential nature of user behavior and make recommendations that will maximize user engagement over the long term.RL is used to train self-driving cars and drones. The agents learn to make decisions based on the environment and the current situation, which could include navigating traffic, changing lanes, overtaking, etc. In finance, RL can be used for algorithmic trading where the agent learns to buy or sell based on the stock's historical data.

RL is modeled as a Markov Decision Process, where the probability of transitioning to a new state depends only on the current state and action, and not on the history of past states or actions. RL algorithms can be broadly classified into three categories: value-based methods (e.g., Q-learning), policy-based methods (e.g., policy gradients), and actor-critic methods (e.g., Advantage Actor-Critic) that combine the strengths of the first two.

1.1 Value-Based Methods

In value-based reinforcement learning methods, the goal is to learn a value function, which is a measure of how good it is to be in a particular state or to take a particular action in a state. The most common example of this is Q-Learning, where the objective is to learn the action-value function Q(s,a) that measures the expected cumulative reward of taking action a in state s, following a particular policy.

Value-based methods typically involve maintaining and updating a table of values (in tabular methods) or a function approximator (in function approximation methods) for different state-action pairs. The policy is then derived from these values, often by choosing the action that maximizes the value in each state.

1.2 Policy-based methods

Policy-based methods directly learn the policy of the agent without having to learn a value function as an intermediate step. The policy is a mapping from states to actions and can be deterministic or stochastic.

These methods typically parameterize the policy and learn the parameters by maximizing expected reward. They can handle high-dimensional action spaces better than value-based methods and can also learn stochastic policies. However, they often suffer from high variance in their updates.

1.3 Actor-Critic methods

Actor-Critic methods are a hybrid of value-based and policy-based methods. They maintain both a policy (the "actor") and a value function (the "critic").

The actor's job is to choose actions, and the critic's job is to estimate the value function, which is used to critique the actions chosen by the actor. The policy is updated based on the feedback from the critic. This combination can lead to more efficient learning than either method alone.

The advantage of actor-critic methods is that they can leverage the strengths of both policy-based and value-based methods. They can use the value function to reduce the variance of policy-based methods while still being able to handle high-dimensional action spaces and learn stochastic policies. Examples of such methods include Advantage Actor-Critic (A2C) and Deep Deterministic Policy Gradient (DDPG).

In this project, three spicific Actor Critic Algorithms are explored, namely: SAC (Soft Actor Critic), PPO (Proximal Policy Optimization) and DDPG (Deep Deterministic Policy Gradient). A short analysis of the three algorithms are given below:

1.3.1 Soft Actor Critic (SAC)

SAC is a state-of-the-art actor-critic method that excels in continuous action spaces and achieves impressive sample efficiency. It introduces the concept of entropy regularization to balance exploration and exploitation. SAC maximizes

the expected return while also maximizing the policy's entropy, which encourages exploration. By incorporating entropy regularization, SAC can achieve better exploration and discover a more diverse set of policies. SAC also leverages a soft value function that is learned through a soft Q-learning algorithm. The soft value function estimates the state-action value, taking into account both the expected return and the entropy of the policy. By optimizing both the policy and the value function simultaneously, SAC achieves stable and effective learning.

The following are the advantages of using SAC

- **Exploration-Exploitation Trade-off**: SAC incorporates entropy regularization, which balances exploration and exploitation. By maximizing both the expected return and policy entropy, SAC encourages exploration in a principled manner. This leads to the discovery of diverse and effective policies.
- **Sample Efficiency**: SAC achieves high sample efficiency by effectively exploring the state-action space. By utilizing the entropy regularization, it encourages the agent to explore different actions, enabling faster learning and convergence.
- **Continuous Action Spaces**: SAC is well-suited for problems with continuous action spaces. It leverages the advantages of stochastic policies, allowing for exploration and flexibility in action selection.
- **Off-Policy Learning**: SAC can leverage off-policy data, making use of past experiences stored in a replay buffer. This enhances the learning process by increasing data efficiency and enabling more stable updates.
- **Soft Value Function**: SAC employs a soft value function that estimates the state-action value. This value function takes into account both the expected return and the entropy of the policy. By optimizing the soft value function, SAC achieves stability and robust learning.
- **Model-Free Approach**: SAC is a model-free reinforcement learning algorithm, which means it does not require any prior knowledge or explicit modeling of the environment dynamics. It can learn directly from interaction with the environment, making it applicable to a wide range of problems.

SAC has a few limitations as well, some of which are discussed below:

- **Complexity**: SAC can be more complex to implement and tune compared to simpler actor-critic algorithms. It involves training multiple networks, balancing entropy regularization, and selecting appropriate hyperparameters. Proper tuning is crucial for achieving optimal performance.
- **Sensitivity to Hyperparameters**: The performance of SAC can be sensitive to hyperparameter choices, such as the entropy regularization coefficient, learning rates, and neural network architectures. Careful tuning and experimentation are necessary to achieve desirable results.

- **Exploration in Sparse Reward Environments**: While SAC encourages exploration, it may face challenges in environments with sparse rewards. If rewards are rare or sparse, the exploration pressure from entropy regularization alone may not be sufficient to discover optimal policies. Additional techniques, such as reward shaping or auxiliary tasks, may be required.

- **Computational Requirements**: SAC can have high computational requirements, particularly when dealing with complex environments or large action spaces. Training deep neural networks and collecting sufficient samples for effective learning can demand significant computational resources.

Soft Actor-Critic (SAC) offers several advantages, including effective exploration, sample efficiency, suitability for continuous action spaces, and off-policy learning. However, it also has limitations related to complexity, sensitivity to hyperparameters, exploration in sparse reward environments, computational requirements, and the lack of strong theoretical guarantees. Understanding these advantages and limitations is crucial for selecting and applying SAC appropriately in reinforcement learning scenarios.

1.3.2 Proximal Policy Optimization (PPO)

PPO is a state-of-the-art actor-critic method that has gained significant attention for its simplicity, stability, and sample efficiency. It aims to strike a balance between stable policy updates and effective exploration. PPO achieves this by using a surrogate objective function that constrains the policy update to a specified range. PPO leverages the advantages of trust region optimization to ensure that policy updates do not deviate too far from the previous policy, thus maintaining stability. By limiting the policy update, PPO avoids catastrophic forgetting and provides robust and reliable learning. It also employs importance sampling techniques to make efficient use of past experiences stored in a replay buffer.

The advantages of PPO over SAC are discussed below:

- **Sample Efficiency**: PPO is known for its sample efficiency compared to SAC. It achieves this through the use of multiple epochs of minibatch updates on the collected data, effectively utilizing the data more efficiently and reducing the overall number of samples needed for training.

- **Stable Learning**: PPO employs a surrogate objective function that includes a clipping mechanism, which limits the size of policy updates. This clipping helps to prevent large policy updates that could lead to instability during training. As a result, PPO tends to have more stable learning dynamics compared to SAC.

- **Simplicity**: PPO is relatively easier to implement and tune compared to SAC. It has fewer hyperparameters to tune, and the core algorithm is simpler to understand and implement. This simplicity makes it a popular

choice for practical applications and enables faster experimentation and prototyping.

- **Convergence Guarantees**: PPO offers theoretical guarantees on monotonic policy improvement. It ensures that each iteration of the algorithm improves or maintains the performance of the previous policy. This property provides confidence in the convergence of the algorithm and makes it a reliable choice for reinforcement learning tasks.
- **Compatibility with Discrete Action Spaces**: PPO can handle both continuous and discrete action spaces. While SAC is more suitable for continuous action spaces, PPO's flexibility allows it to handle a wider range of problem domains, including tasks with discrete actions.

The PPO has some limitations as well, and they are discussed below:

- **Exploration**: PPO has a tendency to be conservative in exploration. It relies on the collected data and can be limited in exploring beyond the observed states and actions. This conservative exploration may hinder its ability to discover optimal policies in environments with sparse rewards or complex action spaces.
- **Hyperparameter Sensitivity**: Although PPO has fewer hyperparameters compared to SAC, it can still be sensitive to the choice of hyperparameters, particularly the clipping parameter. Improper tuning of hyperparameters can result in suboptimal performance and slow convergence.
- **Off-Policy Learning**: PPO is an on-policy algorithm, meaning it can only use the data it collected during the current training iteration. It does not leverage off-policy data, unlike SAC. This can limit its ability to take advantage of previously collected experiences and reduce sample efficiency in scenarios where data collection is expensive.
- **Handling Continuous Action Spaces**: While PPO can handle continuous action spaces, it requires additional techniques, such as parameterization of the action distribution, to handle them effectively. In contrast, SAC is inherently designed to handle continuous action spaces more naturally.
- **Limited Exploration in High-Dimensional Action Spaces**: PPO may face challenges in environments with high-dimensional action spaces. It may struggle to explore the vast action space efficiently, leading to suboptimal policies. SAC, with its stochastic policies, can provide better exploration capabilities in such scenarios.

Proximal Policy Optimization (PPO) offers advantages such as sample efficiency, stable learning, simplicity, convergence guarantees, and compatibility with discrete action spaces. However, it may have limitations in terms of exploration, sensitivity to hyperparameters, off-policy learning, handling continuous

action spaces, and limited exploration in high-dimensional action spaces. Understanding these trade-offs is crucial for selecting the appropriate algorithm for reinforcement learning tasks based on the specific requirements of the problem domain.

1.3.3 Deep Deterministic Policy Gradient (DDPG)

DDPG is another popular actor-critic algorithm that is well-suited for continuous action spaces. It extends the ideas of deep Q-networks (DQNs) to policy optimization. DDPG uses an off-policy approach, where a replay buffer is utilized to store experiences for learning. The key component of DDPG is the actor-critic architecture, where the actor network determines the policy by directly mapping states to actions, while the critic network evaluates the quality of the actions chosen by the actor. This separation allows for more stable learning as the critic provides a feedback signal to the actor. DDPG employs the deterministic policy gradient algorithm to update the actor and critic networks, enabling efficient optimization in high-dimensional action spaces.

DDPG has both advantages and disadvantages over SAC and PPO. First we discuss the advantages:

- **Continuous Action Spaces**: DDPG is specifically designed for environments with continuous action spaces, making it well-suited for tasks that require precise control and fine-grained actions. In contrast, PPO and SAC can handle both continuous and discrete action spaces but may require additional techniques for continuous action spaces.
- **Off-Policy Learning**: DDPG is an off-policy algorithm, meaning it can learn from data collected by any policy, including older policies. This allows for more efficient use of collected experience, as it can leverage previously learned behaviors and improve sample efficiency compared to on-policy methods like PPO and SAC.
- **Actor-Critic Architecture**: DDPG employs an actor-critic architecture, where the actor learns the policy and the critic learns the action-value function. This separation of policy and value estimation enables more stable and effective learning. PPO and SAC also use actor-critic architectures, but their specific implementations may differ.
- **Deterministic Policy**: DDPG learns deterministic policies, meaning that for a given state, it always outputs the same action. Deterministic policies can be advantageous in certain scenarios where precise control is required, as they provide more predictable and reliable actions. PPO and SAC, on the other hand, typically use stochastic policies.

Disadvantages of Deep Deterministic Policy Gradient (DDPG) compared to Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC):

- **Exploration**: DDPG can struggle with exploration in environments with sparse rewards or complex action spaces. Its deterministic nature makes it less effective at exploring different actions, potentially leading to suboptimal policies. PPO and SAC, with their stochastic policies, generally have better exploration capabilities.

- **Hyperparameter Sensitivity**: DDPG is sensitive to the choice of hyperparameters, such as the learning rate, exploration noise, and target network update frequency. Improper tuning of these hyperparameters can lead to unstable learning and slow convergence. PPO and SAC, while still requiring hyperparameter tuning, may have more forgiving parameter choices.

- **Sample Efficiency**: DDPG is known to be less sample efficient compared to PPO and SAC. It often requires a large number of samples to learn effective policies, which can be impractical or time-consuming in real-world scenarios. PPO and SAC, with their optimization techniques, tend to be more sample efficient.

- **Convergence**: DDPG may suffer from convergence issues, especially in complex environments or with large neural networks. It can be challenging to find stable and optimal policies, and training can be prone to oscillations or getting stuck in local minima. PPO and SAC, with their specific algorithms, provide more reliable convergence guarantees.

- **Handling Discrete Action Spaces**: DDPG is primarily designed for continuous action spaces and may not handle discrete action spaces efficiently. PPO and SAC, on the other hand, can handle both continuous and discrete action spaces with appropriate modifications.

Deep Deterministic Policy Gradient (DDPG) has advantages such as handling continuous action spaces, off-policy learning, and the actor-critic architecture. However, it may face challenges in exploration, hyperparameter sensitivity, sample efficiency, convergence, and handling discrete action spaces. When selecting an algorithm for a particular reinforcement learning task, it is essential to consider the specific requirements of the problem domain and the trade-offs associated with each algorithm. PPO and SAC provide alternative approaches that may be more suitable in certain scenarios.

2 Behavioral Cloning

Imitation Learning and Behavioral Cloning are two terms often used interchangeably, but they differ slightly in their scope.Imitation Learning is a broader concept which refers to the process of learning a behavior or task from an expert, aiming to mimic the expert's behavior in similar situations. This often involves learning a policy, which is a mapping from states to actions. Imitation

Learning encompasses several techniques, including Behavioral Cloning, Inverse Reinforcement Learning (IRL), Dagger, and others.

Unlike traditional reinforcement learning which aims to find the optimal policy given a known reward function, IRL does the reverse - it aims to derive a reward function given the expert's optimal policy. The key idea is that the expert is acting optimally according to some unknown reward function, and the goal of IRL is to infer this function. Once this reward function is obtained, traditional reinforcement learning can be applied to find the optimal policy.

Dagger is a method designed to overcome one of the main issues in Behavioral Cloning, namely the problem of distributional shift (the agent's policy deviating from the expert's policy, leading the agent into unseen states). In Dagger, initially, the agent's policy is purely based on the expert's actions. Then, the agent begins to make decisions using a mix of its own and the expert's policy. The expert also provides corrections for the agent's decisions. The new data, along with the old data, are used to iteratively retrain the model. Over time, the agent starts relying more on its own policy and less on the expert's policy, as it becomes more competent.

Behavioral Cloning (BC) is a method of teaching a RL Agent to replicate an expert's actions. It falls under the umbrella of Supervised Learning techniques in machine learning.

In the context of reinforcement learning, the goal of Behavioral Cloning is to train an agent to perform specific tasks by imitating the behavior of an expert agent, without explicitly considering the rewards or the long-term consequences of the actions.In Behavioral Cloning the expert agent interacts with the environment, and its actions are recorded along with the corresponding states.The Behavioral Cloning agent is then trained to mimic the expert's behavior. Specifically, it learns a policy that maps from the observed states to actions, using the expert's state-action pairs as training data. This training process usually involves a standard supervised learning algorithm, such as regression or classification, depending on the nature of the actions.The trained agent can then be deployed in the environment, where it should behave similarly to the expert.Behavioral Cloning is typically used when we have access to an expert agent but do not have the ability or the resources to train an agent from scratch using reinforcement learning.

Let's consider a supervised learning setup where we have an expert policy π_{expert}. Given a state 's', the expert policy generates the corresponding action 'a'. Our aim in Behavioral Cloning is to train an agent policy π_agent that, given the same state 's', should generate the same action 'a' as the expert.

Mathematically, we are interested in minimizing the difference between the actions predicted by π_{agent} and π_{expert}. In practice, this is done by minimizing the loss function which measures the difference between the two policies.

If we denote the action space as A and the state space as S, we can express this mathematically. For instance, let's say we're using Mean Squared Error (MSE) as our loss function, we would have:

$$L(\theta) = \mathbb{E}_{s \in S, a \in A} \left[(\pi_{expert(a|s)} - \pi^2_{agent(a|s;\theta)} \right] \quad (1)$$

Where $L(\theta)$ is the loss function which we want to minimize. θ represents the parameters of the agent's policy network. The expectation $\mathbb{E}_{s \sim S, a \sim A}$ is taken over states 's' in state space 'S' and actions 'a' in action space 'A'. $\pi_{expert(a|s)}$ is the action taken by the expert policy given the state 's'. $\pi_{agent(a|s;\theta)}$ is the action taken by the agent's policy given the same state 's'. In the training process, we optimize the parameters θ of the policy network to minimize this loss function, using techniques like stochastic gradient descent.

The actual form of the loss function might vary depending on the specific problem and the action space. For instance, a cross-entropy loss might be used if the action space is discrete. The overall idea remains the same: to make the agent's policy as close to the expert's policy as possible.

3 Datasets

This project uses three environments of varying complexity to test the theories, namely: Pendulum, a Continuous version of Lunar Lander and Car Racing.

3.0.1 Pendulum

The Pendulum environment in OpenAI Gym simulates a pendulum swinging back and forth. The task is to control the pendulum by applying torques to the joint, such that it remains upright as long as possible. The observation consists of the current angle and angular velocity of the pendulum, and the action space allows the agent to apply continuous torques within a specified range. The goal is to minimize the energy used to keep the pendulum balanced, making it a classic control problem. Reinforcement learning algorithms can be used to learn optimal control policies for stabilizing the pendulum.

3.0.2 Lunar Lander

The Lunar Lander environment simulates a lunar lander attempting to land on the moon's surface. The task is to control the lander's thrust and orientation to safely land it on a landing pad. The observation space includes the lander's position, velocity, angle, and angular velocity, while the action space consists of discrete actions to control the main engine, left engine, or right engine. The goal is to land the lander as softly as possible while conserving fuel. This environment presents challenges in balancing exploration and exploitation, as well as mastering the control of a complex vehicle.

3.0.3 Car Racing

The Car Racing environment is a 2D driving simulator that allows agents to learn how to control a car and navigate a race track. The goal is to complete

the track as quickly as possible while staying on the road. The observation space includes a top-down view of the track and the car's position, velocity, and angles, while the action space consists of continuous actions representing steering, acceleration, and braking. This environment requires learning complex driving skills, such as cornering, overtaking, and maintaining control at high speeds. It is commonly used for training autonomous driving agents and testing various control algorithms.

These OpenAI Gym environments provide diverse challenges for reinforcement learning agents, ranging from simple control tasks (Pendulum) to more complex navigation and control tasks (Lunar Lander and Car Racing). They serve as benchmark environments to develop and evaluate reinforcement learning algorithms and control policies.

4 Results

In general, a complete Behavioral Cloning training pipeline would consist of the following steps:

- Creating Dataset of expert demonstrations
- Training a Behavioral Cloning Agent from those expert demonstrations and recording the loss
- use our trained Behavioral Cloning agent to interact with our environment, to observe it's policy through rewards.

The Python Libraries used include numpy, matplotlib and TensorFlow Agents. TensorFlow Agents (TF-Agents) is a library developed by the TensorFlow team that provides a collection of reusable components and tools for building reinforcement learning (RL) agents. It aims to simplify the development and implementation of RL algorithms using TensorFlow.TF-Agents offers a variety of state-of-the-art RL algorithms, including deep Q-networks (DQN), Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), and more. These algorithms are implemented using TensorFlow, allowing for efficient computation on both CPUs and GPUs.The library follows a modular design, providing separate components for agents, networks, policies, replay buffers, and more. This modular architecture allows for easy customization and extension of different components to adapt them to specific RL problems. TF-Agents seamlessly integrates with TensorFlow, leveraging its computational graph and automatic differentiation capabilities. This integration enables efficient training and optimization of RL agents using TensorFlow's powerful ecosystem.It provides a flexible framework that allows users to define their custom RL architectures, loss functions, exploration strategies, and more. It also supports easy configuration through the use of Gin, a configuration framework for TensorFlow.

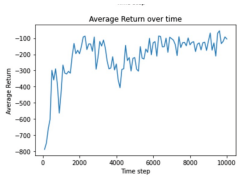
As assumed, letting a pre-trained agent interact with an environment should generate greater rewards than that of an untrained agent.

For this project, each of the three environments are acted upon by agents, whose policies are optimised using the three algorithms mentioned above. For example: an agent is allowed to optimize it's policy using Soft Actor Critic and test it on the Pendulum environment.Similarly PPO and DDPG are tested. Of the three algorithms, the best performing one is used to generate expert demonstrations for the Behavioral Cloning Agent. Once the behavioral cloning agent learns from expert demonstrations , it essentially learns to imitate the expert's behavior. This trained agent then uses DDPG to search for a policy better than the one it imitates. Finally we verify weather the behavioraly cloned agent performs better on the DDPG algorithm than a non-behavioraly cloned agent.

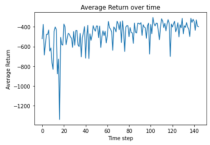
This training pipeline is then tested on two more environments, Lunar Lander and Racing Car. the latter is more complex than the former, therefore the agent's policy takes longer to converge to an optimal policy in case of Racing Car than in case of Lunar Lander.In general reinforcement learning algorithms need a lot of training to converge, but a general trend of increasing rewards is observed in case of all the three environments.

The performance of the algorithm is judged by tracking the average reward every few intervals and plotting it over the entire training period. Ordinarily for any kind of supervised learning, the plot of loss gives us a good idea of the model's performance. However, in Reinforcement Learning, there is inherrent randomness in a stochastic system, and event the agent's actions are not deterministic. Therefore, until policy convergence happens, loss fluctuations are rather common. Therefore, we use loss as proof of the agent's exploration rather than as a parameter to judge it's learning ability.

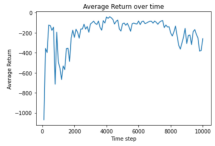
In case of Behavioral Cloning the learning from expert demonstrations is supervised, therefore a converging loss function is indicative of the Behavioral Cloning agent's cloning capabilities.



(a) Mean rewards for SAC



(b) Mean rewards for PPO

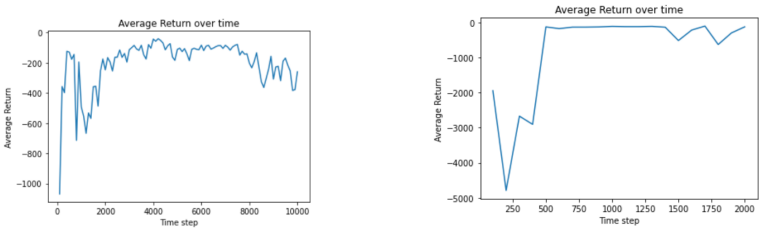


(c) Mean rewards for DDPG

Figure 1: Performance of SAC,PPO and DDPG algorithms on Lunar Lander

In case of Lunar Lander, where the plots of average rewards, is indicative of policy convergence, it is observed that DDPG performs better than PPO and SAC, in it that DDPG rewards fluctuate less and stay stabilized for longer time.

This is the expert demonstrations for the Behavioral Cloning agent.Using these demonstrations, the behavioral cloning learns to imitate the DDPG agent. As the Behavioral Cloning agent now knows the DDPG agent's policy, further optimizing this policy should generate better rewards.



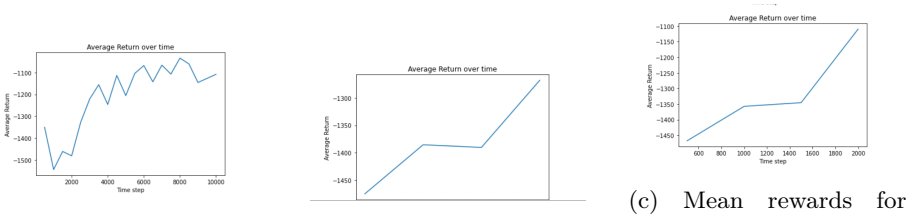
(a) Mean rewards for Vanilla agent trained using DDPG

(b) Mean rewards for Behaviorally Cloned agent on DDPG

Figure 2: Vanilla Agent vs Behaviorally Cloned agent on Lunar Lander environment.

Just as observed, with behavioral cloning, the agent learns a good enough policy, such that optimizing it helps keep rewards steadily at small negative values over majority of 2000 steps. Further training should generate positive rewards.

Similar observations are seen in Pendulum environment. While the range of rewards is higher in case of Pendulum than Lunar Lander, convergence is still observed, even within 2000 iterations.



(a) Mean rewards for SAC

(b) Mean rewards for PPO

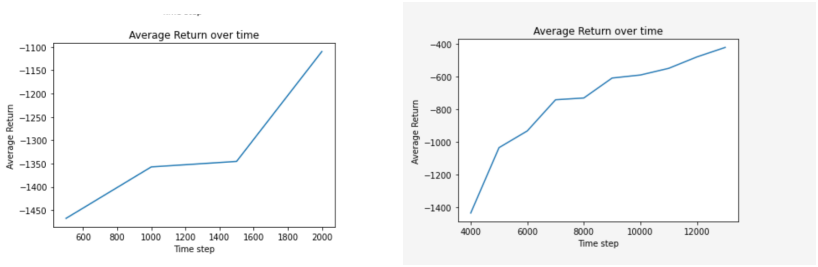
(c) Mean rewards for DDPG

Figure 3: Performance of SAC,PPO and DDPG algorithms on pendulum

The DDPG algorithm performs best in case of Lunar Lander, and serves as expert demonstrations for Behavioral Cloning. Comparing Vanilla and Behaviorally Cloned agents, it is observed that Behavioral Cloning does indeed improve rewards.

Finally , the training pipeline is tested on Car Racing environment. The three algorithms generate quite stable results in Car Racing, but DDPG was chosen for uniformity. The Behaviorally cloned DDPG agent showed only positive growth in rewards throughout the 2000 iterations, and should converge to an optimal value with more training.

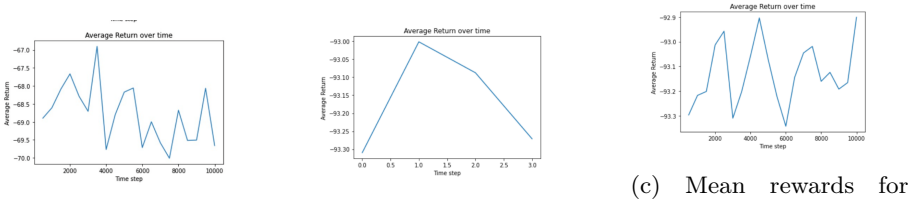
if the Loss values are to be compared we will see that the loss value stabilizes with training only in Racing Car Environment, even though the mean rewards were seen to improve in all the three environments. This warrants for better parameters to compare performance



(a) Mean rewards for Vanilla agent trained using DDPG

(b) Mean rewards for Behaviorally Cloned agent on DDPG

Figure 4: Vanilla Agent vs Behaviorally Cloned agent on pendulum environment.

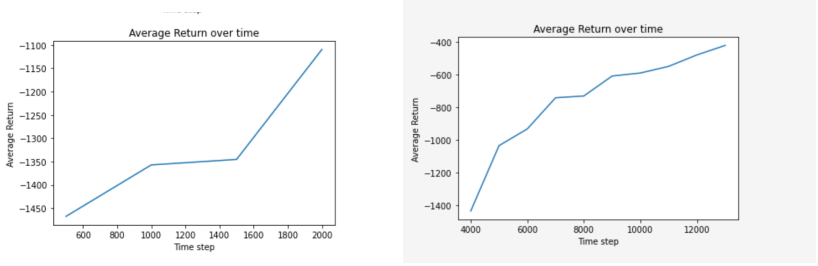


(a) Mean rewards for SAC

(b) Mean rewards for PPO

(c) Mean rewards for DDPG

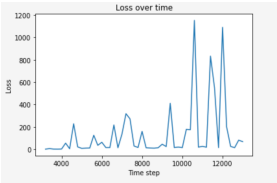
Figure 5: Performance of SAC,PPO and DDPG algorithms on pendulum



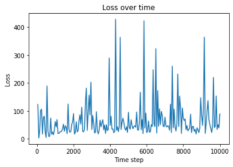
(a) Mean rewards for Vanilla agent trained using DDPG

(b) Mean rewards for Behaviorally Cloned agent on DDPG

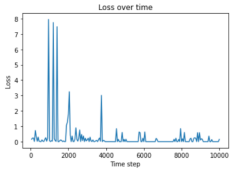
Figure 6: Vanilla Agent vs Behaviorally Cloned agent on pendulum environment.



(a) DDPG Loss for Pendulum



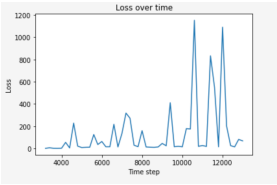
(b) DDPG Loss for Lunar Lander



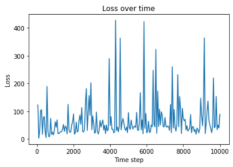
(c) DDPG Loss for Car Racing

Figure 7: DDPG Losses over the three environments

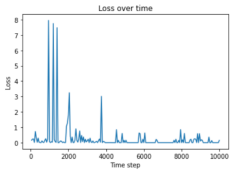
Finally we compare the Behavioral Cloning loss for all the three environments. Since behavioral cloning is a supervised learning approach, a loss function is indicative of good learning ability.



(a) DDPG Loss for Pendulum



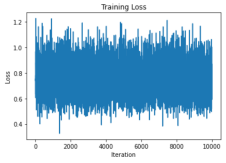
(b) DDPG Loss for Lunar Lander



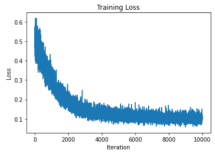
(c) DDPG Loss for Car Racing

Figure 8: DDPG Losses over the three environments

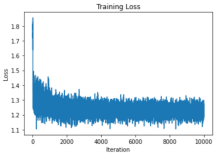
The Learning loss for Pendulum is too varied and does not provide much information on the model's learning ability, but a steady decreasing loss in case of Lunar Lander and Car Racing environment is a promising indication of the model's learning ability. Pendulum's loss may be varied because the expert demonstrations provided aren't good enough to begin with, which underlines the supervised nature of the model.



(a) BC Loss for Pendulum



(b) BC Loss for Lunar Lander



(c) BC Loss for Car Racing

Figure 9: BC Losses over the three environments

5 Summary

A training pipeline was developed using Deep Deterministic Policy Gradient (DDPG) as the expert model for behavioral cloning in reinforcement learning applications. While DDPG may not always be the most optimal choice, its uniformity simplified the process of policy optimization. This method proved effective in high-certainty environments like Lunar Lander and Car Racing, showing a noticeable improvement in the policy optimization process. In future work, the applicability of this DDPG-based behavioral cloning pipeline could be extended to areas like robotic perception, autonomous driving, and other robotic learning environments.

Behavioral Cloning (BC) provides several compelling advantages when implemented in robotic perception, autonomous driving, and other robotic learning environments. Notably, BC significantly reduces training time by leveraging expert demonstrations, which can expedite the learning of a satisfactory policy as compared to standard reinforcement learning methods. BC also promotes safety, especially crucial in high-risk environments like autonomous driving, by mimicking an expert's policy, thereby minimizing harmful actions during the training process. Additionally, BC alleviates the need for costly or risky exploration since learning is derived directly from expert demonstrations. This approach tends to be highly sample-efficient, optimizing the use of given data to learn effective policies swiftly. Moreover, BC enables knowledge transfer from human experts to the machine learning system, particularly beneficial in complex tasks like robotic perception where human expertise can enhance learning efficiency. Finally, BC can better manage tasks that are challenging to learn through traditional reinforcement learning due to sparse rewards, as it can learn from the expert's demonstrations without requiring a dense reward signal. However, it is worth noting that the effectiveness of a BC-trained policy heavily relies on the quality of the expert demonstrations provided. Any poor or unrepresentative demonstrations may negatively impact the performance of the BC agent.

References

1. CSE-676 Project (Spring23) at `https://github.com/KC-decoder/CSE-676-Final-Project`
2. `https://github.com/tensorflow/agents`
3. Pomerleau, D.A., 1991. Efficient training of artificial neural networks for autonomous navigation. Neural Computation, 3(1), pp.88-97
4. Russell, S., 1998, July. Learning agents for uncertain environments. In Proceedings of the eleventh annual conference on Computational learning theory (pp. 101-103). ACM
5. `https://github.com/openai/gym`

6. https://gist.github.com/Kenneth-Schroeder/6dbdd4e165331164e0d9dcc2355698e2

7. https://www.gymlibrary.dev/

8. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D. (2015). Continuous control with deep reinforcement learning. In International Conference on Learning Representations (ICLR).

9. https://docs.ray.io/en/latest/rllib/key-concepts.html

10. https://docs.ray.io/en/latest/rllib/rllib-algorithms.html#maddpg