HW2: Understanding Reinforcement Learning

Archisman Ghosh

CSE 584.

Contributing authors: apg6127@psu.edu;

Abstract

In this report, we focus on a reinforcement learning-based solution to the Atari Breakout game using Deep Q-Learning (DQN), where an AI agent learns to play by interacting with the game environment. The agent observes game frames and selects actions—such as firing the ball or moving the paddle left or right—based on predicted Q-values generated by a convolutional neural network (CNN). The model stabilizes training and mitigates instability issues by employing experience replay and a target network. The objective of the agent is to maximize its score by breaking bricks while preventing the ball from falling.

Keywords: Reinforcement Learning.

1 Problem

The problem at hand is to train an AI agent to play Breakout by Atari, a classic arcade game where the player controls a paddle to bounce a ball towards a wall of bricks, breaking them for points. The goal is for the agent to learn how to efficiently break all the bricks by selecting optimal actions that keep the ball in play. The game provides visual frames (210×160 RGB images) as input, and the agent must choose from four possible actions: no-op (do nothing), fire (launch the ball), move left, and move right. The performance is evaluated based on the cumulative reward, which increases as more bricks are broken.

To achieve this, the agent must learn how to act in various game states—deciding when to move the paddle, which direction to move, and when to fire—by understanding the game dynamics. A major challenge is that the state space is vast (all possible visual frames) and the reward is sparse, making the learning process difficult.

A reinforcement learning (RL) approach is best suited for this task because RL algorithms, like Deep Q-Learning, can handle complex environments where actions influence long-term outcomes, and reward feedback is delayed, allowing the agent to discover strategies through exploration and experience [1].

2 Solution

We discuss the solution and implementation by the authors in [2].

2.1 Overview

The solution uses Deep Q-Learning (DQN) to train an agent to play Breakout by mapping game states (visual frames) to optimal actions. The code processes raw game frames by converting them to grayscale and resizing them to reduce computational complexity. The agent utilizes a convolutional neural network (CNN) to extract features from the frames and estimate Q-values for each possible action.

To improve stability, the DQN algorithm incorporates two key techniques: experience replay and target networks. Experience replay stores the past experiences and samples mini-batches to break the correlation between consecutive frames, aiding in efficient learning. A separate target network, updated less frequently, is used to compute target Q-values, preventing large oscillations during training. The agent follows an epsilon-greedy policy to balance exploration (random actions) and exploitation (using learned strategies). By minimizing the loss between predicted and target Q-values, the agent progressively learns optimal strategies to maximize its performance in the game.

2.2 A deep-dive

In this subsection, we take a look at the details of the implementation of the DQN algorithm.

2.2.1 Initialisation and State Processing

The OpenAI library `gym`, provides the *Breakout-v0* environment. Four actions, $[0 : null, 1 : fire, 2 : left, 3 : right]$ are defined as the possible actions for the agent, and the state processor class pre-processes the game frames (converting to grayscale and resizing to 84×84 pixels) to make them trainable.

2.2.2 Q-Value Estimator

The `Estimator` class builds a CNN to estimate the Q-value for each action, taking four consecutive game frames of size 84×84 pixels stacked as input to provide a temporal context about the movement of the ball. The first convolutional layer applies 32 filters of size 8x8 with a stride of 4, reducing the spatial dimensions while extracting basic visual features. The second layer applies 64 filters of size 4x4 with a stride of 2, further refining the extracted features. The third convolutional layer uses 64 filters of size 3x3 with a stride of 1 to capture fine details in the input. After these convolutional layers, the output is flattened into a 1D vector and passed through a fully connected layer of 512 units, which connects to the final output layer. A more detailed description is provided in Table 1.

Layer	**Type**	**Input Dim.**	**Output Dim.**	**Kernel/Stride**
1	Input	84x84x4	84x84x4	N/A
2	Convolutional 1	84x84x4	20x20x32	8x8 / 4
3	Convolutional 2	20x20x32	9x9x64	4x4 / 2
4	Convolutional 3	9x9x64	7x7x64	3x3 / 1
5	Flatten 1D	7x7x64	3136	N/A
6	Fully Connected	3136	512	N/A
7	Output	512	4	N/A

Table 1: CNN Architecture for Q-Value Estimation in DQN

The output of the final layer of the CNN shows the prediction for the Q-values, which can be formalized as

$$Q(s, a) = CNN(s)$$

where, $Q(s, a)$ be the predicted Q-value, s be the current state, and a, be the corresponding action. We also observe that an epsilon-greedy action selection scheme is used during the exploitation and exploration phase of the algorithm. During the exploitation phase, the agent maximizes the expected future reward, $a = maxQ(s, a')$, and the agent randomly selects an action with some probability ϵ to prevent the network from converging too early to a suboptimal policy.

The Q-values are updated by the Bellman equation which expresses the value of a state-action pair as the sum of the immediate reward and the discounted value of the best action of the best state.

$$Q_{target}(s, a) = r + \gamma \cdot maxQ_{target}(s, a')$$

where, r be the reward on choosing action a, γ be the discount factor (~ 1), and s' be the following state. The Q-network implementation has a squared loss function to minimize the difference between the predicted Q-values and the target Q-values.

$$loss = \frac{1}{N}\sum_{i=1}^{N}(Q(s_i, a_i) - Q_{target}(s_i, a_i))^2$$

The loss is optimized by an RMSProp algorithm during the training phase. The Q_{target} values for the loss function are calculated by a separate target network (same architecture as the Q-network), that helps to stabilize training by fixing the target Q-values for a few iterations. This is done by updating the target network less frequently.

2.2.3 Training

The training loop follows an iterative process where the agent interacts with the environment to learn optimal actions. At each step, the agent starts by selecting an action using an epsilon-greedy policy, which balances exploration (random actions) and exploitation (choosing the best-known action). The environment then responds by providing the next state, a reward, and an indication of whether the episode has ended. The agent stores this experience (state, action, reward, next state) in a replay memory buffer. Periodically, the agent samples a mini-batch of experiences from this buffer to train the Q-network, calculating the target Q-values using the Bellman equation. The weights of the network are updated by minimizing the difference between the predicted Q-values and the target values using a loss function. Additionally, a target network is updated at regular intervals to stabilize the training process, ensuring more consistent learning over time. This loop continues until the agent becomes proficient at playing the game. Table 2 shows details of the training parameters.

Hyperparameter	**Typical Value**
Discount Factor (γ)	0.99
Learning Rate	0.00025
Replay Memory Size	1,000,000
Mini-batch Size	32
Target Network Update Frequency	10,000
Initial Epsilon (ϵ)	1.0
Final Epsilon (ϵ)	0.1
Epsilon Decay Rate	1,000,000
Replay Start Size	50,000
Max Steps per Episode	50,000
Optimizer	RMSProp
RMSProp Learning Rate	0.00025
RMSProp Decay	0.99
RMSProp Epsilon	1e-6
Frame Skipping	4
Stacked Frames	4

Table 2: Hyperparameters of the Deep Q-Learning Algorithm

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with Deep Reinforcement Learning (2013). https://arxiv.org/abs/1312.5602

[2] Britz, D.: Deep Q Learning Solution. https://github.com/dennybritz/reinforcement-learning/blob/master/DQN/Deep%20Q%20Learning%20Solution.ipynb. Accessed: 2024-10-14 (2017)