Reinforcement Learning with Neural Networks: Playing Atari Games using Deep Q-Networks

Student:
Eng. Cătălin-Aurelian Ciocîrlan

University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology

February 2024

Table of contents

Introduction

Reinforcement learning

Definition

- Type of machine learning where an agent learns to make decisions by interacting with an environment
- Aims to maximize cumulative rewards through trial and error, using feedback from the environment to improve decision-making over time

Applications

- Gaming
- Robotics
- Finance
- Healthcare

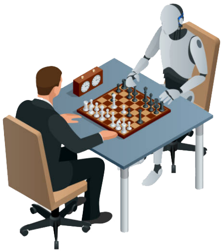

Figure 1: Illustration of RL agent

State of the Art

Deep Q-Network (DQN) [1] [2]

- Combines deep neural networks with Q-learning to learn optimal strategies in reinforcement learning tasks with discrete action spaces
- Approximates an action-value function by mapping states to the expected cumulative rewards of taking each action
- Through experience replay and target networks, DQN efficiently learns from past interactions to improve decision-making over time

Deep Deterministic Policy Gradient (DDPG) [3]

- Actor-critic algorithm extended from DQN for continuous action spaces
- Simultaneously learns a deterministic policy (actor) and a value function (critic) using deep neural networks
- Employs off-policy learning and target networks to stabilize training and efficiently learn complex control tasks

State of the Art

Double DQN [4]

- Extension of DQN that addresses overestimation bias in action values
- Uses two separate networks to decouple action selection and value estimation, resulting in improved performance

N-Step DQN [5]

- Variant of DQN that updates the Q-values using N-step returns
- Combines multiple intermediate rewards from consecutive time steps to improve the efficiency of learning

Dueling DQN [6]

- Extension of the DQN that separates the value estimation into two streams: one for the state value and one for the action advantages
- Allows the agent to learn the value of being in a particular state and the advantage of taking a specific action within that state independently, leading to more efficient and stable learning

Theoretical overview

Q-learning

Task

- An agent interacts with an environment $\mathcal{E}$ by operating at every timestep t on the current sequence of observations $\mathbf{s_t}$, taking action $\mathbf{a_t}$, receiving a reward $\mathbf{r_t}$ and changing the observations to $\mathbf{s_{t+1}}$
- The strategy used for choosing the action is called policy π
- The goal is to maximize the discounted cumulative return at time t

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_t, \tag{1}$$

where γ represents a discount factor and T is the final timestep

Q-learning

Q function

- In order to evaluate the quality of the actions taken by the agent we use an action-value function Q, which estimates the long-term rewards associated with each action in a given state
- The optimal action-value function is the maximum expected return achievable by following any strategy

$$Q^*(s;a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right] \tag{2}$$

Bellman equation

- Q^* follows an important identity known as the Bellman equation

$$Q^*(s;a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma Q^*(s', a') | s, a\right] \tag{3}$$

- **Intuition**: if the optimal value $Q^*(s', a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximising the expected value of $r + \gamma Q^*(s', a')$

Q-learning

Q-networks

- In theory, one could estimate the action value function by using the Bellman equation as an iterative update, $Q_{i+1}(s; a) = \mathbb{E}\left[r + \gamma Q_i(s', a')|s, a\right]$, which would converge to $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$
- In practice, it is common to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$, such as a neural network with the weights θ called Q-network

Q-learning

Learning

- The network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i)\right], \tag{4}$$

  where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a named *behaviour distribution*
- The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function
- The algorithm is *model-free* and *off-policy*
- **Exploration vs. exploitation**: the behaviour distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy dictated by the model with probability $1-\epsilon$ and selects a random action with probability ϵ

Experimental setup

Atari environments

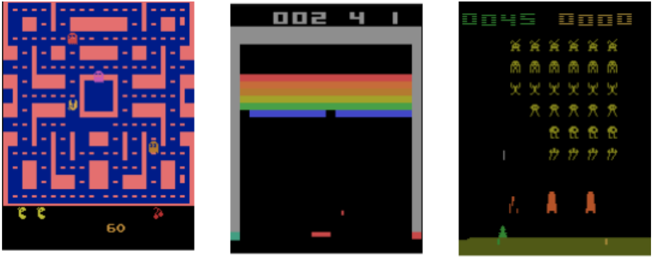

Figure 2: Studied Atari Games. From left to right: Ms. Pacman, Breakout, Space Invaders

Dataset

Dataset

- We utilize a technique known as experience replay, where we store the agent's experiences at each time-step, $e_t = (s_t, a_t, s_{t+1}, r_t)$ in a dataset $\mathcal{D} = \{e_1, \ldots, e_N\}$, pooled over many episodes into a replay memory
- During the inner loop of the algorithm, we apply minibatch updates to samples of experience, $e \sim \mathcal{D}$, drawn at random from the pool of stored samples
- **Advantages**: data efficiency, reduced correlation between data samples, more stable training process

Models

Models

- 5 fully connected (FC) models, implemented using PyTorch, that use RAM observations as input
- 1 convolutional neural network (CNN), implemented using StableBaselines3, that use raw images from the environments as input

Model \Layers	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
FC1	FC 64	FC 32	FC k	-	-
FC2	FC 256	FC 64	FC k	-	-
FC3	FC 128	FC 64	FC 32	FC k	-
FC4	FC 512	FC 128	FC 32	FC k	-
FC5	FC 512	FC 128	FC 64	FC 16	FC k
CNN	Conv 32x8x8, stride 4	Conv 64x4x4, stide 2	Conv 64x3x3, stride 1	FC 512	FC k

Table 1: Architecture of all trained models. The size of the action space is denoted by k. Fully connected layers are expressed by the number of neurons they contain, while the convolutional layers by the shape and stride of the kernel

Training details

Training details

- For the FC models, the input is a 1D vector of 128 elements, while for the CNN the inputs are RGB images of 210 x 160 px
- Separate target network instead of using previous weights to compute the targets y_i. This network is optimized via soft update using a hyperparameter $\tau = 0.005$
- Adam optimizer, learning rate of $1e - 4$, batch size of 128
- Huber loss
- $\gamma = 0.99$
- ϵ-greedy policy starting with an exploration probability of 0.9 and exponentially decays to its final value of 0.05 over roughly the first 10% of the training process
- The FC models were trained for 1K episodes, with an additional training for the first model for 10K games
- The CNN model was trained for 1M and 10M time-steps model weights
- Maximum memory size of 10K samples

Algorithm

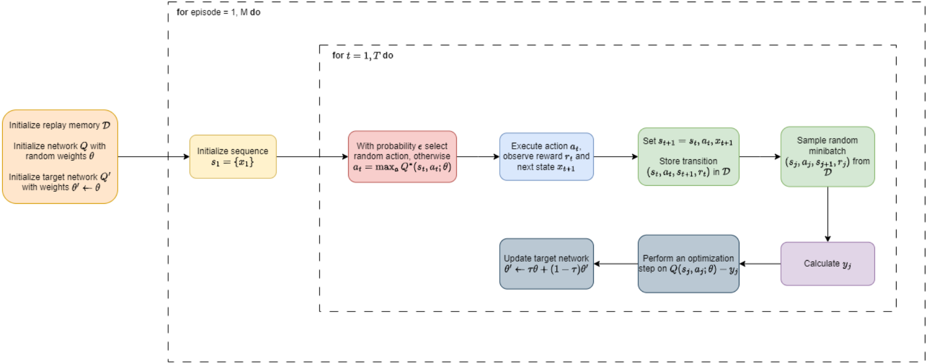

Figure 3: Illustration of our DQN algorithm

Performance evaluation

Training

Training evaluation

- Current episode score
- Moving average of the scores from the past 100 games

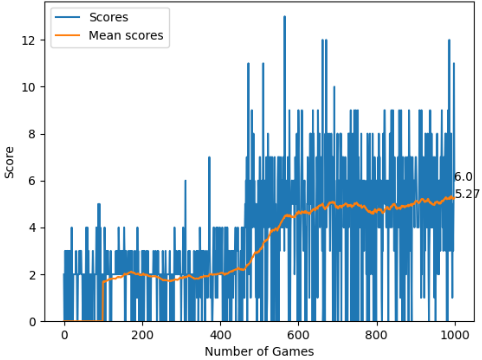

Figure 4: Training curves for the first fully connected model, in the case of Breakout

Testing results

Agent	Score	Ms. Pacman	Breakout	Space Invaders
Random Play [2]	avg	307.3	1.7	148
FC1	best	2730	15	945
	avg (± std)	573.5 (± 323.5)	4.3 (± 2.3)	272.5 (± 133.1)
FC1-10K	best	2500	**24**	1070
	avg (± std)	772.9 (± 437.7)	**10.3 (± 3.7)**	**379.6 (± 175.1)**
FC2	best	4000	19	935
	avg (± std)	1267.5 (± 714.9)	3.8 (± 2.0)	278.7 (± 140.0)
FC3	best	3600	18	980
	avg (± std)	497.2 (± 353.0)	6.2 (± 2.6)	253.0 (± 126.7)
FC4	best	4180	21	**1150**
	avg (± std)	1401.0 (± 508.6)	6.7 (± 2.5)	357 (± 138.3)
FC5	best	2270	17	845
	avg (± std)	885.3 (± 270.5)	7.3 (± 2.4)	269.6 (± 114.1)
CNN	best	4100	10	660
	avg (± std)	1252.6 (± 514.7)	1.5 (± 1.4)	138.7 (± 110.2)
CNN-10M	best	**6620**	16	760
	avg (± std)	**2086.3 (± 739.4)**	5.5 (± 2.2)	188.0 (± 117.3)
Mnih *et al.* 2015 [2]	avg (± std)	2311 (± 525)	401.2 (± 26.9)	1976 (± 893)

Table 2: Results for all the trained models

Demo

A demo of our algorithm can be viewed using this link

Conclusions

Conclusions

Conclusions

- Explored application of DQN models to classic Atari games for high-score achievement via deep Q-networks
- Evaluated various neural network architectures, including FC models and CNNs, across Ms. Pacman, Breakout, and Space Invaders
- Noted relevance of input data, particularly influential in Breakout and Space Invaders, impacting environment adaptation
- For Ms. Pacman achieved an average score close to the original study

References

Mnih et al.
Playing atari with deep reinforcement learning
arXiv preprint arXiv:1312.5602

Mnih et al.
Human-level control through deep reinforcement learning
Nature Publishing Group, volume 518, pages 529 - 533, 2015

Lillicrap et al.
Continuous control with deep reinforcement learning
arXiv preprint arXiv:1509.02971

Van Hasselt et al.
Deep reinforcement learning with double q-learning
Proceedings of the AAAI conference on artificial intelligence, volume 30, 2016

Mnih et al.
Asynchronous methods for deep reinforcement learning
International conference on machine learning, pages 1928 - 1937, 2016

Wang et al.
Dueling network architectures for deep reinforcement learning
International conference on machine learning, pages 1995 - 2003, 2016

Thank you for your attention!