Reinforcement Learning for Playing Games

Ashwin Nellimuttath
Computer Engineering
University of California
Riverside, USA
anell003@ucr.edu

***Abstract*—Deep Reinforcement Learning (DRL) has indeed shown great promise in various sectors, including playing video games. In this article, we will review the current and cutting-edge research developments in DRL specifically applied to video games.**

Reinforcement learning (RL) is a machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment to maximize a reward signal. DRL extends this concept by utilizing deep neural networks to handle complex representations and improve learning capabilities. Deep neural networks are capable of processing high-dimensional input, making them suitable for tasks like image and audio processing, which are common in video games. There are several essential strategies within DRL that have been successfully applied to the video game space:

Value-Based Methods: These methods aim to estimate the value of different states or state-action pairs. One popular algorithm is Deep Q-Networks (DQN), which combines RL with deep neural networks to learn a value function. DQN has achieved impressive results in playing Atari games solely from visual input.

Policy Gradient Methods: Instead of estimating the value function, policy gradient methods directly learn the optimal policy. Algorithms like Proximal Policy Optimization (PPO) have been effective in training agents to play complex video games.

***Index Terms*—reinforcement learning, deep learning, deep reinforcement learning, game AI, video games.**

I. Introduction

ARTIFICIAL intelligence (AI) in video games is a long-standing research area. It studies how to use AI technologies to achieve human-level performance when playing games. More generally, it studies the complex interactions between agents and game environments. Various games provide interesting and complex problems for agents to solve, making video games perfect environments for AI research. These virtual environments are safe and controllable. In addition, these game environments provide infinite supply of useful data for machine learning algorithms, and they are much faster than real-time. These characteristics make games the unique and favorite domain for AI research. Two areas of ML that have recently become very popular due to their high level of maturity are supervised learning (SL), in which neural networks learn to make predictions based on large amounts of data, and reinforcement learning (RL), where the networks learn to make good action decisions in a trial-and-error fashion, using a simulator. With these components, there are some crucial challenges and proposed solutions. The first challenge is that the state space of the game is very large, especially in strategic games. With the rise of representation learning, the whole system has successfully modeled large-scale state space with deep neural networks. The second challenge is that learning proper policies to make decisions in dynamic unknown environment is difficult. For this problem, data-driven methods, such as supervised learning and reinforcement learning (RL), are feasible solutions. The third challenge is that the vast majority of game AI is developed in a specified virtual environment. How to transfer the AI's ability among different games is a core challenge. A more general learning system is also necessary. in the last few years, deep learning (DL) has achieved remarkable performance in computer vision and natural language processing. The combination, deep reinforcement learning (DRL), teaches agents to make decisions in high-dimensional state space in an end-to-end framework, and dramatically improves the generalization and scalability of traditional RL algorithms. Especially, DRL has made great progress in video games, including Atari, ViZDoom, StarCraft, Dota2, and so on.

The reason why it is so powerful and promising for real-life decision making problems is because RL is capable of learning continuously — sometimes even in ever changing environments — starting with no knowledge of which decisions to make whatsoever (random behavior).

In this report we will be focusing on applying RL techniques in playing games provided by the OpenAI Gym environment. Mainly we will be applying Value based methods such as DQN and policy gradient method such as PPO to learn OpenAI gym environment. We will be giving our results to 2 games - The car racing environment and the Mountain climb.

II. OpenAI Gym

OpenAI Gym is a popular toolkit for developing and comparing reinforcement learning algorithms. It provides a collection of pre-built environments that simulate various tasks and challenges. These environments are designed to be simple to understand and interact with, making them suitable for both learning and evaluating reinforcement learning agents.

A. Environments

The fundamental building block of OpenAI Gym is the Env class. It is a Python class that basically implements a simulator that runs the environment you want to train your agent in. Open AI Gym comes packed with a lot of environments, such as one where you can move a car up a hill, balance a swinging pendulum, score well on Atari games, etc. Gym also provides you with the ability to create custom environments as well.

III. Deep Q-Network

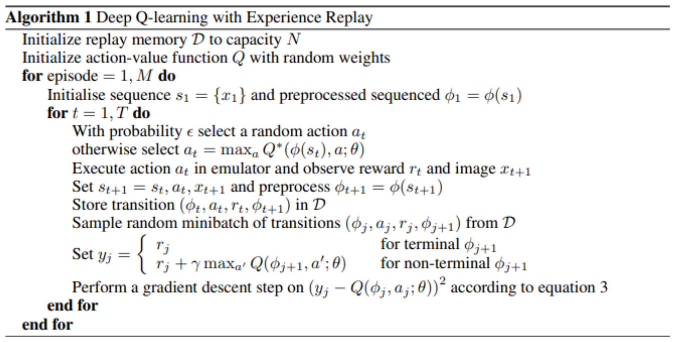
The deep Q-network (DQN) algorithm is a model-free, online, off-policy reinforcement learning method. A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. DQN is a variant of Q-learning. For more information on Q-learning, see Q-Learning Agents.

DQN has achieved human-level control in many of the games provided by the Open AI environment mainly using the 2 underlying techniques.

Experience Replay
Experience Replay stores experiences including state transitions, rewards and actions, which are necessary data to perform Q learning, and makes mini-batches to update neural networks. This technique expects the following merits.it reduces correlation between experiences in updating DNN. It increases learning speed with mini-batches and reuses past transitions to avoid catastrophic forgetting.

Target Network
In TD error calculation, target function is changed frequently with DNN. Unstable target function makes training difficult. So Target Network technique fixes parameters of target function and replaces them with the latest network every thousands steps. Additionally, the Q-Network is usually optimized towards a frozen target network that is periodically updated with the latest weights every steps (where is a hyperparameter). The latter makes training more stable by preventing short-term oscillations from a moving target. The former tackles autocorrelation that would occur from on-line learning, and having a replay memory makes the problem more like a supervised learning problem. Below I have added Qvalue update function.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$



```
Algorithm 1 Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Fig. 1. DQN Algorithm

IV. PPO - Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a family of model-free reinforcement learning algorithms developed at OpenAI in 2017. PPO algorithms are policy gradient methods, which means that they search the space of policies rather than assigning values to state-action pairs. When we talk about evaluating an agent, we generally mean evaluating the policy function to find out how well the agent is performing, following the given policy. This is where Policy Gradient methods play a vital role. When an agent is learning and doesn't really know which actions yield the best result in the corresponding states, it does so by calculating the policy gradients. It works like a neural network architecture, whereby the gradient of the output, i.e, the log of probabilities of actions in that particular state, is taken with respect to parameters of the environment and the change is reflected in the policy, based upon the gradients.

While this tried and tested method works well, the major disadvantages with these methods is their hypersensitivity to hyperparameter tuning such as choice of stepsize, learning rate, etc , along with their poor sample efficiency. Unlike supervised learning which has a guaranteed route to success or convergence with relatively less hyperparameter tuning, reinforcement learning is a lot more complex with various moving parts that need to be considered. PPO aims to strike a balance between important factors like ease of implementation, ease of tuning, sample complexity,sample efficiency and trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small. PPO is in fact, a policy gradient method that learns from online data as well. It merely ensures that the updated policy isn't too much different from the old policy to ensure low variance in training. The most common implementation of PPO is via the Actor-Critic Model which uses 2 Deep Neural Networks, one taking the action(actor) and the other handles the rewards(critic). In short, PPO behaves exactly like other policy gradient methods in the sense that it also involves the calculation of output probabilities in the forward pass based on various parameters and calculating the gradients to improve those decisions or probabilities in the backward pass. It involves the usage of importance sampling ration like it's predecessor, TRPO. However, it also ensures that the old policy and new policy are at least at a certain proximity (denoted by ϵ), and very large updates are not allowed. It has become one of the most widely used policy optimization algorithms in the field of reinforcement learning.

V. DQN Car Racing

For the purpose of implementing and learning the RL algorithms we are planning to apply DQN to the car racing environment provided by the openAI environment. The car racing environment has an image observation space. We can see how we can convert an image observation space into a Markovian environment. We have to know that we are given only one current game frame for each step. This observation setting cannot satisfy the Markov property. We cannot guess if the car is moving forward or backward from only one frame, which means we cannot predict the next frame for given the current frame. Thus, we need to stack the previous frames together. We also need preprocessing on each given frame. we can clip the image size to resize frames to 84 * 84 ones. We can also convert the image into greyscale to reduce the size. For converting and processing the image we are following the same methods as in the original DQN paper - The input to the neural network consists is an 84 × 84 × 4 image produced by . The first hidden layer convolves 16 8 × 8 filters with stride 4 with the input image and applies a rectifier nonlinearity [10, 18]. The second hidden layer convolves 32 4 × 4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fullyconnected linear layer with a single output for each valid action. We can now provide this as input to our DQN algorithm. For the DQN we will be needing to configure the replay buffer size and it requires careful consideration because of the memory requirements. The car racing environment is generally a time consuming algorithm for our agent to learn and we will require some early exit mechanisms to speed up the training process. We can also evaluate our agent as we are learning phase. Initially while I started running the algorithm without any custom termination control, it was taking a lot of time to learn few timesteps. As the number of tiles in the environment is very high reaching the endgoal or covering all the tiles requires a lot of time. So our agent was taking lot of resources to learn but with minimal results. After that we tried Early termination. In the CarRacing environment, you can implement early termination by monitoring the cumulative negative reward over a certain number of steps. If the episode receives a negative reward for a specified number of consecutive steps, you can terminate the episode early. After this steps I was able to train a little longer for about 50,000 timesteps until my collab kenel stopped functioning. We did try debugging the situation but was unable to reach a proper solution. For the timesteps the algorithm was able to learn, our RL agent was not able to conquer the racing track.

A. Results

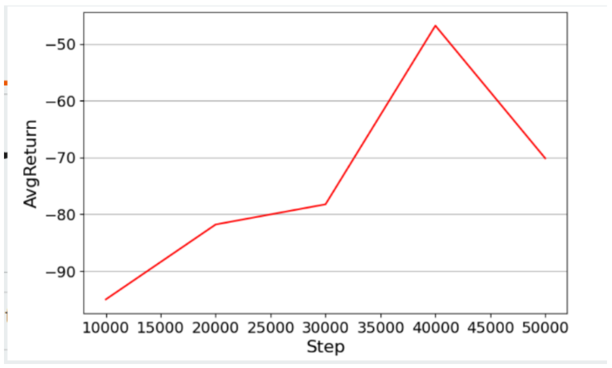


Fig. 2. Average returns per steps.

We were able to learn mainly how to apply RL techniques to applications where we have images and this applies more to the general applications we have in the world. We are also able to learn how we can speed up training by effectively implementing termination mechanisms. We learnt how memory and GPU intensive ML applications can be.

VI. Mountain Climb

In this part we are trying to solve the mountain car climb problem using Reinforcement learning. The environment is provided by OpenAI environment and our goal is to reach the top of the mountain. The cars engine is not strong enough to reach the top. We can only move left or right. We have to gain momentum and then reach the top of the hill. We will mainly be focusing on the DQN algorithm and PPO algorithm.

A. DQN

The DQN algorithm combines Q-learning, a classic reinforcement learning algorithm, with a deep neural network to approximate the Q-values of different state-action pairs.

Here's a step-by-step explanation of the implementation:

Environment Setup: First, you need to set up the MountainCar environment. This environment provides an interface for the agent to interact with the car's actions and observe the current state. The state in MountainCar is represented by the position and velocity of the car.

Deep Q-Network Architecture: The next step is to define the architecture of the DQN. The network typically consists of several layers of neurons, with the input layer taking the state as input and the output layer producing Q-values for each possible action. In MountainCar, there are three discrete actions: move left, move right, or do nothing.

Experience Replay: To stabilize the learning process, experience replay is employed. This technique involves storing the agent's experiences (state, action, reward, next state) in a replay buffer. During training, mini-batches of experiences are randomly sampled from the replay buffer to decorrelate the data and improve learning efficiency.

Q-Network Training: Initially, the DQN starts with random weights. The agent selects an action based on an exploration-exploitation trade-off, using an ϵ-greedy policy. With a certain probability ϵ, the agent selects a random action to explore the environment; otherwise, it selects the action with the highest Q-value.

Action Execution and Observation: The agent executes the selected action in the environment, which transitions it to the next state and provides a reward. In MountainCar, the agent moves the car according to the chosen action, and the environment calculates the new position and velocity. The agent observes the new state and reward.

Q-Value Update: Using the observed reward and the new state, the DQN updates the Q-value of the previous state-action pair. The Q-value update follows the Q-learning equation, which uses the Bellman equation to estimate the optimal action-value function. The DQN approximates the Q-value using the deep neural network.

Experience Replay and Target Network: After updating the Q-values, the experience (state, action, reward, next state) is stored in the replay buffer. Periodically, a target network is created as a copy of the main Q-network. The target network is used to calculate the target Q-values during the Q-value update. This separation helps stabilize the learning process and mitigate the risk of divergence.

Iterative Training: The weights of the Q-network are updated through backpropagation, minimizing the difference between predicted Q-values and target Q-values.

Evaluation: Once the training is complete, the performance of the trained DQN can be evaluated. The agent interacts with the environment using its learned policy without any exploration. The performance metrics, such as the number of steps taken or the average reward, can be used to assess the agent's performance.

By following these steps, the DQN algorithm can learn to navigate the MountainCar environment and eventually reach the goal of reaching the top of the hill. The combination of Q-learning with a deep neural network allows the agent to approximate the Q-values efficiently and handle complex state-action spaces.

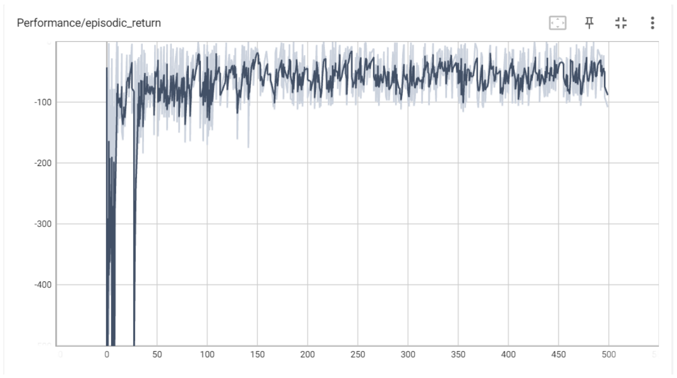
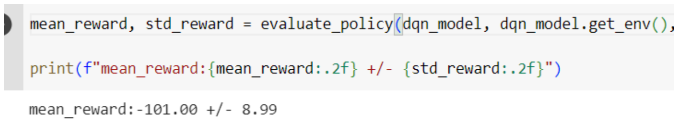


Fig. 3. Performance- episodicreturn



```
mean_reward, std_reward = evaluate_policy(dqn_model, dqn_model.get_env(),

print(f"mean_reward:{mean_reward:.2f} +/- {std_reward:.2f}")

mean_reward:-101.00 +/- 8.99
```

Fig. 4. dqnmean

1) Results: After training I was able to get average of -101 +- 10 consistently. Our DQN algorithm implementation was very quick to learn the mountaincar environment. In MountainCar, the agent receives a negative reward at each time step until it reaches the goal state. This makes it challenging to learn the optimal policy directly. DQN addresses this issue by using the Q-learning algorithm, which accounts for future rewards. By approximating the Q-values with a deep neural network, DQN can effectively estimate the long-term rewards and learn to take actions that maximize cumulative rewards.

VII. PPO

Here is a high-level overview of implementing the PPO algorithm in the MountainCar environment:

Policy Network: Create a neural network, known as the policy network, which takes the current state of the car as input and outputs a probability distribution over the available actions. This network represents the policy that the agent uses to select actions.

Value Network: Additionally, create another neural network, known as the value network, which estimates the expected return (or value) for a given state. This network helps in estimating the advantage or the value of each action taken by the agent.

Data Collection: Generate multiple trajectories by running the current policy in the environment. A trajectory consists of a sequence of states, actions, rewards, and next states experienced by the agent while interacting with the environment. Collect the trajectories for a certain number of steps or episodes.

Compute Rewards and Advantages: Calculate the rewards and advantages for each state-action pair in the collected trajectories. The rewards are typically computed as the sum of future discounted rewards from that state onwards, and the advantages are calculated by subtracting the estimated value of the state from the total reward obtained.

Policy Update: Use the collected trajectories and the computed rewards and advantages to update the policy network. The PPO algorithm uses a surrogate objective function to optimize the policy while ensuring that the policy update is within a certain trust region. This ensures that the new policy does not deviate too far from the old policy, preventing catastrophic changes.

Value Function Update: Update the value network using a regression loss function that minimizes the difference between the predicted values and the discounted rewards obtained during data collection.

Repeat: Iterate the steps of data collection, policy update, and value function update for a certain number of iterations or until the agent's performance reaches a satisfactory level.

1) Implementation: Stable Baselines is a Python library that provides high-quality implementations of various reinforcement learning (RL) algorithms. It is built on top of OpenAI Gym, which is a popular RL toolkit that provides a collection of pre-implemented environments for testing and benchmarking RL algorithms.

Stable Baselines offers a set of algorithms that are widely used in the RL community. For the implementation of the PPO algorithm we are using the stable baseline version.

2) Results: We were able to achieve almost similiar results as the DQN version of the algorithm after a certain timesteps. For the same number of timesteps as the DQN algorithm our PPO implementation had a much higher average of 150 +- 10. Eventually our PPO implementation was to learn the environment. For the code to provide different environment I have implemnted for the continuos version. To change into discrete version we just need to make a small change in the code use_sde=True.

VIII. Comparison - DQN,PPO

In the MountainCar environment, the goal is for the car to reach the top of a hill. However, the environment provides a sparse reward signal, meaning the agent only receives a positive reward when it reaches the top and a negative reward otherwise. This creates a challenge for learning because reaching the goal is a rare event, making it difficult for on-policy algorithms like PPO to learn effectively.

With PPO being an on-policy algorithm, it updates its policy based on the data collected within each episode and discards it afterward. In the case of MountainCar, where reaching the goal is infrequent, it's unlikely that a single policy gradient update will be enough to consistently reach the goal. As a result, PPO can get stuck without a learning signal until it happens to reach the goal again by chance.

On the other hand, off-policy algorithms like DQN store past experiences in a replay buffer instead of immediately discarding them. When the agent finally reaches the goal, that event is stored in the replay buffer, preserving the rare successful trajectory. The off-policy nature of DQN allows it to learn from these rare events over multiple updates. In summary, off-policy algorithms like DQN have an advantage in sparse reward environments like MountainCar because they can store and learn from rare successful events, while on-policy algorithms like PPO may struggle due to the infrequent occurrence of positive rewards and the immediate discarding of collected data.

1) Note: While both DQN and PPO are reinforcement learning algorithms, they have distinct differences:

Value-based vs. Policy-based: DQN is a value-based algorithm that learns the action-value function, while PPO is a policy-based algorithm that directly optimizes the policy.

Exploration: DQN incorporates exploration through an ϵ-greedy strategy, whereas PPO does not have an explicit exploration strategy. Instead, exploration in PPO is implicitly achieved by sampling actions according to the policy.

Action Selection: DQN typically uses a discrete action space, selecting actions based on the Q-values. PPO, on the other hand, can handle both continuous and discrete action spaces.

Stability: PPO is known for its stability and sample efficiency, as it performs updates in a conservative manner. DQN can be more sensitive to hyperparameters and may require careful tuning to ensure stable learning.

Sample Efficiency: PPO is often considered more sample-efficient than DQN, as it can make effective use of collected data by performing multiple optimization epochs. DQN, especially in its original form, can be data inefficient due to the correlations between consecutive experiences.

Both DQN and PPO have their strengths and weaknesses, and the choice between them depends on the specific problem, environment, and requirements of the task at hand.

References

[1] https://spinningup.openai.com/en/latest/
[2] https://ieeexplore.ieee.org/document/9781752
[3] https://www.anyscale.com/blog/an-introduction-to-reinforcement-learning-with-openai-
[4] https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2
[5] https://paperswithcode.com/method/dqn
[6] https://arxiv.org/pdf/1912.10944.pdf#page=11&zoom=100,416,646
[7] https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3
[8] https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
[9] https://www.geeksforgeeks.org/a-brief-introduction-to-proximal-policy-optimization/
[10] https://openai.com/research/openai-baselines-ppo
[11] https://anurag2das.medium.com/reinforcement-learning-train-your-own-agent-using-de
[12] https://www.toptal.com/machine-learning/ deep-dive-into-reinforcement-learning
[13] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
[14] https://hiddenbeginner.github.io/study-notes/contents/tutorials/2023-04-20_CartRacing-v2_DQN.html.
[15] https://colab.research.google.com/drive/16-H1ZiyClJCxVJlhwxBHIS4vOCq0pIek# scrollTo=LHwQvrCC5sRd
[16] https://chat.openai.com/