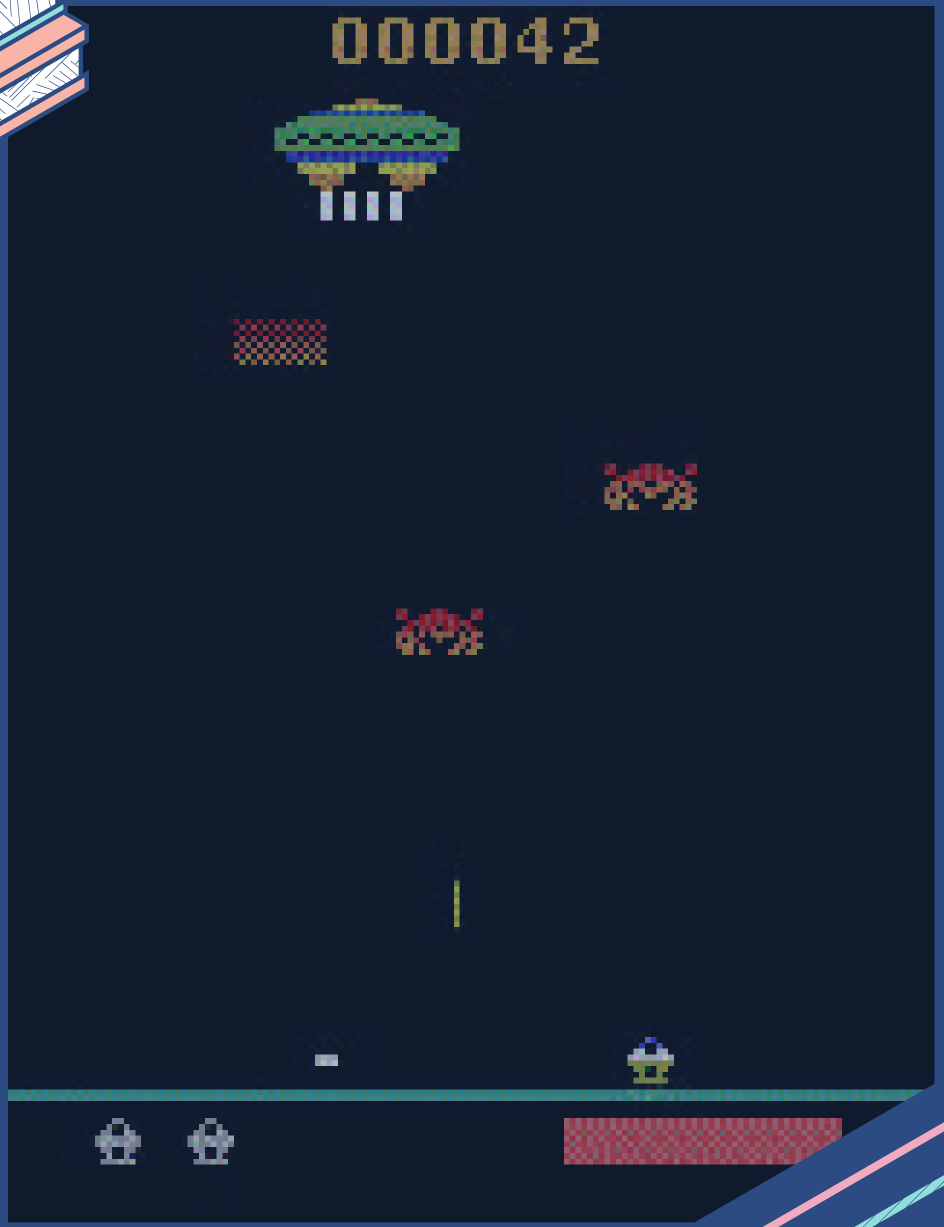


Reinforcement Learning Algorithms for Atari Game "Assault"

Group 7
Ku, Shih-Chieh (8906826)
Alakkaparambil Somasundaran, Darshik (8847842)
Soni, Lakshay (8888349)

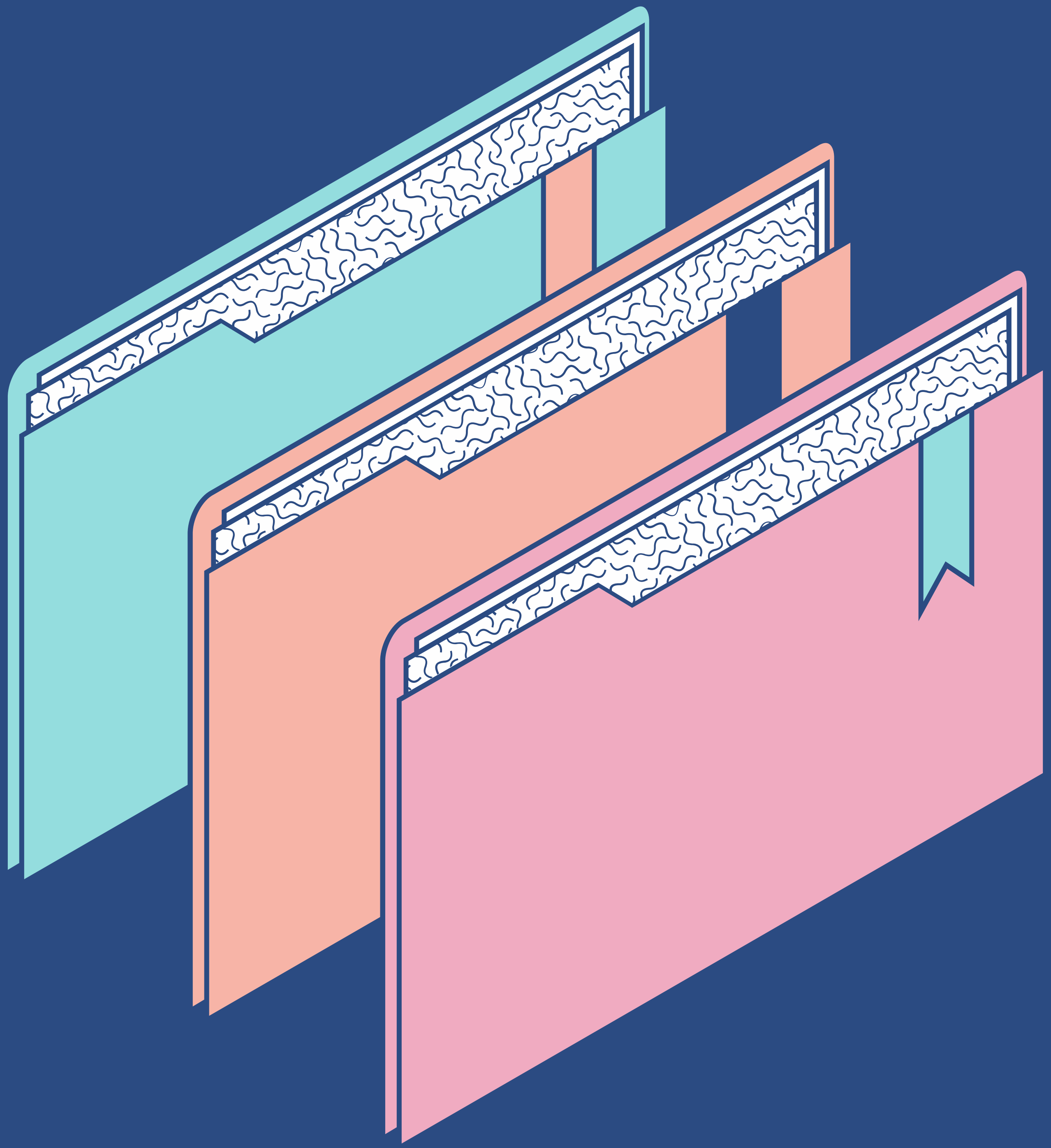
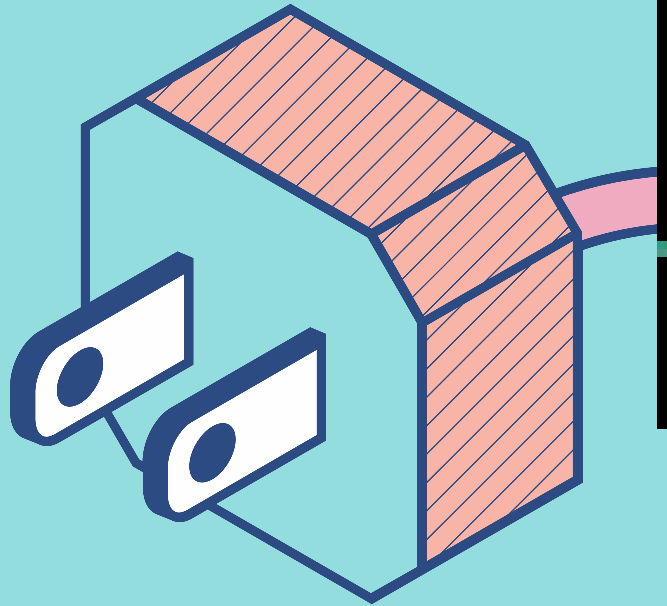
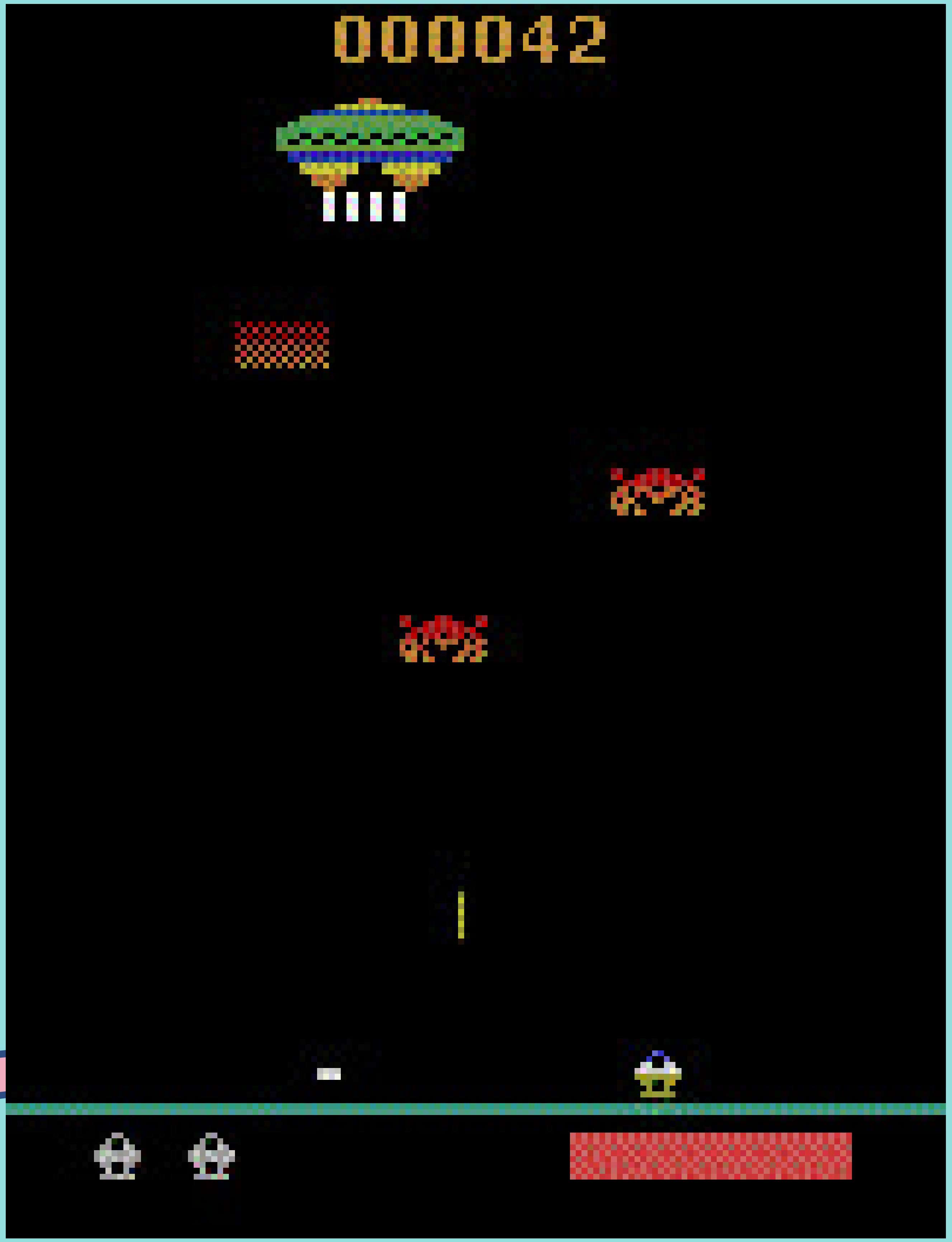


Table of contents

Assault-v5

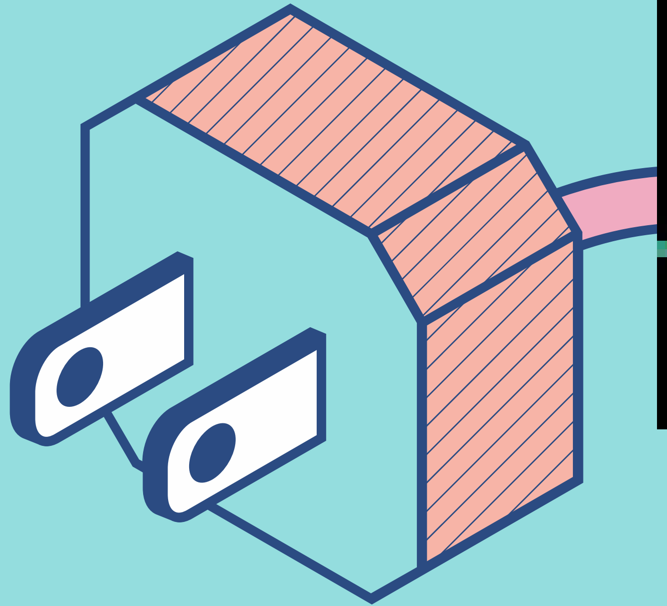
Assault-v5 is a classic Atari game in the Gymnasium environment. The player controls a ship at the bottom of the screen, aiming to shoot down enemy drones while dodging attacks. A large mothership constantly releases new enemies. The game rewards points for destroying enemies, challenging AI agents to learn both attack and defense strategies in a fast-paced setting.

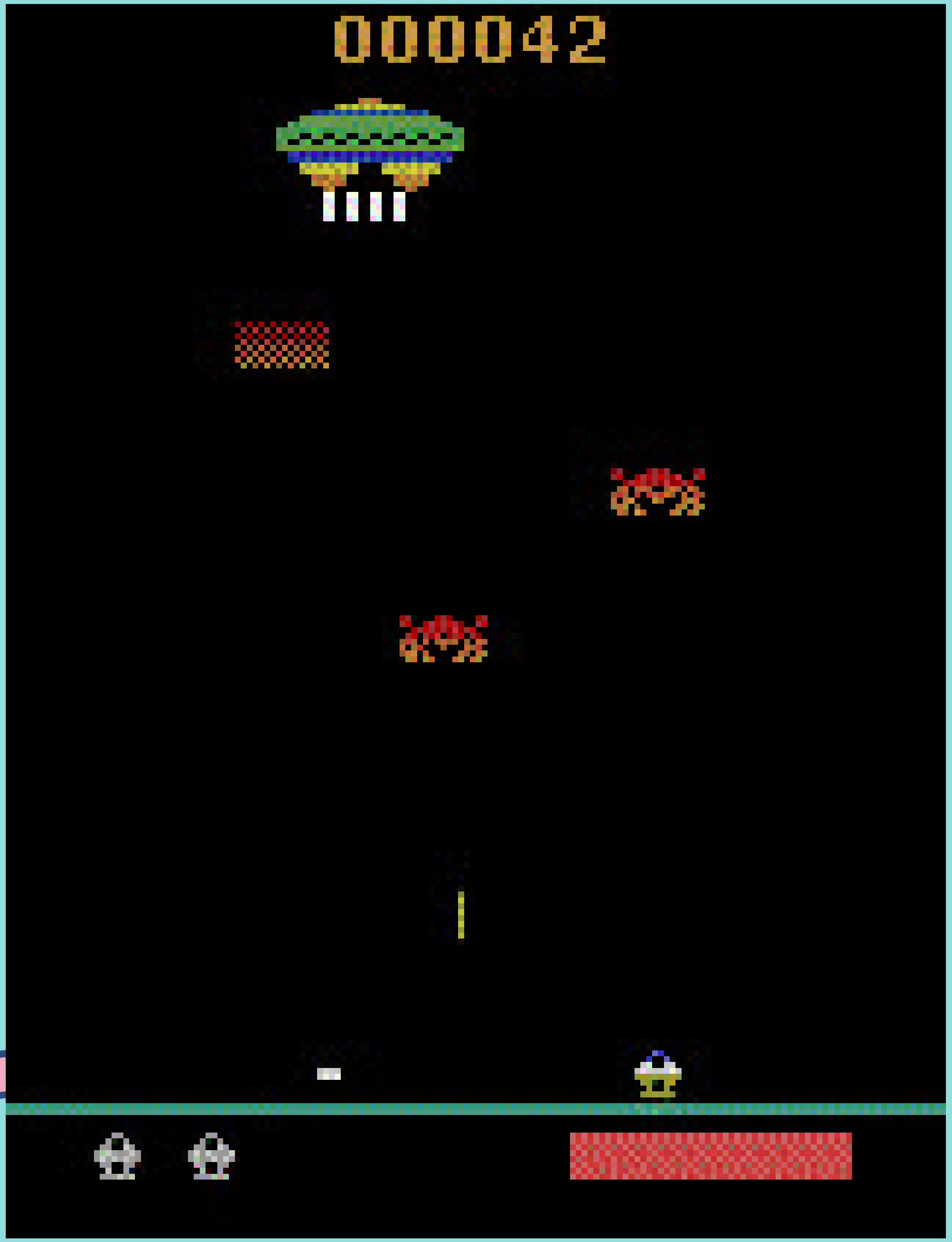




Goal

The goal is to enhance the agent's performance by leveraging and comparing different RL algorithms, namely Deep Q-Networks (DQN), and Advantage Actor-Critic (A2C).





Description as MDP

States (S):

- The state is represented by the RGB image of the game screen, which has dimensions (210, 160, 3).
- The state includes the position and status of the player's vehicle, enemy drones, the mothership, and any projectiles.

Actions (A):

The action space is discrete with up to 18 possible actions. However, a reduced action space is often used for this game:

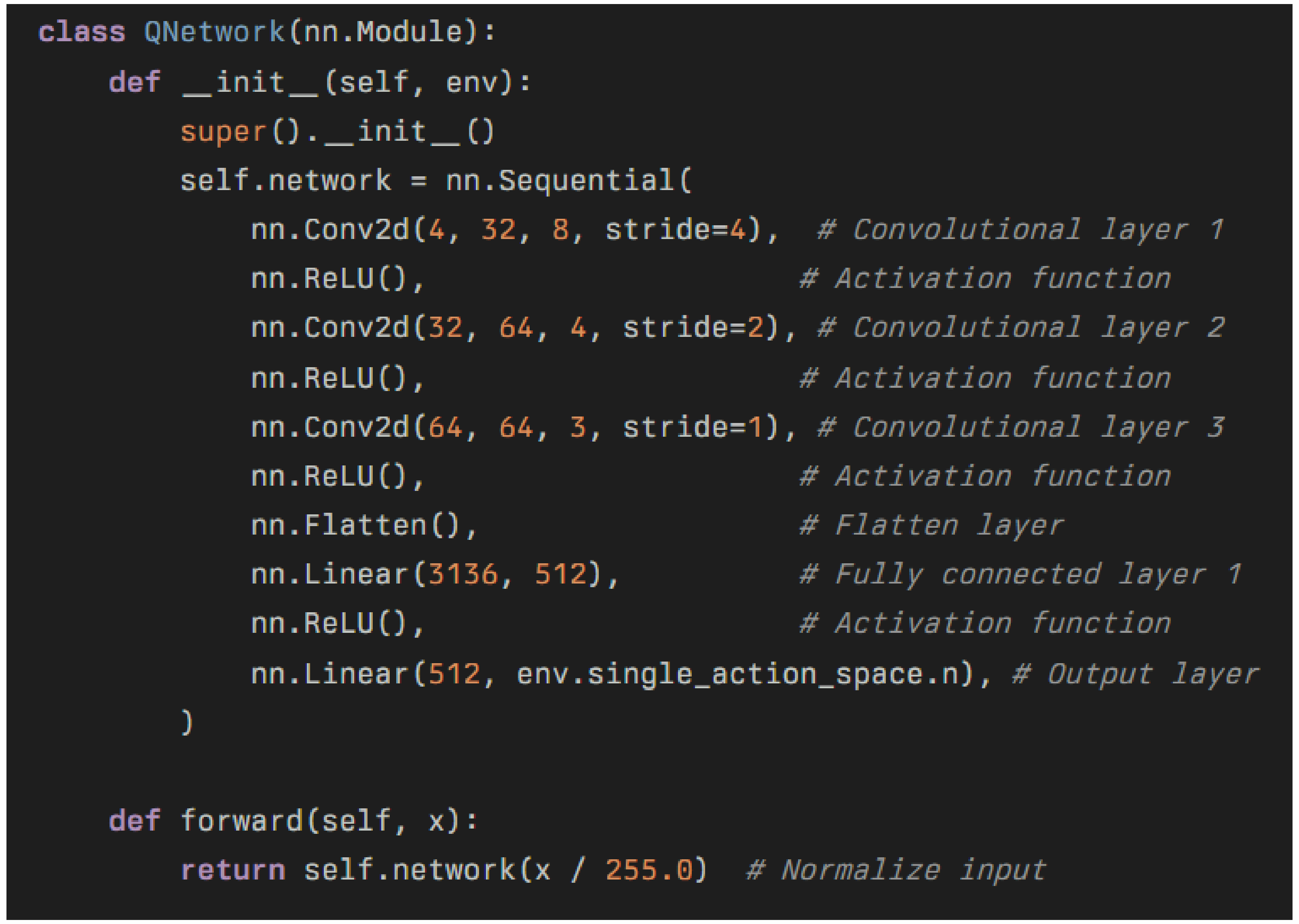
- 0: NOOP (No operation)
- 1: FIRE (Shoot)
- 2: UP (Move up)
- 3: RIGHT (Move right)
- 4: LEFT (Move left)
- 5: RIGHTFIRE (Move right and shoot)
- 6: LEFTFIRE (Move left and shoot)

These actions allow the player to control the vehicle's movement and firing.

Rewards (R):

- The reward function provides positive rewards for destroying enemy drones.
- Negative rewards or penalties may be given for being hit by enemy attacks or losing a life.
- The reward structure encourages the player to destroy enemies while avoiding damage.

Deep Q-Network



```
class QNetwork(nn.Module):
    def __init__(self, env):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(4, 32, 8, stride=4),  # Convolutional layer 1
            nn.ReLU(),                     # Activation function
            nn.Conv2d(32, 64, 4, stride=2), # Convolutional layer 2
            nn.ReLU(),                     # Activation function
            nn.Conv2d(64, 64, 3, stride=1), # Convolutional layer 3
            nn.ReLU(),                     # Activation function
            nn.Flatten(),                  # Flatten layer
            nn.Linear(3136, 512),          # Fully connected layer 1
            nn.ReLU(),                     # Activation function
            nn.Linear(512, env.single_action_space.n), # Output layer
        )

    def forward(self, x):
        return self.network(x / 255.0)  # Normalize input
```

DQN (Deep Q-Network) is a reinforcement learning algorithm that uses a neural network to approximate Q-values, combining Q-learning with deep learning to handle complex state spaces in environments like Atari games.

NETWORK ARCHITECTURE

- Convolutional Layers: Extract spatial features from the input frames.
  - Layer 1: 4 input channels, 32 filters, kernel size 8, stride 4
  - Layer 2: 32 input channels, 64 filters, kernel size 4, stride 2
  - Layer 3: 64 input channels, 64 filters, kernel size 3, stride 1
- Functions: ReLU used after each convolutional and fully connected layer.
- Flatten Layer: Converts 3D feature maps to 1D feature vectors.
- Fully Connected Layers: Process the flattened features.
  - Layer 1: 3136 input units, 512 output units
  - Output Layer: 512 input units, 7 actions output units
- Normalization: Input pixel values are normalized by dividing by 255.0.

How algorithm works - DQN

How It Works:

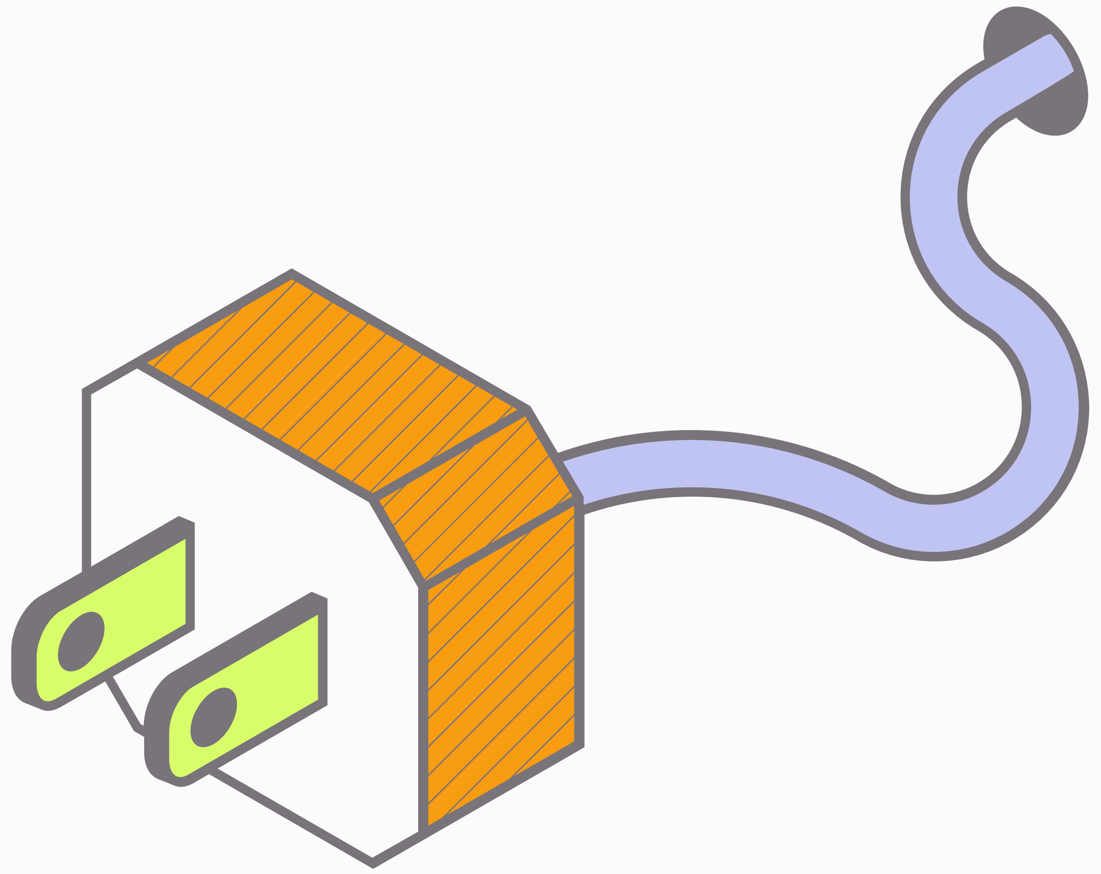
- Q-Learning:
    - DQN is based on Q-learning, a value-based method of reinforcement learning where the goal is to learn the optimal action-value function, which tells the agent the expected reward for taking action *(A)* in state *(S)*.
- Deep Neural Network:
    - A deep neural network is used to approximate the Q-function. The network takes the state (a stack of recent frames) as input and outputs Q-values for all possible actions.
- Experience Replay:
    - Experiences (state, action, reward, next state) are stored in a replay buffer. During training, random samples from this buffer are used to break the correlation between consecutive experiences and stabilize learning.
- Target Network:
    - DQN uses a separate target network to compute the target Q-values. This target network is a copy of the Q-network and is updated periodically to provide stable targets for learning.

ReplayBuffer from Baselines 3

`ReplayBuffer` is a data structure used to store and manage experiences collected by the agent during training. It is primarily used in off-policy algorithms like DQN, where experiences are sampled randomly from the buffer to break the correlation between consecutive experiences and to stabilize learning.

What is Stable-Baseline3

Stable Baselines3 (SB3) is a set of reliable implementations of reinforcement learning algorithms in PyTorch. The implementations have been benchmarked against reference codebases, and automated unit tests cover 95% of the code. The algorithms follow a consistent interface and are accompanied by extensive documentation, making it simple to train and compare different RL algorithms



A2C from Baseline3

Advantage Actor-Critic (A2C) is a synchronous, deterministic variant of Asynchronous Advantage Actor-Critic (A3C). A2C is implemented in the Stable-Baselines3 library as a popular reinforcement learning algorithm. It is designed to handle both discrete and continuous action spaces and is suitable for a wide range of environments.

Actor-Critic Architecture:

- **Actor: The policy network (actor) outputs a probability distribution over actions given the current state. The agent samples actions from this distribution.**
- **Critic: The value network (critic) estimates the value function V(s)V(s)V(s), which is the expected return from state sss.**

Advantage Function:

- **The advantage function *A(s,a)* measures how much better taking action (A) is compared to the average action in state (S). It is calculated as:**

$$A(s,a)=Q(s,a)-V(s)$$

Since $Q(s,a)$ is often approximated by $r+\gamma V(s')$, the advantage can be simplified to:

$$A(s,a)=r+\gamma V(s')-V(s)$$

How algorithm works - A2C

Collecting Rollouts:

- The agent interacts with the environment for a fixed number of steps (rollouts) and collects experiences in the form of states, actions, rewards, and next states.

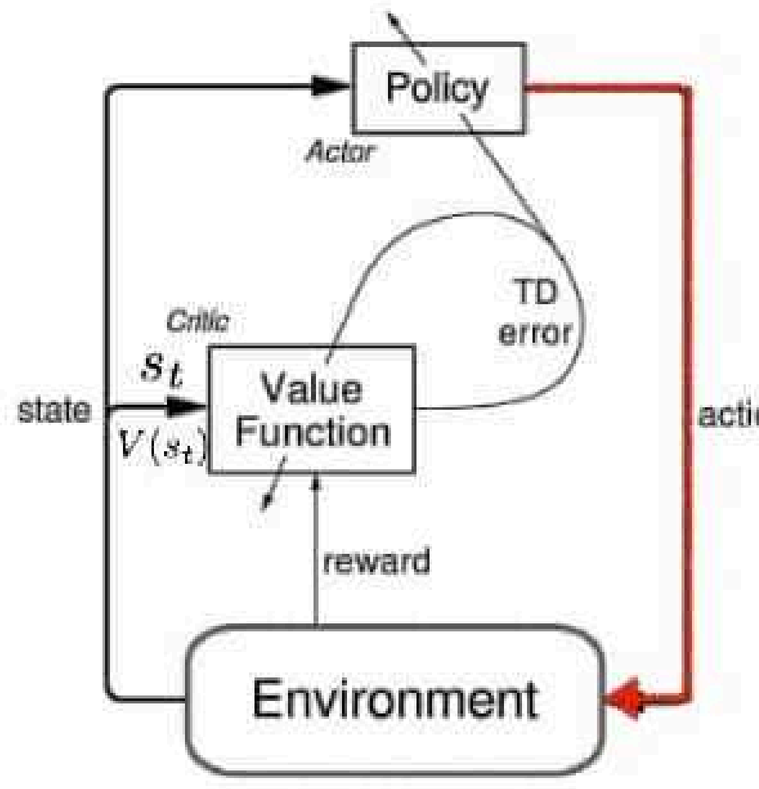
Calculating Advantages:

- The advantages are computed using the collected rollouts. This involves estimating the value of the current state and the next state using the critic network and calculating the advantage function.

Updating Networks:

- The policy (actor) network is updated by maximizing the expected advantage, encouraging actions that have higher-than-average returns.
- The value (critic) network is updated by minimizing the mean squared error between the predicted value and the target value, which is computed using the rewards and the estimated values of the next states.

Actor-Critic



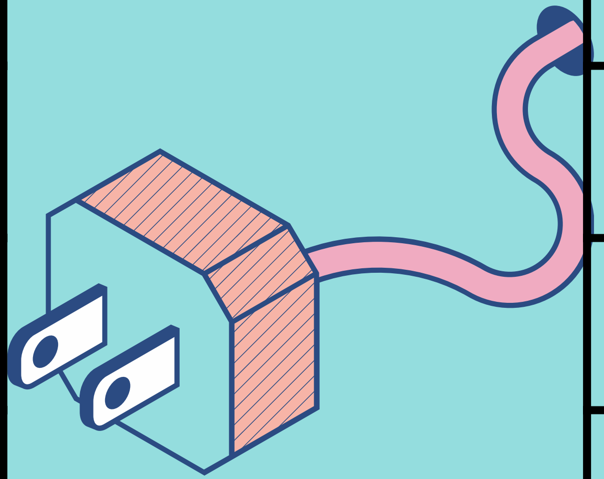
- Actor: decides which action to take
- Critic: tells the actor how good its action was and how it should adjust

(Figure from Sutton & Barto, 1998)

Variable Comparison

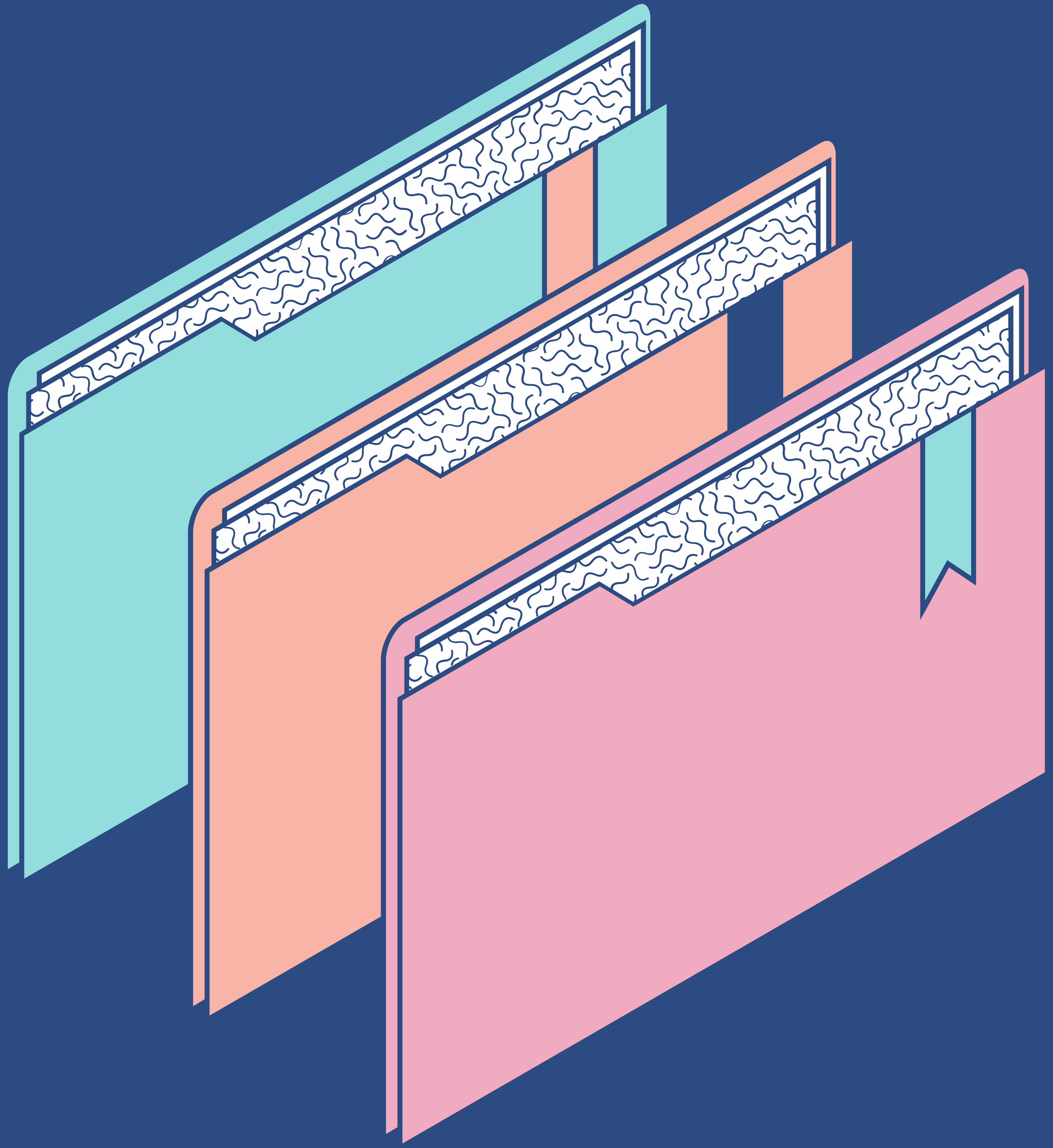
Hyperparameters of DQN

total_steps	1_000_000
learning_rate	1e-4
buffer_size	400_000
gamma	0.99
target_network_frequency	1000/steps
batch_size	32
start_epsiolon	1
end_epsilon	0.01
exploration_fraction	0.1



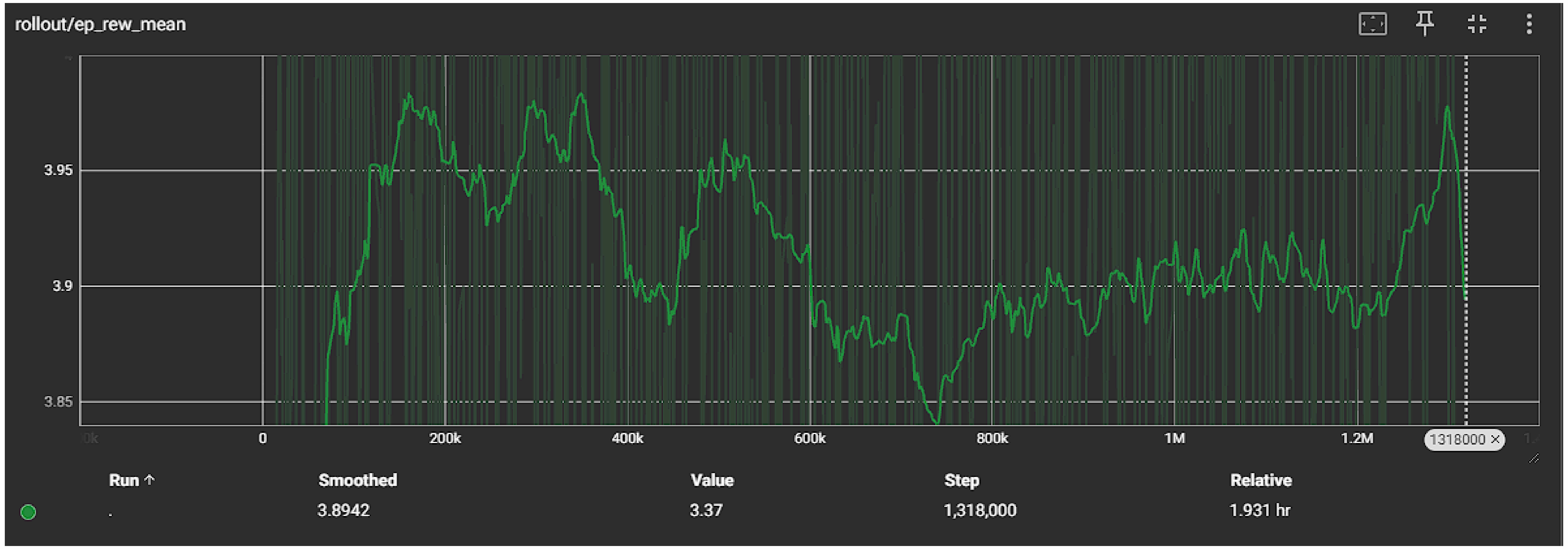
Hyperparameters of A2C

ent_coef	0.01
learning_rate	1e-4
vf_coef	0.5
gamma	0.99
n_step	5
max_grad_norm	0.5
gae_lambda	1
rms_prop_eps	1e-5
stats_window_size	100



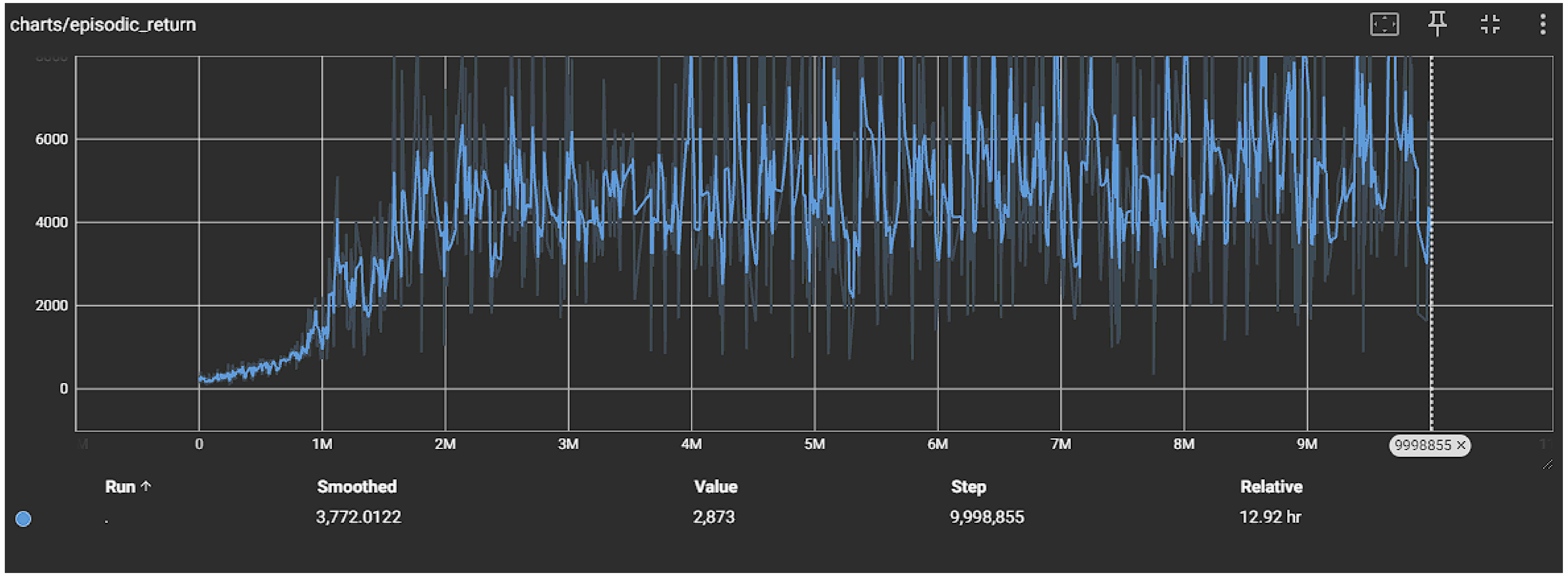
RESULTS AND ANALYSIS

A2C



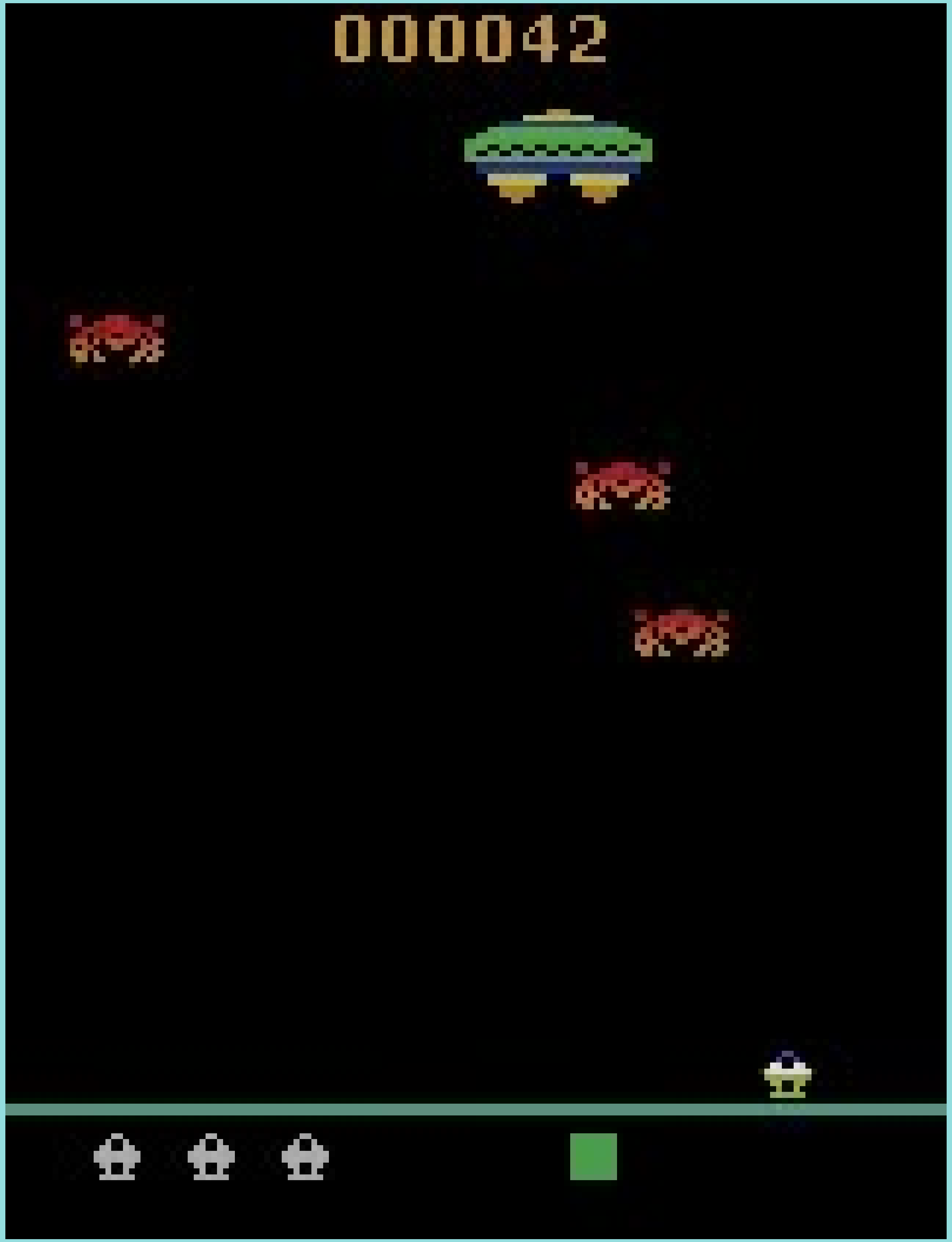
	Run ↑	Smoothed	Value	Step	Relative
●	.	3.8942	3.37	1,318,000	1.931 hr

DQN



	Run ↑	Smoothed	Value	Step	Relative
●	.	3,772.0122	2,873	9,998,855	12.92 hr

DQN



A2C