Training an Agent to play Atari 2600: using Deep Reinforcement Q-Learning

Pratik Jagad, Harshith Manjunath, Dhanvin Patel, Sree Sarika Manikonda, Mingze Sun
Stevens Institute of Technology, Hoboken, NJ

Abstract

This project explores the development of an AI agent using deep reinforcement learning techniques for playing Atari 2600 games. By leveraging Gymnasium as the environment and implementing the Deep Q-Network (DQN) algorithm, the agent learns to interpret game states, make decisions, and maximize its score. The architecture incorporates convolutional layers to extract relevant features from preprocessed grayscale images, while experience replay prevents network divergence. The results showcase the efficacy of deep reinforcement learning in training an AI agent to master complex gameplay tasks.

Introduction

Reinforcement learning techniques can be traced back to last century. It is a subfield of artificial intelligence and machine learning that focuses on how an agent can learn to make decisions in an environment to maximize a reward signal. It is inspired by the way humans and animals learn through trial and error. In late 2013, a then little-known company called DeepMind achieved a breakthrough in the world of reinforcement learning: using deep reinforcement learning, they implemented a system that could learn to play many classic Atari games with human (and sometimes superhuman) performance.

This project is intended to develop an Artificial Intelligence agent, by this, we do not mean to solve the general problem of AI(i.e to develop an AI agent to deal with problems of the real world), but to build a simple AI agent that uses Deep Reinforcement Learning algorithm that can complete a task of some sort of successful ability. Our task is to use a standard environment such as Gymnasium (open source Python library for developing and comparing reinforcement learning algorithms, developed by the Farama Foundation) and build an agent to learn in this environment.

Our focus is on developing an agent that can play an Atari 2600 game(the game environment will be updated once we start making progress in our project), which has a wide range of applications in the field of reinforcement learning and AI. The AI agent must learn to identify the game environment, interpret the game state, and take actions that maximize its score. The challenge lies in designing a reinforcement learning algorithm that can learn complex strategies and overcome obstacles in real time. The solution requires using deep neural networks as function approximators to estimate the action-value function and the Q-learning algorithm to update the parameters of the neural network.

Atari Games

Atari games hold a special place in the history of video gaming, having been developed by Atari, Inc. during the 1970s and 1980s. These games, such as Space Invaders, Pac-Man, and Breakout, are known for their simplicity and addictive gameplay. They provide diverse challenges and serve as an ideal platform for testing and developing AI algorithms. The project focused on utilizing Atari games as a testbed to train an agent capable of achieving high scores and surpassing human-level performance. The wide variety of Atari games available ensured a rich and diverse training environment for the agent.

Models

Various models were employed in the project to enhance the performance of the Atari game-playing agent. Deep reinforcement learning (DRL) algorithms and neural networks played a crucial role in training the agent. DRL algorithms, such as Deep Q-Network (DQN) and Proximal Policy Optimization (PPO), were utilized to learn optimal strategies through interactions with the game environment. Neural networks were used to represent game states and approximate the Q-values, enabling more efficient decision-making by the agent. These models demonstrated their effectiveness in improving the agent's gameplay and overall performance.

Gymnasium Library

The OpenAI Gymnasium library served as a valuable resource throughout the project. OpenAI Gym is an

open-source Python library that provides a standardized interface for interacting with various game environments, including the Atari 2600 series. Gymnasium offered a wide range of Atari game environments, ensuring a comprehensive training and evaluation platform for the agent. It provided utilities for preprocessing game frames, receiving rewards, and performing actions within the game environment. The library's standardized interface streamlined the development process and allowed for easy integration of the agent with different Atari games, making it a powerful tool for training AI agents.

Reinforcement Learning

Reinforcement learning (RL) is a very popular area of Artificial Intelligence research. It is a machine learning technique in which an agent makes decisions based on the context of an environment. The agent learns through trial and error to maximize its reward based on the feedback generated from its actions and experiences. In Figure 1, a pictorial representation of the reinforcement learning model is represented. Various applications are a good match for the RL approach as the environment is unspecified and the outcome of actions are undetermined. Some real-world applications include autonomous cars and robotics.

Video games have been a popular domain for RL implementation as they provide interesting and complex problems for agents to solve. One such example [4] is where an agent learned to play the Atari 2600 with deep reinforcement learning. In this project, we sought to implement a RL algorithm capable of learning how to play the Atari breakout game.

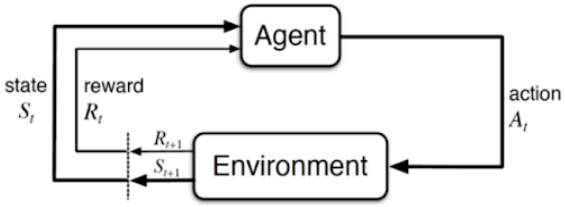

Figure 1. Reinforcement learning cycle

Markov Decision Process

Reinforcement learning is about solving Markov Decision Processes (MDPs). An MDP is a formal way to describe a game in terms of states, actions, and rewards. It defines a mechanism of how certain states & agent's actions lead to other states. Playing Breakout can be formalized as a finite state MDP.

The state is the current situation the agent is in, which can be approximated in the Atari game using the current frame. While a single frame may be useful, it becomes difficult to determine the motion of the ball as going up or going down. Utilizing a single frame may end up violating the Markov property of the MDP, history does not matter. In order to satisfy the Markov property, the previous states must not have any useful information. In the case of Breakout, using a single frame will break the Markov property as the previous frames may help the agent to learn about the speed of objects and infer its acceleration.

The action is a command given to reach a certain state in the game or attain a reward. Using the joystick, one can perform actions like doing nothing, asking for the ball, going left, and going right in the Breakout game.

Last component of the MDPs are the rewards the agent receives upon completion of an action. Rewards are calculated by considering the starting state, action performed, and the end state. The goal of the reinforcement learning program is to learn the optimal behavior in an environment to maximize long term rewards. In the case of Atari 2600, rewards correspond to changes in score as shown in Figure 2.

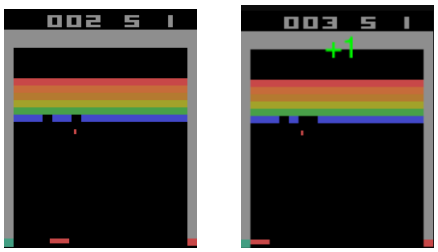

Figure 2. Score increases from 2 to 3, a reward of +1 is attained.

Q-Learning

Quality Learning is a "Model-free" reinforcement learning algorithm that utilizes a recurring update of a table of states and actions to learn how to accomplish a task. So the question arises, how is this so far an extension of what we have learned? So we haven't really utilized the reward, returned by an observation from an environment. And the idea of using a reward is absolutely critical for reinforcement learning and Q learning, which is classical Q-learning, will be our first algorithm to truly learn through reinforcement.

What we should know about reinforcement learning is that it doesn't necessarily require a neural network or even a deep neural network. It's simply the idea or requirement that you're going to be able to create an agent that is going to learn, based on a reward metric return from its observations or states of the environment.

Q Learning really gets to start by understanding the development of the study of how children and humans learn. And this was pioneered in the 1940s, in the 1960s by

John Piaget, who was a Swiss psychologist working on studying child development. He conducted small experiments designed to understand how children learn new skills. He ended up discovering that or defining four stages of intellectual development, such as sensory-motor development, preoperational development, concrete operational development, and then formal operational development, which is essentially your ability to conduct development in a cognitive manner from adolescence to adulthood.

Chris Watkins in the 1980s, discovered that he could apply his general findings to begin thinking of reinforcement learning as a form of incremental dynamic programming, and in 1989, he came out with his thesis called Learning from Delayed Rewards, which was later known as Q-learning. In 1992, Chris Watkins and Peter Diann published a critical article on Q-learning in the journal Machine Learning, demonstrating its convergence to optimal action values with a probability of one when all actions are sampled in all states and the action values are discretely represented.

Later in 2014, DeepMind patented an application of Q-learning to deep learning, titled “Deep Reinforcement Learning” or “Deep Q-Learning”. This paper explains that, if an environment has a discrete state space and a discrete set of actions, it is possible to create a grid or table that matches all possible actions at all possible states. Q is a function that takes in a state ‘s’ and an action ‘a’ and what it produces as output is your long-term expected reward value. So if we can map all possible actions and states as a table grid, we could assign an expected reward to each possible action, and each possible state. So in theory, after enough training, we would want to pick out the maximum long-term expected reward for taking any particular action, any particular state.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Preprocessing

To overcome the complexity and high-dimensionality of Atari 2600 games, The first step involves converting the RGB representation of the frames to grayscale and down-sampling the image to a size of 110x84 pixels (or 105x80 in some implementations) from the original size of 210x160 pixels. The down-sampling is done to reduce the number of pixels in the image while still maintaining the relevant information for gameplay.

Next, a cropping stage is applied to obtain a final image size of 84x84 pixels, which is a square input required for the GPU implementation of 2D convolutions used in the Deep Q-Network. This stage involves cropping a central region of the downscaled image to keep the relevant parts of the gameplay.

The final preprocessing step involves normalizing the image pixel values to the range of [0,1] and converting them to the uint8 data type to reduce memory usage. The rewards are clipped to a range of [-1,1] to limit the scale of error derivatives and allow the use of a single learning rate across multiple games. This step could affect the agent's performance as it cannot differentiate between rewards of different magnitudes.

Also, we can stack the last four preprocessed frames to produce the input to the Q-function to give the agent information about the recent history of the game. Overall, the preprocessing steps allow for more efficient training of the Deep Q-Network on Atari games and improve the agent's ability to learn and play the game at a superhuman level.

Model Definition

The input to the neural network is a preprocessed image with dimensions 84×84×4, obtained by converting the raw RGB frames to grayscale and down-sampling them to 110x80 followed by cropping a 84x84 region. The first hidden layer of the network is a convolutional layer that convolves the input image with 16 8×8 filters with a stride of 4. The output is then passed through a rectifier nonlinearity (ReLU). The second hidden layer is also a convolutional layer, but this time it convolves the output of the first layer with 32 4×4 filters with a stride of 2, again followed by a ReLU nonlinearity.

The final hidden layer is a fully connected layer consisting of 256 rectifier units. The output layer is a fully connected linear layer with a single output for each valid action. In this experiment, the researchers used the RMSProp algorithm with mini batches of size 32 for training the network.

Overall, the model architecture is designed to extract relevant features from the preprocessed images and predict the Q-values for each action given the current state of the game. The use of convolutional layers allows the model to learn spatial features of the game environment, while the fully connected layers enable it to learn more abstract features and combine them for accurate Q-value predictions. The choice of RMSProp and minibatches helps to stabilize the training process and improve convergence.

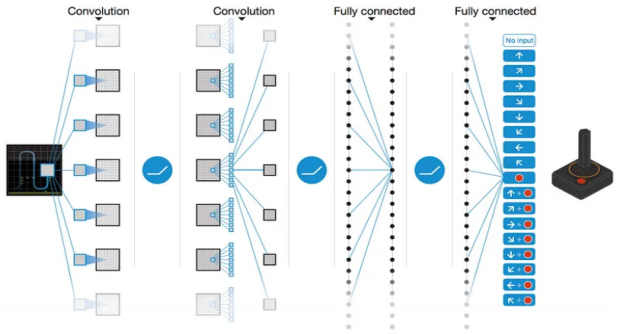

Figure 3. Convolutional Architecture

The DQN model described incorporates experience replay as a key method to prevent the network from diverging. The idea behind experience replay is to store relevant information (current state, next state, action taken, reward, and whether the next state is terminal) from each step of the game in a finite-size memory, rather than updating the model based on just the last step. During each Q-learning iteration, a batch of 32 elements is sampled from this memory and used to update the model.

DQN Model:

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 ϵ 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 $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

To ensure that the network does not forget what it's like to be in states that it hasn't seen in a while, we need to prefill the memory with a random policy up to a certain number of elements (50,000 in their case). This approach is helpful because successive states in reinforcement learning are often highly similar, making it easy for the network to forget about less frequently encountered states. Replaying experience ensures that the network is still showing old frames, helping to prevent this issue.

However, experience replay comes with a significant memory cost. Each preprocessed 110x80 frame takes a minimum of around 9KB to be stored, and with experience replay, the need to store a million of these frames, which amounts to 9GB of RAM. Therefore, it is important to be careful about how memory is used to prevent the DQN from constantly crashing. It is recommended to store frames using the np.uint8 type and converting them to floats in the [0, 1] range at the last moment, as using np.float32 will use 4x as much space and np.float64 (the default float type) will use 8x as much space.

To prevent unnecessary memory usage, it is also recommended to ensure that each unique frame uses the same memory in each state that it is a part of and not calling np.array() multiple times on a given frame or using copy.copy. They also suggest being careful when serializing memory, as most serialization libraries will create multiple copies of the same frame when deserialized.

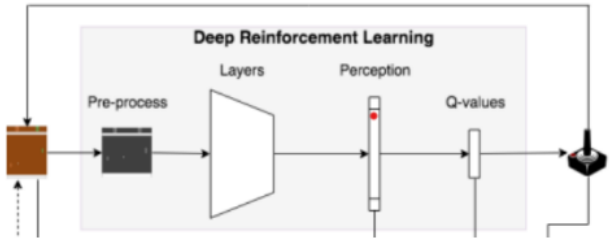

Figure 4. Deep Reinforcement Learning

Experiment

Now for the Atari game: Breakout that we're going to be training on, we need to make a consideration, is color itself actually important to playing the game? And this actually depends on the game. The DQN research and publication actually did learn on the color images. For the Atari breakout, it's probably not important to understand what colors are in the actual bars in that rainbow bar at the top. Therefore we convert the color channels to the grayscale implementation with a dimension of (84 x 84).

We now state that our observations are going to be the pixels in a sequence of frames. This is passing in a sequence of frames in order to understand what is happening in the environment. This sequence of frames is passed as a single experience, we store this experience that could be sampled for the agent to learn. Similarly, we slide to the next experience by fixing the number of frames as window length. The observations of the experience will now consist of a window of frames called a replay buffer.
From the collections framework, we import deque which allows us to keep reading the frames from the environment. This can also be done through Sequential Memory by limiting the number of experiences in the replay buffer and setting the window length.

Then we process each observation through the Image Processor inherited class from keras-rl. We then feed this processed image through the Convolutional 2D layer, using

the ReLU activation function for our neural network. The output layer is essentially our actions to be performed, which uses a linear activation function whose output matches the number of actions. Then we train our DQN agent by implementing an epsilon greedy policy manually, and we then recompile the model using Adam optimizer.

Future Research

First, unlike DQNs, target networks do not have the "chasing the target" issue. Without a set goal, the network's predictions can drastically change with each update, making convergence challenging. Target networks minimize the "chasing" effect and produce more stable updates by allowing a fixed target to be used for a set number of iterations. In the end, this results in faster convergence and better performance.

Second, Huber loss is a more reliable loss function for training DQNs than mean squared error (MSE). Mean absolute error (MAE) is not differentiable at zero, whereas mean standard error (MSE) can severely penalize large errors and slow learning. Huber loss offers a middle ground between MSE and MAE, penalizing large errors more severely than MAE but less severely than MSE while remaining globally differentiable. Faster convergence results from more reliable updates as a result.

Third, utilizing larger networks and giving them more training time is a tip for enhancing DQN performance. By adding an additional layer before the first dense layer and increasing the number of filters in each convolutional layer, the DeepMind team was able to double the size of the network. Additionally, they increased the quantity of units in the concealed dense layer. They trained the network for 50 million frames, five times as many as in the previous iteration, to handle the larger network.

Overall, using target networks, Huber loss and Bigger Networks, Longer Training can greatly improve the performance and training efficiency of DQNs.

References

[1] Li, Y. 2018. Deep reinforcement learning: An overview. arXiv preprint. arXiv:1701.07274 [cs.LG]. Ithaca, NY: Cornell University Library.

[2] Mostafa D. A; and Howard M. S. 2016. Exponential moving average based multiagent reinforcement learning algorithms. *Artificial Intelligence Review* 45 (1): 299-332.

[3] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Andrei A. R. et al. 2015. Human-level control through deep reinforcement learning. *NATURE* 518(1): 529–533. doi:10.1038/nature 14236.

[4] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs.LG]. Ithaca, NY: Cornell University Library.

[5] Jang, B.; Kim, Myeonghwi, K.; Harerimana, G.; and Kim, J. W. 2019. Q-learning algorithms: A comprehensive classification and applications. *IEEE access* 7 (1): 133653-133667.

[6] Laskin, M.; Lee, K.; Stooke, A.; Pinto, L.; Abbeel, P.; and Srinivas A. 2020. Reinforcement learning with augmented data. *Advances in neural information processing systems* 33(1): 19884-19895.

[7] Machado, M. C.; Bellemare, M. G.; Talvitie, E.; Veness, J.; Hausknecht, M.; and Bowling, M. 2017. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. arXiv:1709.06009 [cs.LG]. Ithaca, NY: Cornell University Library.