VIETNAM NATIONAL UNIVERSITY

HỒ CHÍ MINH UNIVERSITY OF TECHNOLOGY

ELECTRICAL – ELECTRONIC ENGINEERING DEPARTMENT

CAPSTONE PROJECT 2

TOPIC:
REINFORCEMENT LEARNING

LECTURER: Dr. PHAM VIET CUONG

NAME: TRAN HOANG TUAN

STUDENT ID: 1951224

HO CHI MINH CITY, MAY YEAR 2023

ACKNOWLEDGEMENT

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them. I would like to express my special gratitude and thanks to industry persons for giving me such attention and time.

My thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.

Ho Chi Minh City, 26tth May, 2023.

Student

Trần Hoàng Tuấn

ABSTRACT

Reinforcement learning, a subfield of machine learning, offers a powerful framework for training intelligent agents to make optimal decisions in dynamic and uncertain environments. This project explores the application of reinforcement learning in a specific domain, aiming to develop an agent that can learn to navigate and solve complex tasks through interaction with the environment.

The project focuses on implementing and evaluating various reinforcement learning algorithms, such as Q-learning, Deep Q-Networks (DQN), and policy gradient methods. These algorithms are applied to a chosen environment or task, allowing the agent to learn from experience and improve its decision-making capabilities over time.

Key components of the project include defining the problem and environment, designing appropriate reward structures, selecting and implementing reinforcement learning algorithms, and fine-tuning the agent's performance through iterative experimentation.

Contents

1. INTRODUCTION

1.1 Overview

Reinforcement learning is a branch of machine learning that focuses on enabling agents or systems to learn and make decisions in an interactive environment. It is inspired by the way humans and animals learn from trial and error to maximize rewards and minimize negative outcomes. In reinforcement learning, an agent learns by interacting with an environment, receiving feedback in the form of rewards or penalties, and adjusting its actions to maximize long-term cumulative rewards.

The fundamental concept in reinforcement learning is the Markov decision process (MDP), which formalizes the interaction between an agent and its environment. The environment is represented as a set of states, and the agent takes actions to transition between states. At each state, the agent receives feedback in the form of a reward signal, which indicates the desirability of the current state-action pair. The agent's goal is to learn a policy—a mapping from states to actions—that maximizes the expected cumulative reward over time.

1.2 Project objective

The project aims to achieve two main objectives: first, to understand the fundamental principles and concepts of reinforcement learning, including value estimation, policy optimization, and exploration-exploitation trade-offs. Second, to develop a trained agent that demonstrates competence in solving the defined task, achieving high performance and efficiency.

2. THEORY

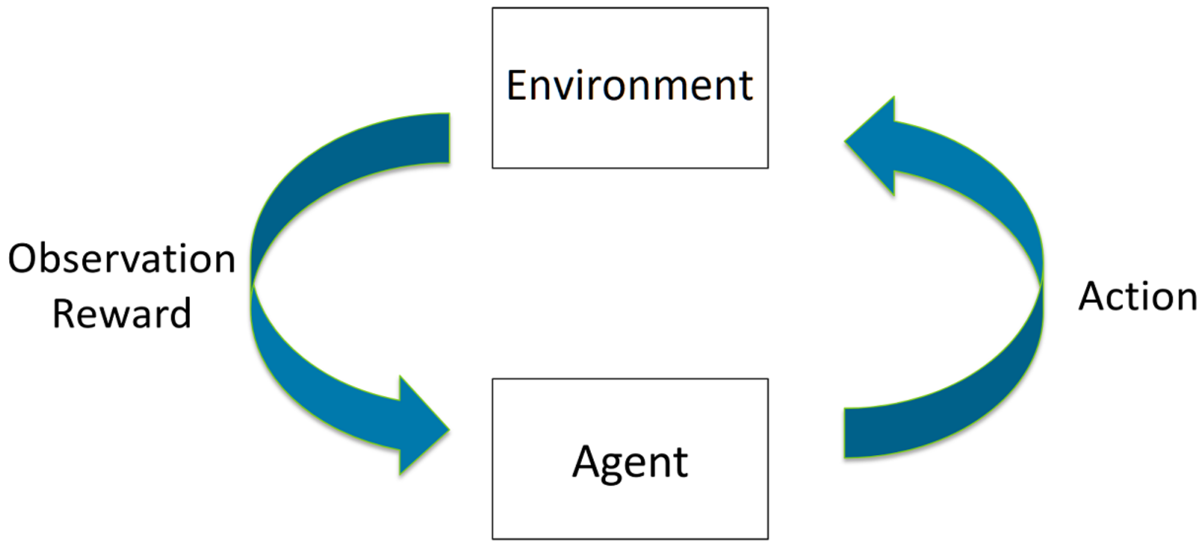
2.1 COMPONENT OF REINFORCEMENT LEARNING

The key components of reinforcement learning include:

1. Agent: The agent is the learner or decision-making entity in the reinforcement learning framework. It interacts with the environment, observes states, takes

actions, and receives feedback in the form of rewards or penalties. The agent's objective is to maximize the cumulative rewards it obtains over time.

2. Environment: The environment represents the external system or world in which the agent operates. It provides the agent with information about the current state, accepts actions from the agent, and generates the next state based on the chosen action. The environment also assigns rewards to the agent based on its actions and the resulting state transitions. It can be as simple as a simulated game environment or as complex as a real-world scenario.
3. State: A state represents the current configuration or situation of the environment. It encapsulates all relevant information necessary for decision-making. The agent's actions are typically chosen based on the current state. In some cases, the environment can be fully observable, meaning the agent has complete information about the state. In other cases, the environment may be partially observable, requiring the agent to infer the state based on available observations.
4. Action: Actions are the choices available to the agent at each state. The agent selects an action based on its policy, which is the decision-making strategy it employs. Actions can have immediate effects on the environment, leading to state transitions and subsequent rewards.
5. Value function: The value function is a crucial component in reinforcement learning. It estimates the expected cumulative reward the agent can obtain from a particular state or state-action pair. There are two main types of value functions: the state-value function (V-function) estimates the value of being in a particular state, and the action-value function (Q-function) estimates the value of taking a specific action in a given state. The value function guides the agent in evaluating the desirability of different states or actions and helps in making informed decisions.
6. Exploration vs. Exploitation: Exploration is the process of trying out different actions to learn about the environment. Exploitation is the process of using the knowledge gained from exploration to take the best action in each state.



Overall, there are four fundamental concepts that underpin most Reinforcement Learning projects are:

- Agent: the actor operating within the environment, it is usually governed by a policy.
- Environment: the place or world for agent can operate in.
- Action: the agent can do something within the environment known as an action.
- Reward and observations: in return the agent receives a reward and a view of what the environment looks like after acting on it.

2.2 OpenAI Gym

OpenAI Gym is an open-source Python library that provides a standardized interface for developing and evaluating reinforcement learning algorithms. It offers a collection of pre-defined environments, including classic control tasks, Atari games, robotics simulations, and more. OpenAI Gym serves as a powerful tool for researchers, developers, and students to experiment with and benchmark their reinforcement learning algorithms.

Some key feature and components in OpenAI:

- Environments: OpenAI Gym provides a wide range of environments, each representing a specific task or problem that an agent can interact with. These environments are designed to simulate real-world scenarios, game environments, or other simulated domains. Examples include CartPole, MountainCar, LunarLander, Pong, ... and many more.

- Standardized Interface: Gym offers a unified interface for interacting with environments. This interface consists of a set of methods that allow agents to interact with the environment. For instance:

```
reset(): reset the environment and obtain initial
observation

render(): visualize the environment

step():apply an action to environment

close(): close down the render frame
```

- Observation Space: The observation space defines the format and range of the information that an agent can observe from the environment. It can be discrete, continuous, or even more complex, like images or text. Gym provides various types of observation spaces:

- Box: n dimensional tensor, range of values

  Ex: Box(0,1,shape = (3,3))
- Discrete: set of items

  Ex: Discrete(3) ➔ value are 0 , 1 , 2
- Tuple: combine space together

  Ex: Tuple((Discrete(2), Box(0, 100, shape=(1,)))) ➔ Box or Discrete
- Dict: dictionary of spaces

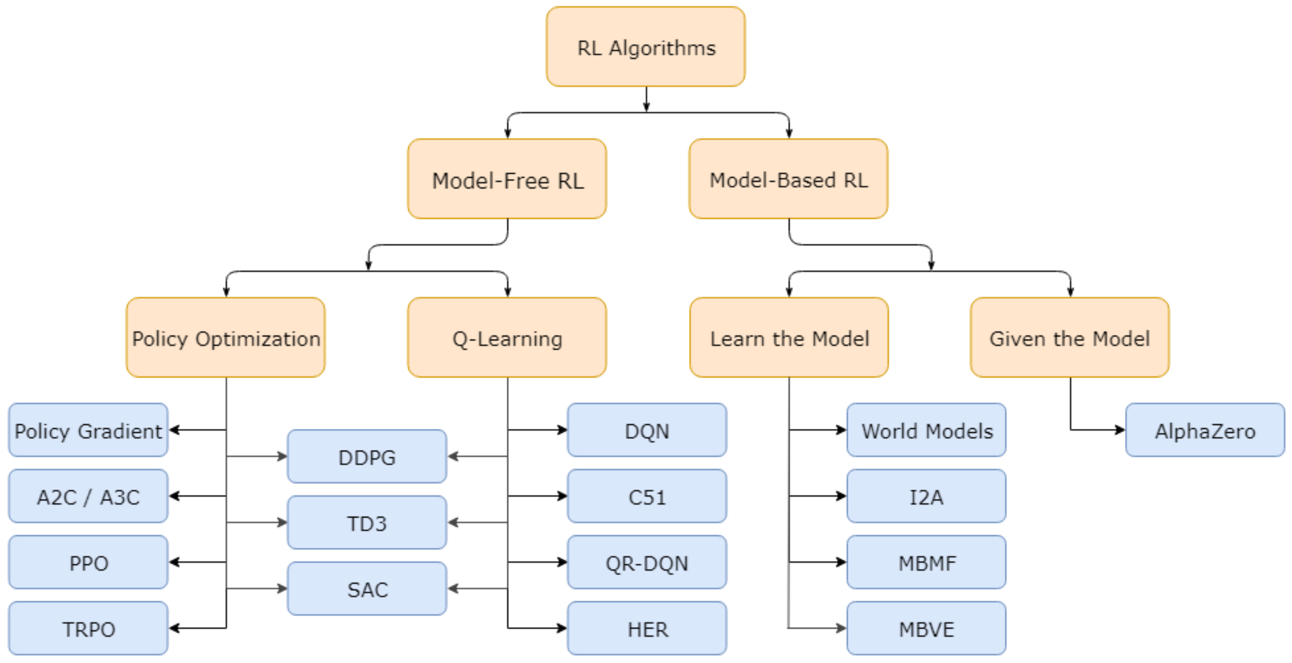
  Ex: Dict((‘height’:Discrete(2), “speed”:Box(0, 100, shape=(1,))))
- MultiBinary: one hot encoded binary values

  Ex: MultiBinary(4) ➔ list of values that have 4 positions (0, 1, 2, 3)
- MultiDiscrete: multiple discrete values

  Ex: MultiDiscrete([5, 2, 2]) ➔ a range of value between 0 and 4; for the first position is 0 and 1, for the second position is 0 and 1

- Action Space: The action space defines the set of possible actions that an agent can take in the environment. It can be discrete, where the agent selects actions from a fixed set of options, or continuous, where the agent chooses actions from a continuous range. Gym supports Discrete and Box action spaces, enabling a wide range of action selection methods.
- Rewards: Gym environments use rewards to provide feedback to the agent. The agent's objective is to maximize the cumulative rewards it receives over time. Rewards can be positive, negative, or zero, and they reflect the desirability of the agent's actions and progress towards its goals.
- Wrapper System: Gym incorporates a flexible wrapper system that allows users to modify the behavior of environments or add additional functionality. Wrappers can be used for preprocessing observations, modifying rewards, applying transformations to actions, or implementing custom logic.
- Monitoring and Evaluation: OpenAI Gym provides tools for monitoring and evaluating the performance of reinforcement learning algorithms. It supports logging of episodes, recording videos of agent-environment interactions, and computing performance metrics such as episode rewards and average episode lengths.

2.3 ALGORITHMS



2.3.1 Model – Free vs Model – Based RL

One of the most important branching points in an RL algorithm is the question of whether the agent has access to (or learns) a model of the environment. By a model of the environment, I mean a function which predicts state transitions and rewards.

The main upside to having a model is that it allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy.

The main downside is that a ground-truth model of the environment is usually not available to the agent. If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally (or super terribly) in the real environment. Model-learning is fundamentally hard, so even intense effort—being willing to throw lots of time and compute at it—can fail to pay off.

Algorithms which use a model are called model-based methods, and those that don't are called model-free. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune.

There are two main approaches to representing and training agents with model-free Reinforcement Learning:

2.3.2 Policy Optimization

Methods in this family represent a policy explicitly as π_θ(a|s). They optimize the parameters θ either directly by gradient ascent on the performance objective J(π_θ), or indirectly, by maximizing local approximations of J(π_θ). This optimization is almost always performed on-policy, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator V_ϕ(s) for the on-policy value function V^π(s), which gets used in figuring out how to update the policy.

A couple of examples of policy optimization methods are:

- A2C / A3C (Asynchronous Advantage Actor-Critic), which performs gradient ascent to directly maximize performance
- PPO (Proximal Policy Optimization), whose updates indirectly maximize performance, by instead maximizing a surrogate objective function which gives a conservative estimate for how much J(π_θ) will change as a result of the update.

2.3.3 Q-Learning

Methods in this family learn an approximator Q_θ(s,a) for the optimal action-value function, Q*(s,a). Typically, they use an objective function based on the Bellman equation. This optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q* and π*: the actions taken by the Q-learning agent are given by

$$a(s) = argmax Q_\theta(s, a)$$

Examples of Q-learning methods include:

- DQN (Deep Q-Networks), a classic which substantially launched the field of deep RL,
- C51 (Categorical 51-Atom DQN), a variant that learns a distribution over return whose expectation is Q*.

2.3.4 Trade-offs Between Policy Optimization and Q-Learning

The primary strength of policy optimization methods is that they are principled, in the sense that you directly optimize for the thing you want. This tends to make them stable and reliable. By contrast, Q-learning methods only indirectly optimize for agent performance, by training Q_θ to satisfy a self-consistency equation. There are many failure modes for this kind of learning, so it tends to be less stable. But, Q-learning methods gain the advantage of being substantially more sample efficient when they do work, because they can reuse data more effectively than policy optimization techniques.

2.3.5 Interpolating Between Policy Optimization and Q-Learning

Serendipitously, policy optimization and Q-learning are not incompatible (and under some circumstances, it turns out, equivalent), and there exist a range of algorithms that live in between the two extremes. Algorithms that live on this spectrum are able to carefully trade-off between the strengths and weaknesses of either side. Examples include:

- DDPG (Deep Deterministic Policy Gradient), an algorithm which concurrently learns a deterministic policy and a Q-function by using each to improve the other,

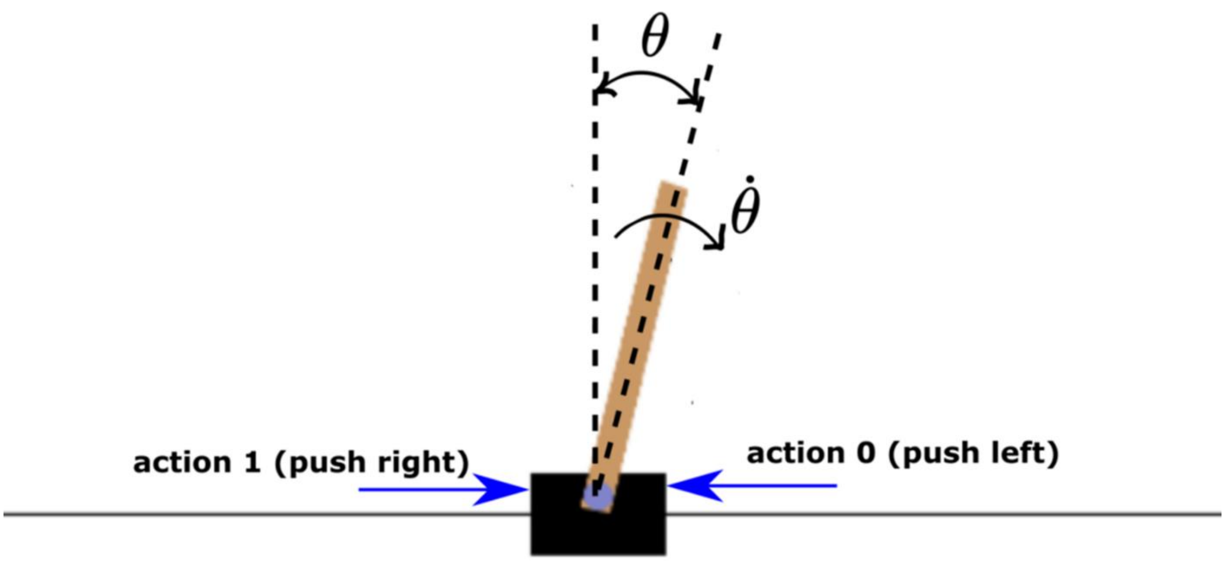
- SAC (Soft Actor-Critic), a variant which uses stochastic policies, entropy regularization, and a few other tricks to stabilize learning and score higher than DDPG on standard benchmarks.

3. SOFTWARE DESIGN

3.1 Cart Pole

In this project, I explore the fascinating world of the Cart Pole environment in reinforcement learning. The task involves balancing a pole on a cart by controlling its movements. I aim to train an intelligent agent that can effectively balance the pole for as long as possible using various reinforcement learning algorithms. Through this project, I will delve into key concepts such as policy learning, value estimation, and exploration-exploitation trade-offs. By mastering the Cart Pole task, I showcase the power of reinforcement learning in solving complex control problems. Whether you're a beginner or an experienced practitioner, this project offers valuable insights and hands-on experience in the field of reinforcement learning.

3.1.1 Parameter



Action space

The action is a n array with shape (1) which can take values {0, 1} indicating the direction of the fixed force the cart is pushed with. The velocity that is reduced or increased by the applied force is not fixed and it depends on the angle the pole is pointing. The center of gravity of the pole varies the amount of energy needed to move the cart underneath it

Number	Action
0	Push cart to the left
1	Push cart to the right

Observation Space

The observation is a n array with shape (4,) with the values corresponding to the following positions and velocities:

Number	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

While the ranges above denote the possible values for observation space of each element, it is not reflective of the allowed values of the state space in an unterminated episode. Particularly:

- The cart x-position (index 0) can be taken values between -4.8, 4.8, but the episode terminates if the cart leaves the -2.4, 2.4 range.
- The pole angle can be observed between 0.418, 0.418 radians or ±24°, but the episode terminates if the pole angle is not in the range 0.2095, 0.2095 (or ±12°)

Rewards

Since the goal is to keep the pole upright for as long as possible, a reward of +1 for every step taken, including the termination step, is allotted.

Starting State

All observations are assigned a uniformly random value in (-0.05, 0.05).

Episode End

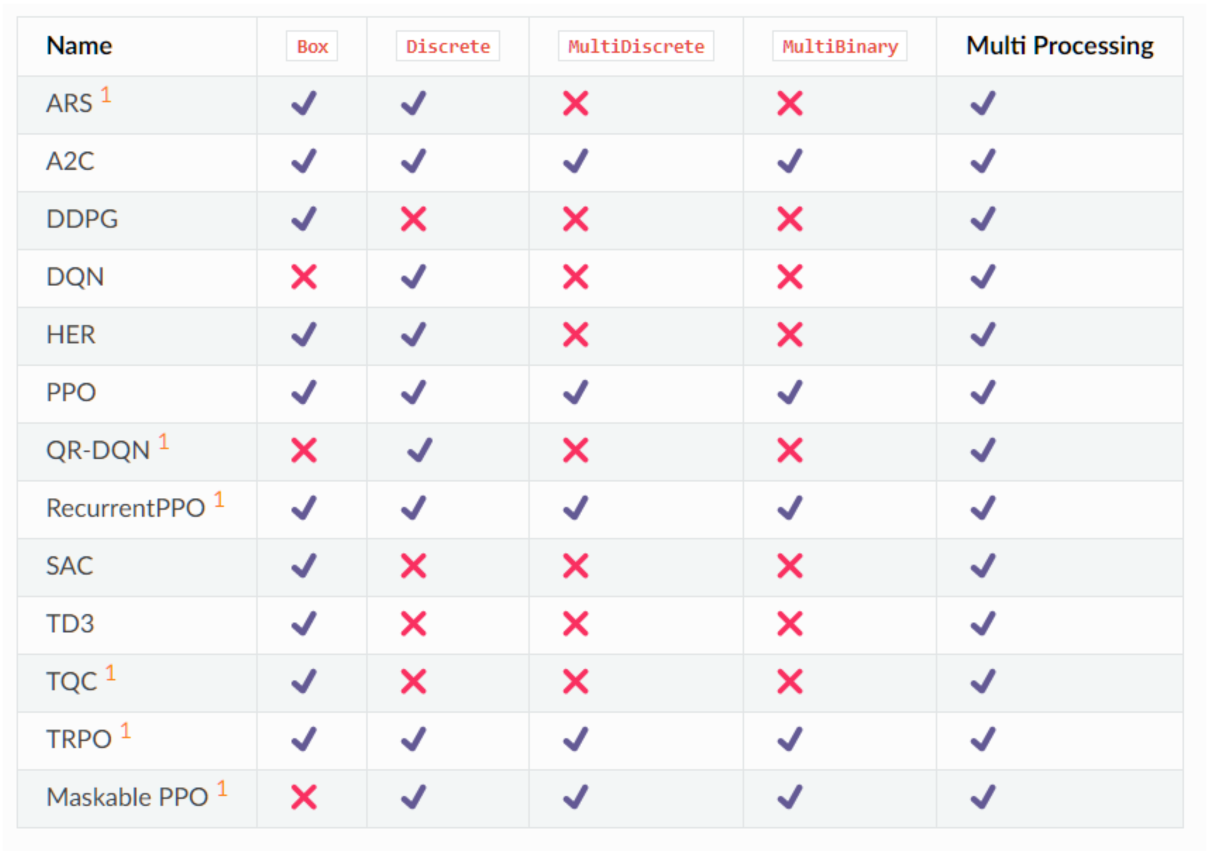
The episode ends if any one of the following occurs:

1. Termination: Pole Angle is greater than ±12°
2. Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than 500 (200 for v0); this project I use v0

3.1.2 Policy Optimization

Stable Baselines3

This table displays the reinforcement learning algorithms that are implemented in this project using Stable Baselines3, along with some useful characteristics: support for discrete/continuous actions, multiprocessing.



Name	Box	Discrete	MultiDiscrete	MultiBinary	Multi Processing
ARS [1]	✓	✓	✗	✗	✓
A2C	✓	✓	✓	✓	✓
DDPG	✓	✗	✗	✗	✓
DQN	✗	✓	✗	✗	✓
HER	✓	✓	✗	✗	✓
PPO	✓	✓	✓	✓	✓
QR-DQN [1]	✗	✓	✗	✗	✓
RecurrentPPO [1]	✓	✓	✓	✓	✓
SAC	✓	✗	✗	✗	✓
TD3	✓	✗	✗	✗	✓
TQC [1]	✓	✗	✗	✗	✓
TRPO [1]	✓	✓	✓	✓	✓
Maskable PPO [1]	✗	✓	✓	✓	✓

Each algorithm supports different of observation spaces.

According to the table given, I should check the observation space of the environment to choose the best suitable algorithms:

```
env.observation_space
```

This will return the output:

```
Box ([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -
3.4028235e+38], [4.8000002e+00 3.4028235e+38
4.1887903e-01 3.4028235e+38], (4,), float32)
```

Each parameter corresponds to the observation of the environment with the type "Box". Moreover, the action space is also a "Discrete" type:

```
env.action_space
```

➔ `Discrete(2)`

In this case, I will use the PPO algorithms.

PPO:

The Proximal Policy Optimization algorithm combines ideas from A2C (having multiple workers) and TRPO (it uses a trust region to improve the actor).

PPO is motivated by the same question as TRPO: how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse? Where TRPO tries to solve this problem with a complex second-order method, PPO is a family of first-order methods that use a few other tricks to keep new policies close to old. PPO methods are significantly simpler to implement, and empirically seem to perform at least as well as TRPO.

There are two primary variants of PPO: PPO-Penalty and PPO-Clip.

- PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.
- PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

Key equations:

PPO-clip updates policies via

$$\theta_{k+1} = argmax\ E[L(s, a, \theta_k, \theta)]$$

typically taking multiple steps of (usually minibatch) to maximize the objective. Here L is given by

$$L(s, a, \theta_k, \theta) = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), clip(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon) A^{\pi_{\theta_k}}(s, a))$$

in which ϵ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old.

This is a pretty complex expression, there's a considerably simplified version of this objective which is a bit easier to grapple with:

$$L(s, a, \theta_k, \theta) = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), g(\epsilon\ , A^{\pi_{\theta_k}}(s, a)))$$

where

$$g(\epsilon\ , \mathrm{A}) = \begin{cases} (1 + \epsilon\ )\mathrm{A} & A \geq 0 \\ (1 + \epsilon\ )A & A \leq 0 \end{cases}$$

Advantage is positive: Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to

$$L(s, a, \theta_k, \theta) = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1 + \epsilon\ )) A^{\pi_{\theta_k}}(s, a)$$

Because the advantage is positive, the objective will increase if the action becomes more likely—that is, if $\pi_\theta(a|s)$ increase, but the min in this term puts a limit to how much the objective can increase. Once $\pi_\theta(a|s) > (1 + \epsilon\ )\pi_{\theta_k}(a|s)$, the min kicks in and this term hits a ceiling of $(1 + \epsilon\ )A^{\pi_{\theta_k}}(s, a)$.

Advantage is negative: Suppose the advantage for that state-action pair is negative, in which case its contribution to the objective reduces to

$$L(s, a, \theta_k, \theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1 - \epsilon\,)\right)A^{\pi_{\theta_k}}(s, a)$$

Because the advantage is negative, the objective will increase if the action becomes less likely—that is, if $\pi_\theta(a|s)$ decreases. But the max in this term puts a limit to how much the objective can increase. Once $\pi_\theta(a|s) < (1 - \epsilon\,)\pi_{\theta_k}(a|s)$. The max kicks in and this term hits a ceiling of $(1 - \epsilon\,)A^{\pi_{\theta_k}}(s, a)$.

3.1.3 Deep Q learning Network

Similarly with using Policy; it also has agent, environment, state, action, reward.

The main neural network tries to predict the expected return of taking each action for the given state. Training and updating are every episode. The target neurol network is used to get the target value for calculating the loss and optimizing it. This neurol network will be updated every N timesteps with the weights of the main networks.

Replay buffer is a list that is filled with the experiences lived by the agent. An experience is represented by the current state, the action taken in the current state, the reward obtained after taking action, whether it is a terminal state or not, and the next state reached after taking action.

Some others parameters are state size, action size, gama, episode, number of steps, epsilon value, epsilon decay, learning rate, target NN update rate.

The algorithms are a bit different from Policy due to Q function:

Source: https://en.wikipedia.org/wiki/Q-learning

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}\right)}^{\text{learned value}}$$

Where r_t is the reward received when moving from the state s_t to the state s_{t+1} and $\propto$ is learning rate ($0 < \propto < 1$)

Note that this is the sum of three factors:

- $(1-\propto)Q(s_t, a_t)$: the current value (weighted by one minus the learning rate)
- $\propto r_t$: the reward $r_t = r(s_t, a_t)$ to obtain if action a_t is taken when in state s_t
- $\propto \gamma \max Q(s_{t+1}, a)$:the maximum reward that can be obtained form state s_{t+1}

3.1.4 Comparing Policy and DQN

Training by policy and Deep Q-learning (DQN) are two popular approaches in reinforcement learning, each with its own characteristics and benefits.

Training by Policy:

- Objective: The main goal in training by policy is to directly learn the optimal policy, which maps states to actions. The agent aims to maximize the expected cumulative reward by improving the policy through iterative updates.
- Policy Representation: Training by policy typically uses a parameterized policy representation, such as a neural network, that directly outputs actions based on observed states.
- Exploration-Exploitation: Training by policy requires a trade-off between exploration and exploitation. Techniques like epsilon-greedy exploration or adding noise to the policy can be employed to balance exploration of new actions and exploitation of the learned policy.
- Convergence: Training by policy can be challenging in terms of convergence. The learning process can be sensitive to initialization, and the optimization of the policy parameters can be prone to local optima.
- Continuous Action Spaces: Training by policy can naturally handle continuous action spaces by using policy parameterization methods like Gaussian policies or deterministic policies with continuous outputs.

Deep Q-Learning (DQN):

- Objective: The main objective in Deep Q-learning is to learn the optimal action-value function (Q-function), which estimates the expected cumulative reward for taking a specific action in a given state. The agent learns Q-values for state-action pairs and uses them to select actions.

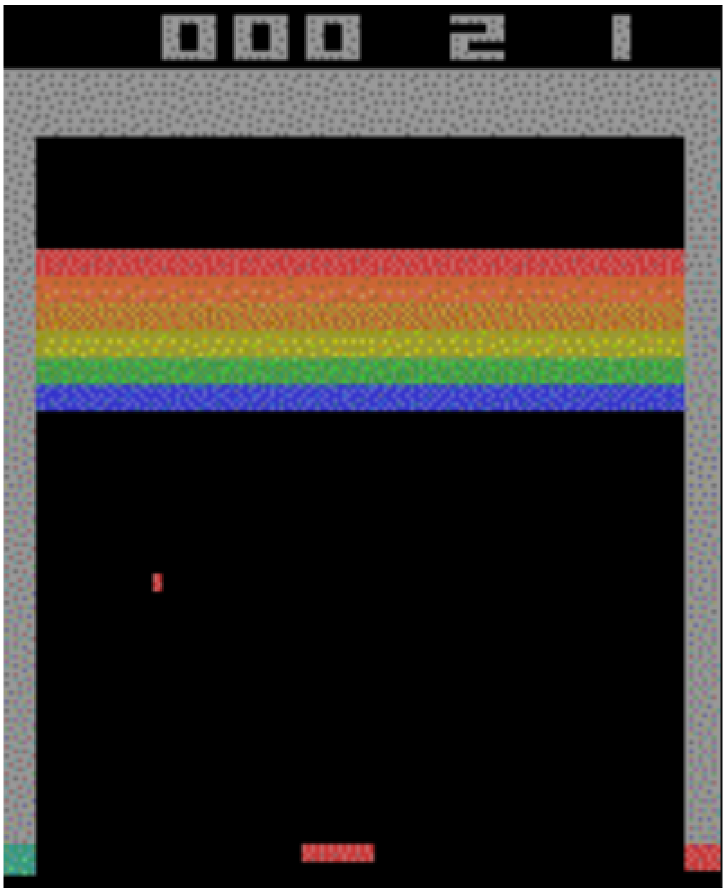
- Value Iteration: Deep Q-learning employs value iteration, which involves iteratively estimating and updating the Q-values using the Bellman equation. The agent learns Q-values by interacting with the environment and performs updates to improve the estimation of Q-values.
- Exploration-Exploitation: Deep Q-learning typically uses an exploration strategy, such as epsilon-greedy, to balance exploration and exploitation. Initially, the agent explores the environment by taking random or suboptimal actions and gradually shifts towards exploiting the learned Q-values.
- Convergence and Stability: Deep Q-learning algorithms are generally more stable and well-established. They have convergence guarantees under certain conditions and employ techniques like experience replay and target networks to stabilize the learning process.
- Discrete Action Spaces: Deep Q-learning is well-suited for discrete action spaces, as it relies on estimating and updating Q-values for each state-action pair separately.

Overall, training by policy focuses on directly learning the optimal policy, while Deep Q-learning aims to learn the optimal action-value function. Training by policy involves parameterized policies and requires a trade-off between exploration and exploitation, while Deep Q-learning employs value iteration and exploration strategies. Deep Q-learning algorithms are generally more stable, while training by policy can be more challenging in terms of convergence. The choice between the two approaches depends on the problem domain, action space, and desired learning outcomes.

3.2 Atari - Breakout

One fascinating application of reinforcement learning is the Atari Breakout project. Inspired by the classic arcade game, this project utilizes reinforcement learning algorithms to teach an agent how to play Breakout autonomously. By providing the agent with a reward signal based on its performance, it learns to maximize its score by strategically moving a paddle to bounce a ball and break bricks. Through countless iterations and interactions with the game, the agent gradually develops strategies and reflexes, showcasing the potential of reinforcement learning to master challenging tasks. The Atari Breakout project serves as a captivating example of how this approach can unlock new possibilities in the field of artificial intelligence.

3.2.1 Parameters



Action space:

In OpenAI Gym's Atari Breakout environment, the action space is discrete and consists of four possible actions. These actions are represented by integers from 0 to 3. Here's a breakdown of each action:

Number	Action
0	Not moving the paddle (no-op)
1	Fire the ball
2	Move the paddle to the right
3	Move the paddle to the left

Observation space:

By default, the environment returns the RGB image that is displayed to human players as an observation. However, it is possible to observe

- The 128 Bytes of RAM of the console
- A grayscale images

instead. The respective observation spaces are

- `Box ([0 ... 0], [255 ... 255], (128,), uint8)`
- `Box ([[0 ... 0] ... [0 ... 0]], [[255 ... 255] ... [255 ... 255]], (250, 160), uint8)`

Reward:

The score points by destroying bricks in the wall. The reward for destroying a brick depends on the color of the brick. There are six rows of bricks. The color of a brick determines the points when hitting it with the ball.

- Red - 7 points
- Orange - 7 points
- Yellow - 4 points
- Green - 4 points
- Blue - 1 point

3.2.2 Policy

Advantage Actor-Critic (A2C):

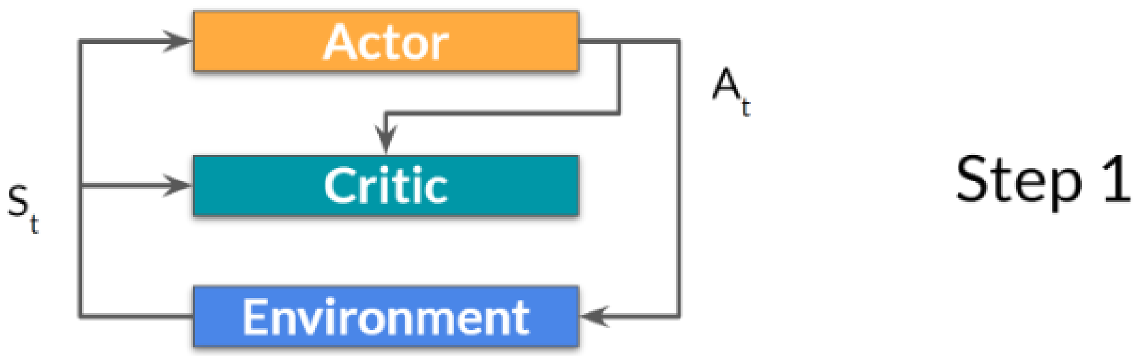
The solution to reducing the variance of the Reinforce algorithm and training our agent faster and better is to use a combination of Policy-Based and Value-Based methods: the Actor-Critic method.

At starting point, it tries some randomly actions. The Critic observes actions and provides feedback. Learning from feedback, it will update its policy and be better. On the other hand, the Critic will also update their way to provide feedback so it can be better next time. This is the idea behind Actor – Critic:

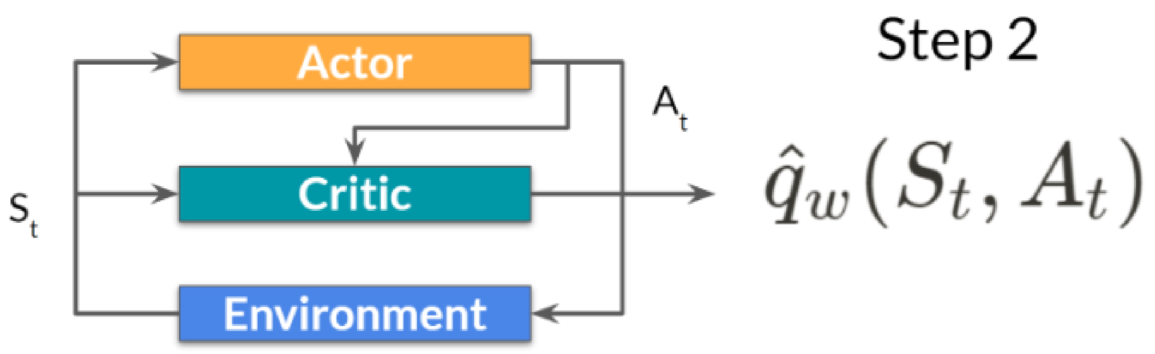
- Actor - A Policy that controls how agent acts: $\pi_\theta(s)$
- Critic - A value function to assist the policy update by measuring how good the action taken: $\widehat{q_w}(s, a)$

The Actor-Critic Process:

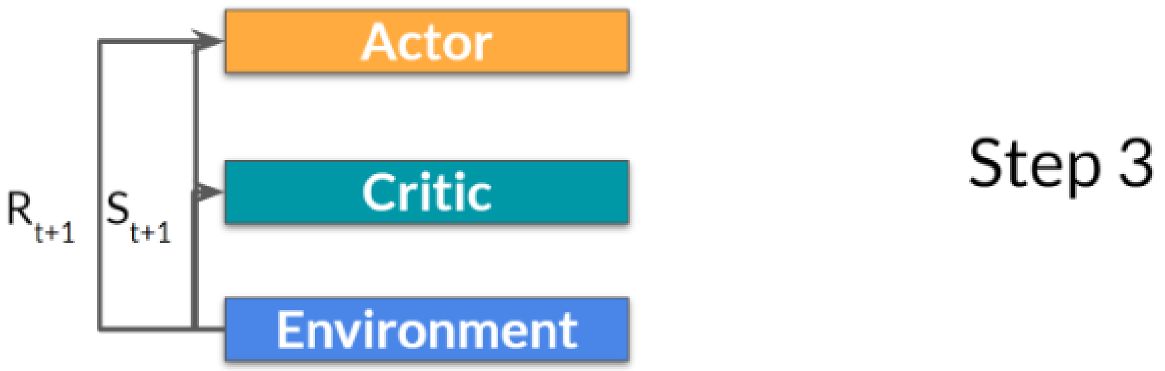
- At each timestep, t, we get the current state S_t, from the environment and pass it as input through our Actor and Critic.
- Our Policy takes the state and outputs an action A_t.



- The Critic takes that action also as input and, using S_t and A_t, computes the value of taking that action at that state: the Q-value.



- The action A_t performed in the environment outputs a new state S_{t+1} and a reward R_{t+1}

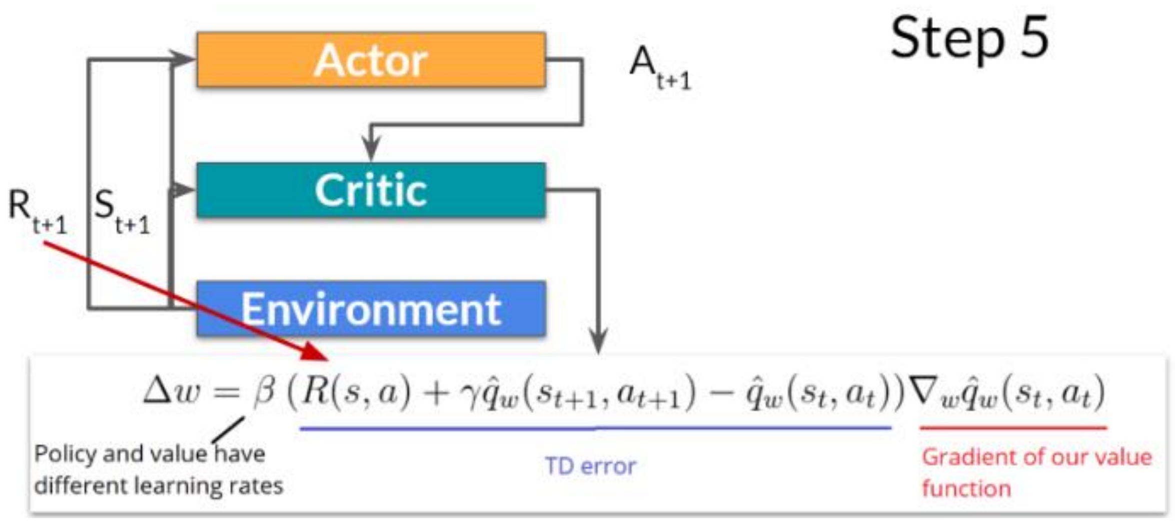


- The Actor updates its policy parameters using the Q value.

Step 4

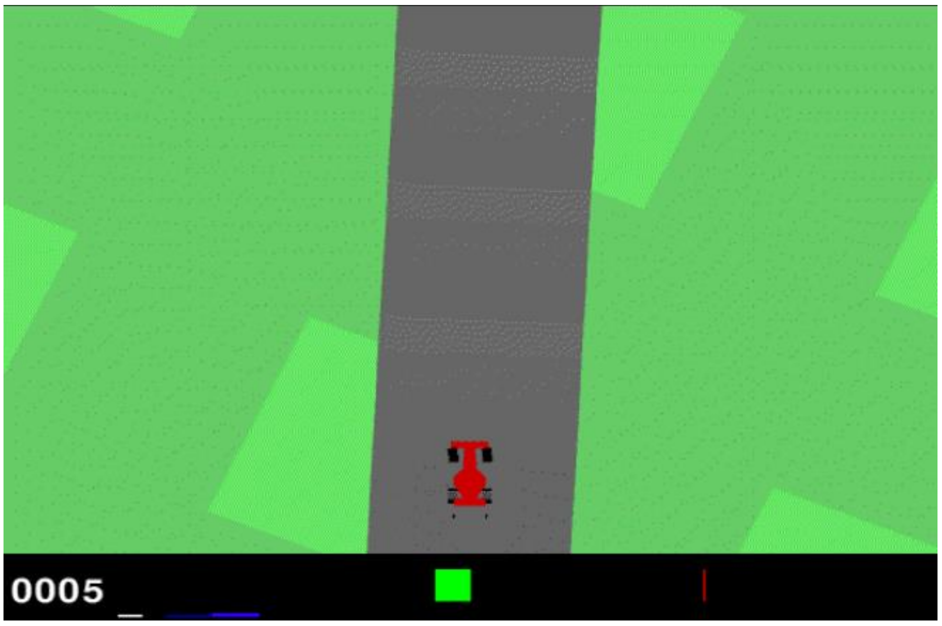
$$\underbrace{\Delta\theta}_{\text{Change in policy parameters (weights)}} = \alpha\nabla_\theta(log\pi_\theta(s,a))\underbrace{\hat{q}_w(s,a)}_{\text{Action value estimate}}$$

- The Actor produces the next action to take a A_{t+1} given the new state S_{t+1}. The Critic then updates its value parameters.


Step 5
Actor
Critic
Environment
A_{t+1}
R_{t+1}
S_{t+1}
$\Delta w = \beta\,(R(s,a) + \gamma \hat{q}_w(s_{t+1},a_{t+1}) - \hat{q}_w(s_t,a_t))\nabla_w \hat{q}_w(s_t,a_t)$
Policy and value have different learning rates
TD error
Gradient of our value function

3.3 Self-Driving Car

In the car racing environment, the objective is to control a car and navigate it through a track while optimizing certain performance metrics like speed, stability, and completing the course in the shortest time possible. The environment provides a two-dimensional view of the track from a top-down perspective.



3.3.1 Parameters

Action space:

```
Box ([-1. 0. 0.], 1.0, (3,), float32)
```

If continuous: There are 3 actions: steering (-1 is full left, +1 is full right), gas, and breaking. If discrete: There are 5 actions: do nothing, steer left, steer right, gas, brake.

Observation space:

```
Box (0,255, (96,96,3), uint8)
```

This is the image 96x96x3 which is able to train the model.

Rewards:

The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if it has finished in 732 frames, the reward is 1000 - 0.1*732 = 926.8 points.

Episode Termination:

The episode finishes when all of the tiles are visited. The car can also go outside of the playfield - that is, far off the track, in which case it will receive -100 reward and die.

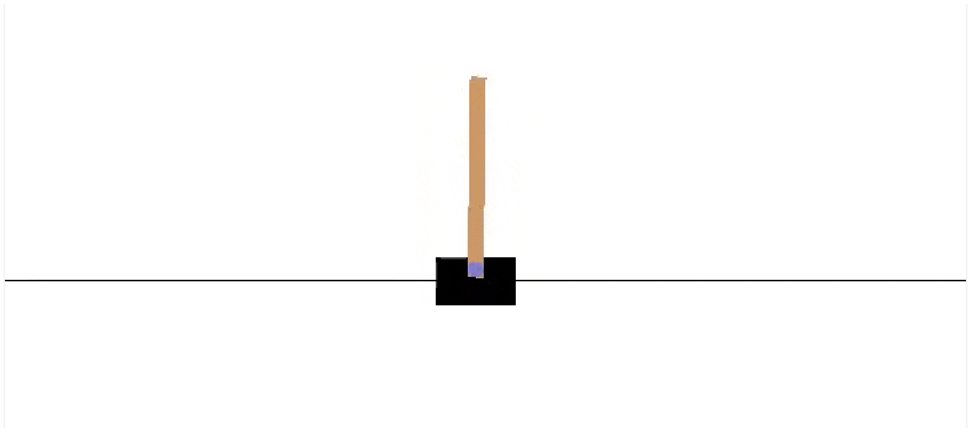
3.3.2 Policy

This time I turn back to use PPO policy because the environment meet a suitable condition as I explained above in section 3.1.2.

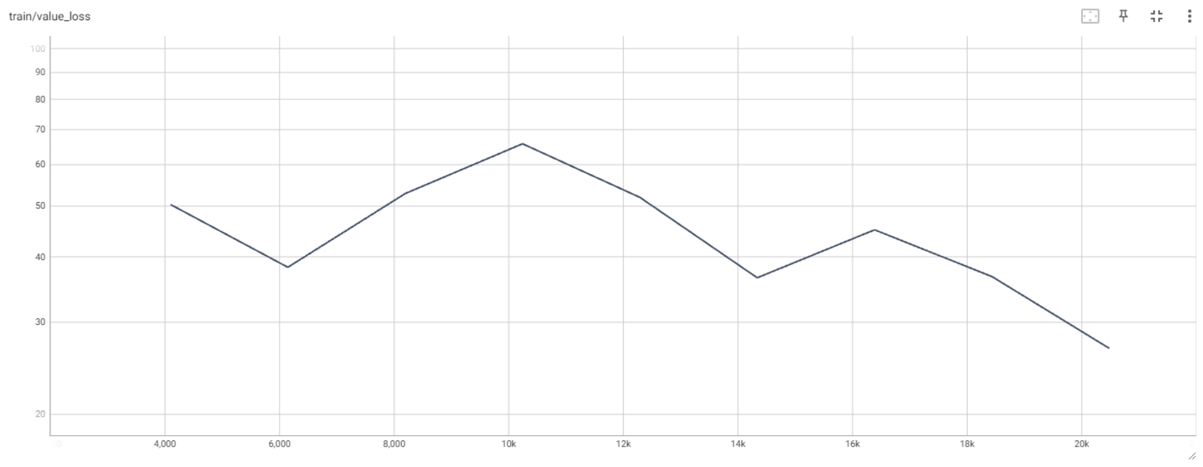
4. RESULT

4.1Cart Pole

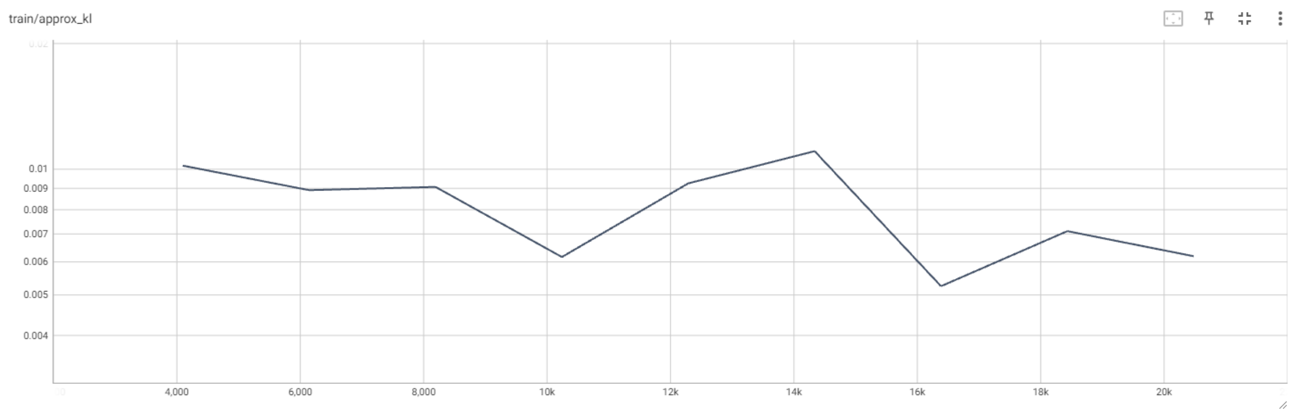
The agent of Cart Pole is trained successfully. By policy which is using PPO algorithms, the model has trained and given the best performance at 10240 timestep. When using the Deep Q learning Neural network, it took around 500 episodes, each episode has 500 timestep.



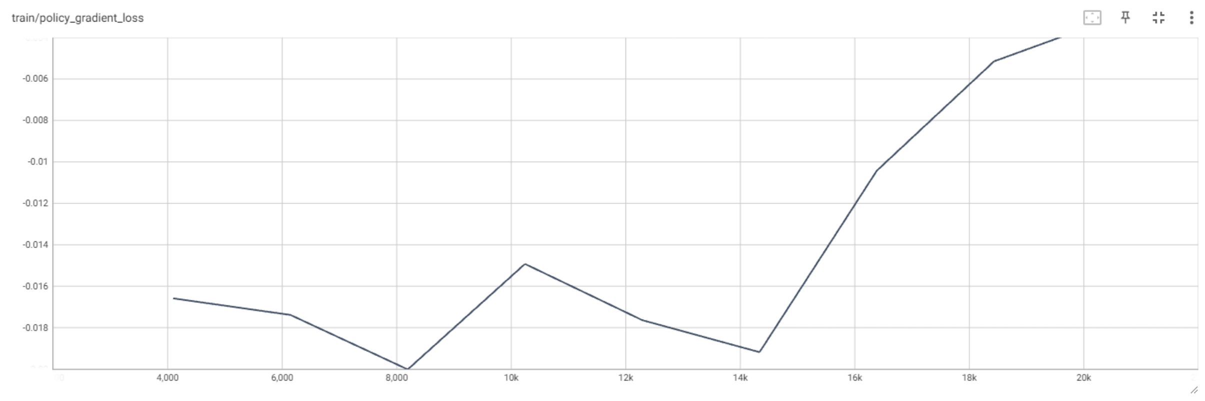
Graphs:



Value Loss — The mean loss of the value function update. Correlates to how well the model is able to predict the value of each state. This should increase while the agent is learning, and then decrease once the reward stabilizes. These values will increase as the reward increases, and then should decrease once reward becomes stable.



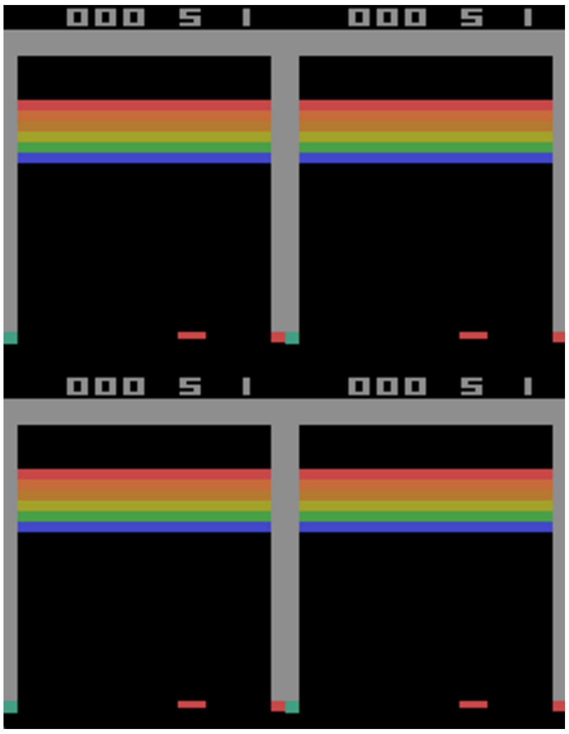
Approximate mean KL divergence between old and new policy (for PPO), it is an estimation of how much changes happened in the update.



This is the current value of the policy gradient loss in every timesteps.

4.2Atari – Breakout

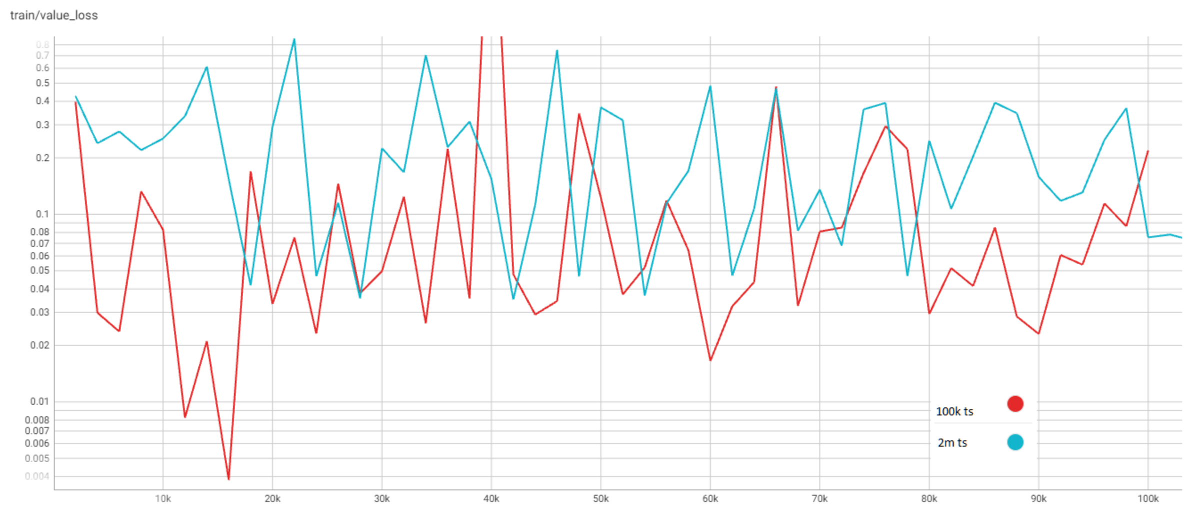
For faster training, I vectorize the environment particularly with multiple environments, allows to train the agent faster by training in parallel. I tried to training 4 environments at the same time:

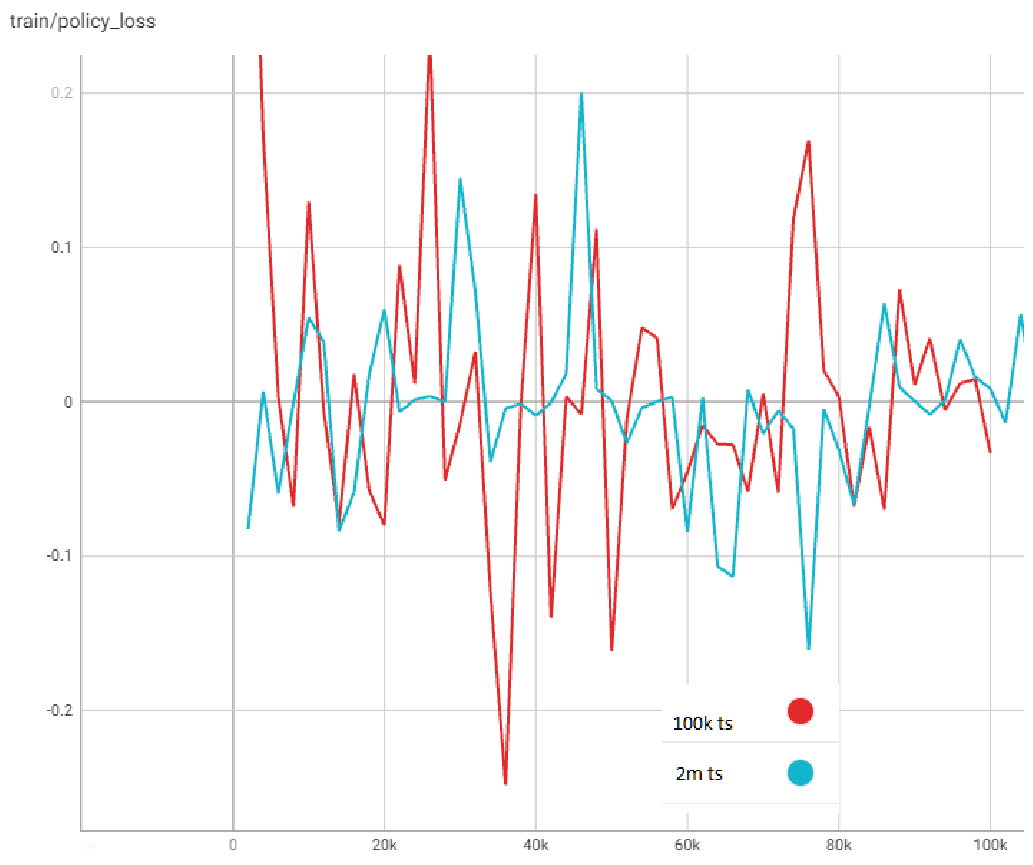


Within 100.000 steps, the agent gives the average score is 7.0

Within 2.000.000 steps, the agent gives the average score is 23.1

In the training process, comparing 100.000 timestep with 2.000.000 timestep; the model behaves as follow:





4.3 Self-Driving Car

I have test 2 models; one has trained for 1 million timesteps and one has trained for 2 million timestep and get a big difference. After 1 million timestep, the model can only drive in a short period of time and then keep spinning around. The average score even reaches to the negative value (-62.15). After 2 million timestep, the self-driving car model has reach to the average score is 664.09. So, every episode, it can track with the score around 610 to 775 points.

5. CONCLUSION AND DEVELOPMENT

5.1 Conclusion

Reinforcement learning has emerged as a powerful framework for training intelligent agents to make optimal decisions in complex environments. Through the use of trial-

and-error interactions with the environment, reinforcement learning algorithms enable agents to learn from experience and improve their performance over time.

Developments in reinforcement learning have led to significant advancements in various domains, including robotics, game playing, recommendation systems, and more. Researchers and practitioners continue to explore new algorithms, architectures, and techniques to tackle increasingly complex problems and achieve higher levels of performance.

One of the key developments in reinforcement learning has been the advancement of deep reinforcement learning, which combines reinforcement learning with deep neural networks. Deep Q-learning, in particular, has shown remarkable success in learning complex control policies, enabling agents to surpass human-level performance in challenging tasks.

5.2 DEVELOPMENT

Advancements in policy gradient methods, such as Proximal Policy Optimization (PPO) and Trust Region Policy Optimization (TRPO), have improved the stability and efficiency of learning policies in high-dimensional continuous action spaces.

Additionally, the field has seen advancements in exploration strategies, model-based reinforcement learning, multi-agent reinforcement learning, and meta-learning, which have expanded the capabilities of reinforcement learning algorithms and extended their applicability to a wide range of problems.

As the field continues to evolve, future developments in reinforcement learning are expected to focus on addressing challenges such as sample efficiency, generalization to unseen environments, safety, and interpretability. Improving the theoretical foundations, scaling algorithms to large-scale problems, and incorporating human feedback into the learning process are also active areas of research.

6. REFERENCES

[1] John N.Tsitsiklis, IEEE, Benjamin Van Roy, “An Analysis of Temporal-Difference Learning with Function Approximation”, IEEE Transactions on Automatic Control

[2] Csaba Szepesvari, “Algorithms for Reinforcement Learning”, Morgan and Claypool Publishers, June 9, 2009

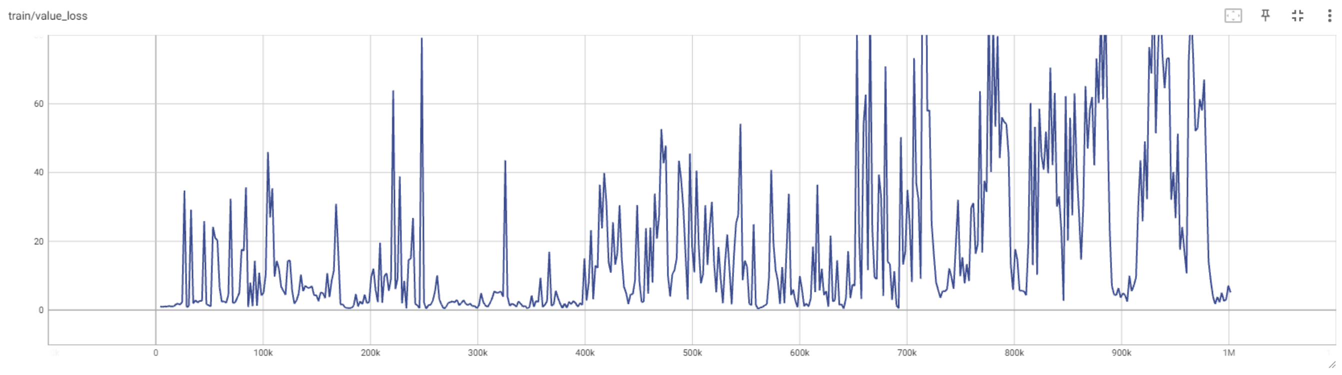
[3] Volodymyr Mnih, Alex Graves, Mehdi Mirza and members, “Asynchronus Methods for Deep Reinforcement Learning”, June 16, 2016

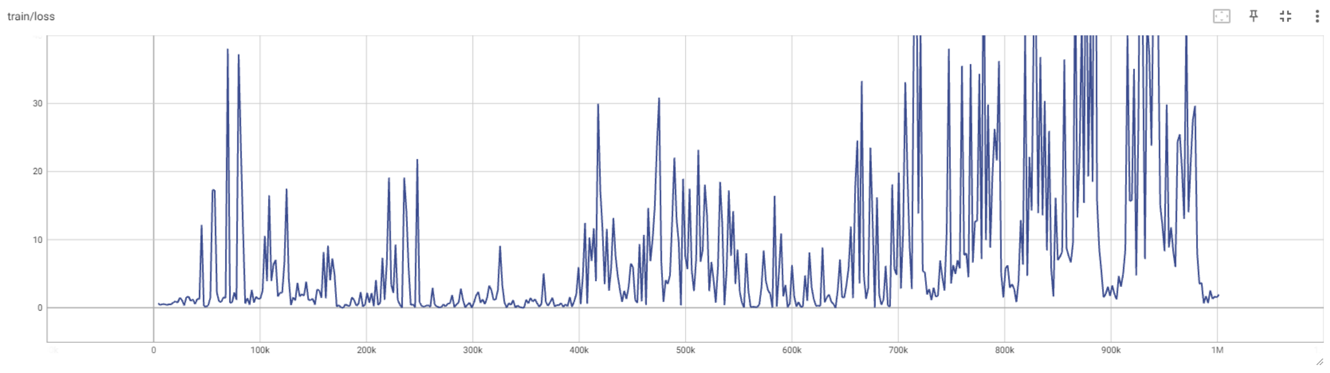
[4] Andrew G.Barto, member, IEEE, Richard S.Suton and Charles W.Anderson, “Neuronlike Adaptive Elements That can Solve Difficult Learning Control Problems”, IEEE Transactions of Systems

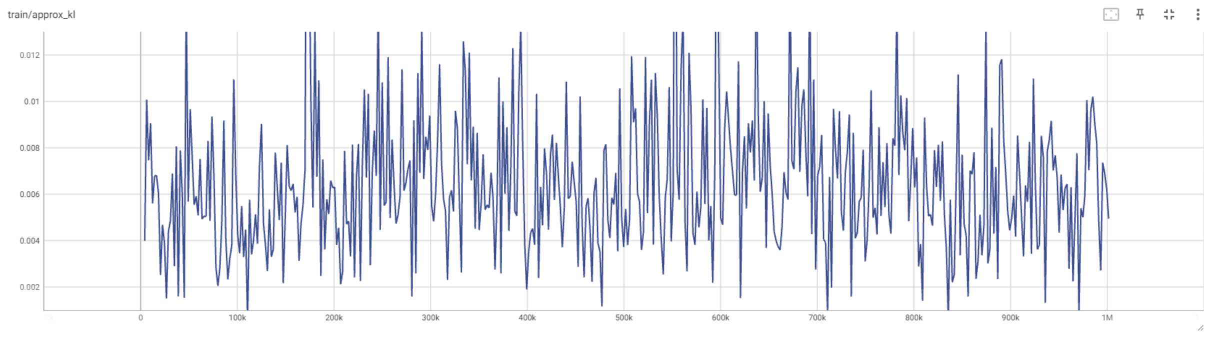
[5] RL Algorithms Docs, https://spinningup.openai.com/en/latest/algorithms

[6] CartPole parameter, https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py

7. APPENDIX






train/approx_kl
0.012
0.01
0.008
0.006
0.004
0.002
0
100k
200k
300k
400k
500k
600k
700k
800k
900k
1M