Playing Atari with Deep Reinforcement Learning

1 Introduction

In this paper, we study the first deep reinforcement learning model that was successfully able to learn control policies directly from high dimensional sensory inputs, as applied to games on the Atari platform [1]. This first model was a convolutional neural network (CNN) that takes in raw pixels as input and outputs the value function Q estimating future rewards. This value estimate is then used to decide on the best action to take at the current time step. Back when this paper was written, deep nets were only commonly applied to supervised learning problems, where there is a corresponding label for every classification of the image; this was the first approach to show that it is possible to train a CNN even with sparse "rewards" and to solve the "credit assignment problem" [2]. This model was also shown to be able to generalize across multiple games [1]: the same architecture was shown to perform decently on six Atari 2600 games in the Arcade Learning Environment, and even surpasses a human expert on three of them.

However, this approach has a major flaw in that the deep network used to approximate the Q function gives a biased estimate: there is an overestimation caused by taking maximum estimated values in the Bellman equation [3] [4]. Besides, this variant of Q-learning also involves bootstrapping, which requires learning estimates from estimates. This makes the overestimation problem even more severe and can even cause divergence in the worst case. It has also been shown that the first DQN mentioned above suffered from substantial overestimations in some games in the Atari 2600 domain [3].

1.1 Double Q Learning & Alternatives

Hence, in this paper, we also look into double Q-learning and its variants, which aim to reduce the aforementioned overestimation bias. Although these methods still do not completely eliminate the bias, they produce significant improvements over baseline algorithms for the same Atari benchmark games, while also demonstrating further stabilization in the training phase and better derived policies [3].

2 Related Work

2.1 TD-Gammon

TD-Gammon was the first reinforcement learning algorithm that achieved success in playing games, and is considered the seminal work in this field [5]. TD-Gammon plays backgammon, a board game that involves both luck and skill, and it works based on temporal-difference learning. It achieved master-level play starting from zero initial knowledge.

Temporal difference works by updating its estimates of the value function as it steps through the environment, based on its observations. A useful analogy to think about is if the weather on Sunday depends on the weather from Monday to Saturday, even if it is only just Friday today we can already get a pretty good estimate on what Sunday might look like based on our previous datapoints. This is more powerful than a related technique called Monte Carlo methods, which can only make a prediction once all the datapoints are available [6].

2.2 AlphaGo

AlphaGo was the first program to beat a professional human Go player. Most famously, AlphaGo defeated the 9-dan Lee Sedol in a five-game match in 2016, the first time a 9-dan player has been beaten. AlphaGo primarily uses Monte Carlo Tree Search (MCTS) with a policy network, which is a form of reinforcement learning [7].

MCTS tries to estimate the value function by sampling many paths in the game tree in order to guess which paths might be the best. This is done by balancing between exploration (i.e exploring new paths) and exploitation (i.e going down paths that we already know to be good) [7]. This allows it to tackle games with large state spaces like Go [8]. The results from sampling are used to build its own game tree via backpropagation.

2.3 Dota 2

More recently, OpenAI Five defeated the reigning world champions in Dota 2 in 2019. OpenAI used deep reinforcement learning and also used MCTS [9]. The novelty factor was in scaling existing reinforcement learning algorithms to new heights, allowing it to be trained continuously on thousands of GPUs over a period of 10 months.

Check Your Understanding (1) *AlphaGo and Dota 2 both relied heavily on MCTS. Why might that be the case? What similarities are there between Go and Dota 2?*

3 Background

3.1 Arcade Learning Environment & Related Problem Definitions

Firstly, we define the environment $\mathcal{E}$, which is the Atari emulator. At each time step t, the agent can select an action a_t from a set of legal game actions, $\mathcal{A} = \{1, ..., K\}$.

This setup is considered a **partially observable** environment: the agent only observes an image produced by the emulator $x_t \in \mathbb{R}^d$ which is a vector of raw pixel values from the screen. The agent is unable to obtain any information about the internal state of the emulator which decides the game score and further changes of the environment.

We consider the history s_t, a sequence of actions and corresponding observations (which are the images on the screen).

$$s_t = x_1, a_1, x_2, a_2, ...a_{t-1}, x_t$$

The goal of the agent is to maximize future rewards, which is represented as the game score in this setting. We define the *discounted return* at time t, R_t, as follows

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

where γ is the the discount factor and T is the terminal time step.

We define the optimal action value function $Q_*(s, a)$ as a function which gives the maximum expected returns achievable, given the history s and the action a to be taken at the current time step. This Q_* can be expressed recursively as follows

$$Q_*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q_*(s', a') | s, a \right] \tag{1}$$

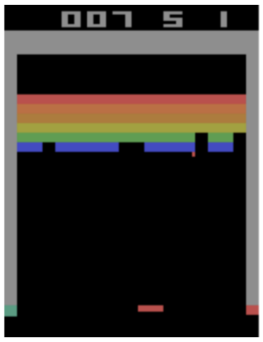
3.2 Deep Q-learning

The goal of most reinforcement learning algorithms is to estimate the action value function using an approximator $Q(s, a; \theta) \approx Q_*(s, a)$. Q-learning algorithms use linear as well as non-linear functions as approximators for Q, such as deep neural networks [1]. In this case, θ refers to the weights and parameters in the neural network.

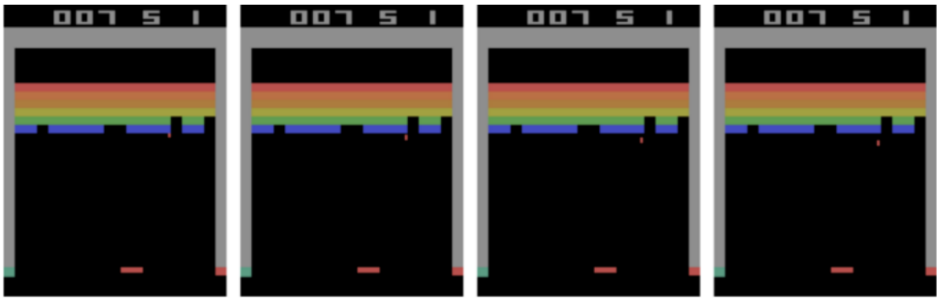
This Q-network is trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i of gradient descent.

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim p(\cdot)} \left[(y_i - Q(s, a; \theta_i))^2 \right]$$

We interpret this as just a least squares expected loss function of the form similar to $L_i(\theta_i) = \mathbb{E}\left[(y_i - f(x_i))^2\right]$ at each iteration i. In this definition, y_i, is the desired "output", which is the



(a) Single Frame



(b) Multiple Frames

Figure 1: Extracting temporal cues via stacking frames

supposed correct future rewards estimation. We obtain y_i using bootstrapping, as follows.

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$$

The context here is that we have already simulated the game to the current state and hence, know the true value of current reward, r, to be obtained after action a is chosen. s' is the state that is obtained after executing a in the state s. We then use the Q approximator obtained at the previous iteration iteration $i-1$ to estimate the future rewards for all legal actions a' at the state s', and maximizing this value.

4 Training Deep Q Networks

4.1 Experience Replay

An interesting technique that contributed largely to the success of this approach is experience replay [10], a method used to train this Q-network.

We define an experience at each time step t as $e_t = (\phi_t, a_t, r_t, \phi_{t+1})$. Further, we define $\phi_t = \phi(s_t)$, where we have ϕ a function that gives a fixed length representation of an arbitrarily long sequence s, so that it can be more easily handled by a neural network.

For example, in this specific implementation of the DQN for Atari, ϕ takes in the last 4 screen image frames and stacks them as input for the neural network, as given in 1(b). Given a screen image size of 84 x 84, this ϕ just stacks the images to create an 84 x 84 x 4 input. This allows us to capture more subtle information like direction of movement and velocity of target objects on screen, besides also minimizing variances like flickering of projectiles.

We then pool multiple of these experiences into a dataset $\mathcal{D} = e_1, ..., e_N$. The general idea here is that we randomly sample minibatches from this dataset as input (without replacement) during each iteration of gradient descent. This has two main advantages:

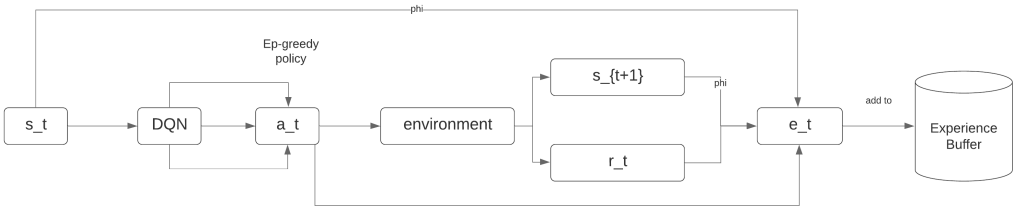


Figure 2: Brief overview of how the experience replay training is done

1. Data efficiency: Each experience e_t can be potentially re-used in many iterations, and allow for better data efficiency
2. Randomization: The randomization allow us to break correlations between consecutive samples. This allows for better convergence behaviour because the data is more like i.i.d. data which is assumed in most supervised learning convergence proofs. This is similar to how we shuffle input data in supervised learning used during training.

4.2 ϵ-greedy policy

Besides, another popular technique used during the training of the model is the *ϵ-greedy action selection* [11]. This is a simple method to balance exploration and exploitation: where ϵ refers to the small probability of time which we select a completely random action to take for "exploration". Otherwise, for the other $1 - \epsilon$ probability of times, we simply select the best action which maximizes the Q estimates.

Exploration-exploitation dilemma:
The agent can either

1. choose a policy, evaluate it, and move on to better policies (exploitation)
2. collect more information and use it simultaneously to construct a better policy (exploration)

Check Your Understanding (2) *How is Q-learning different from other reinforcement learning algorithms? What are the possible advantages/ disadvantages?*

4.3 Training Algorithm

In the previous two sections we described two interesting attributes of the training approach that makes this first DQN successful. Now we briefly summarize the training algorithm which is illustrated in Figure 2 [1], and also fill in the specific details, also specifically in terms of how the dataset of experiences is constructed.

For each episode m, we firstly initialize a sequence $s_1 = x_1$ and $\phi_1 = \phi(s_1)$.
For each iteration, t, we first populate and add to the dataset, before we do the gradient update, as described below:

Populating Dataset $\mathcal{D}$

1. Select $a_t = \begin{cases} \text{any random action} & \text{with probabilty } \epsilon \\ \arg\max_a Q_*(\phi(s_t), a; \theta) & \text{with probability } 1 - \epsilon \end{cases}$

   This is the effect of the ϵ-greedy policy.
2. We execute action a_t on the previous screen x_t. We are then able to observe the reward r_t and the new screen x_{t+1}. With this, we can find $\phi_{t+1} = \phi(s_{t+1})$.
3. Now we write $e_t = (\phi_t, a_t, r_t, \phi_{t+1})$ and we add this new experience into the dataset $\mathcal{D}$

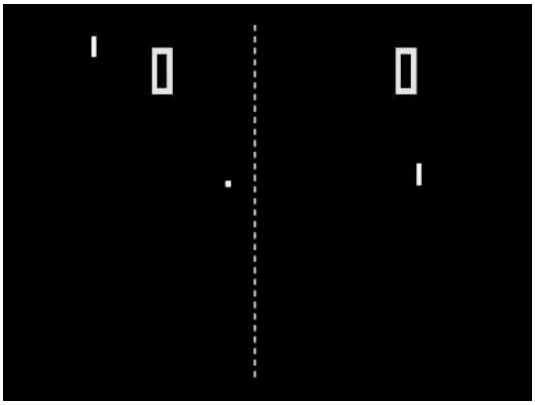


Figure 3: A game of Pong shown at the start, with both player's scores 0

4. Set $\epsilon = c\epsilon$ where $c < 1$ is a decay factor

Check Your Understanding (3) *Why do we decay ϵ? What if we do not decay it?*

Updating the Gradient

1. We sample a random mini-batch of experiences from the dataset $\mathcal{D}$.
2. Perform a gradient descent step using $(y_j - Q(\phi_j, a_j; \theta))^2$, where j is an index for items in the minibatch.

5 Experiments

5.1 Framework

We use the DQN algorithm described previously and trained it on Atari Pong using the OpenAI Gym framework. Gym is an open-source toolkit for developing and evaluating reinforcement learning algorithms [12]. We used the `PongNoFrameskip-v4` environment for the classic Atari game Pong that performs no frame skips with 0 repeat action probability. Frame skipping is a technique to sample only every kth frame in order to introduce stochasticity into the game environment. In order to simulate the Atari environment, Arcade Learning Environment (ALE) that is powered by the Stella Atari Emulator is run under the hood [13].

5.2 Pong

A game of Pong is played between a player and the computer similar to ping pong. Each side has a paddle that can be moved vertically, and the goal is to use the paddle to return the ball to the other side. When the opponent fails to return the ball, the other player wins a point. The first side to reach 21 points wins.

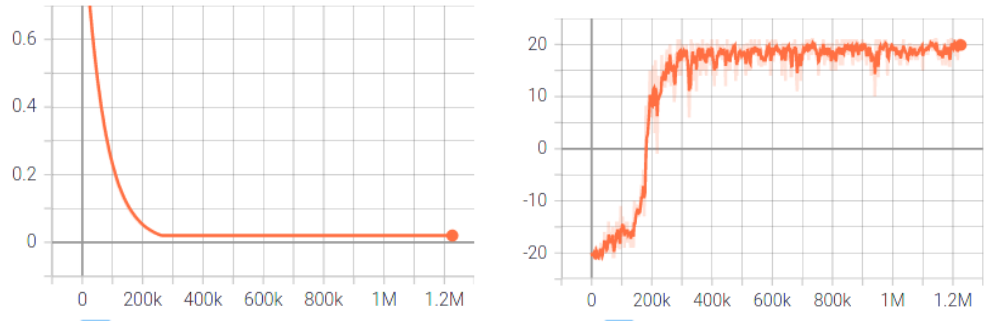
In our reinforcement learning formulation, winning a round gives 1 reward, while losing gives -1. We trained the model until it could reach reward of 19 [14].

5.3 Results

Figure 4 shows the results of training our DQN on Pong (see companion code in [14]). We trained for a total of 661 games that produced 1226284 frames. Later games required more frames as the AI got better and games started taking longer.

In Figure 4(b) we see that we initially start with a reward of -20 (i.e losing every game), to be eventually close to beating the computer on almost every round.

We decay ϵ by 0.999985 each round (verify this in our provided code in `pong.ipynb`), and the value of ϵ over time is plotted in Figure 4(a). If we recall our training algorithm, at each step we choose a



(a) Epsilon (b) Reward

Figure 4: Epsilon and Reward over Training Frames

random action with probability ϵ, and the best known action with probability $1 - \epsilon$. We can therefore see that the initial randomness allows for exploration and allows us to find strategies to improve our reward, and over time after we have a good sense of which actions to take we also decay ϵ and therefore our strategies become more deterministic.

5.4 Limitations of DQN

One significant downside of DQN is that it is prone to significantly overestimating Q-values. To understand why, revisit the definition of Q_* in equation (1).

Check Your Understanding (4) *Based on Equation 1, why does DQN frequently result in an overestimation?*

This overestimate may cause the agent to repeatedly explore states that it believes is promising at the expense of exploring other states. This results in unstable training rounds and a poor estimate of the Q-values being learnt. This can be overcome by decoupling the choice of the agent's action from the learning of the Q-values, by using two different networks. The resulting strategy is called Double Q-Learning (DDQN).

Check Your Understanding (5) *Consider a state s in which all the true optimal action values are equal at $Q_*(s, a) = V_*(s)$ for some $V_*(s)$. The estimation errors can be expressed as $\epsilon_a = Q_t(s, a) - V_*(s)$. Let these estimation errors be uniformly random in $[-1, 1]$. Given that m is the number of legal actions available, what is the overoptimism, $\mathbb{E}[\max_a Q_t(s, a) - V*(s)]$?*

5.5 Double DQNs

A well known approach to reducing this overestimation bias is using two separate and independent Q-value estimators, each of which is used to update the other. In fact, with proper assumptions, DDQNs have even been proven to be give underestimates rather than an overestimates [3]. However, in practice, obtaining two independent Q-value estimators is usually impractical especially for large-scale tasks, and causing DDQNs to still suffer from overestimation [3]. To address these empirical concerns, the clipped DDQN variant was proposed, which takes the minimum over two value estimations.

6 Conclusion

Reinforcement learning is a powerful technique that for agent learning that is increasingly being applied successfully to more games. We have been able to replicate the success in groundbreaking papers in this field, and we anticipate more promising results to come.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[2] Benjamin James Lansdell, Prashanth Ravi Prakash, and Konrad Paul Kording. Learning to solve the credit assignment problem, 2020.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[4] Chris Yoon. Double Deep Q Networks, Jul 2019.

[5] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[6] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[7] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'08, page 216–217. AAAI Press, 2008.

[8] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[9] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[10] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay, 2020.

[11] Michel Tokic. Adaptive epsilon-greedy exploration in reinforcement learning based on value differences. *KI 2010: Advances in Artificial Intelligence Lecture Notes in Computer Science*, page 203–210, 2010.

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[13] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

[14] Various. Github Repository fanpu/dqn-pong, Deep-Q Learning on Pong.

Solutions for Exercises

(1) Both games are similar in that there is an exponential number of possible states in the game, and therefore there is no way to explore all of them. Monte Carlo Tree Search helps by stochastically choosing paths which are more promising, and therefore implicitly prunes the state space that must be explored.

(2) Q-learning is an example of an off policy learning algorithm which learns the value of the optimal policy independently of the agent's actions/ motivations (even if the actions to explore are chosen randomly). It is also commonly compared to other on-policy algorithms like Value Iteration, Policy Iteration (which are mentioned in class), Sarsa etc. We briefly compare these two categories of learning algorithms as follows:

Off policy learning algorithms evaluate and improve a policy that is different from the actual policy used for action selection.

1. Gather information from (partially) random moves
2. Evaluate states with the greedy policy
3. Slowly reduce randomness
4. **Disadvantages:** May become trapped in local minima

On policy learning algorithms improve the same policy which is being used to select actions.

1. Start with a simple soft policy
2. Sample state space with this policy
3. Improve policy
4. **Disadvantages:** May be slower because of the need to do exploration, but this means it is more flexible if alternative routes appear.

(3) We decay ϵ so that we can initially set ϵ to a high value to encourage exploration, but after time, as we learn more about our environment and have a better Q^*, we will start converging on our optimal solution and do less random exploration. If we do not decay ϵ, we cannot have a high ϵ to begin with or we will be stuck with a strategy that is still quite randomized.

(4) Consider an example where we have a state s where the true value of Q for all actions equal 0, but the estimated Q values are distributed both above and below zero. Because we are maximizing the Q estimates in the term $\max_{a'} Q_*(s', a')$, we use the maximum Q estimate, which is bigger than zero to update the Q function. This systematically leads to an overestimation of Q values. While this did not manifest during our training run, it has been observed in practice during training in other games.

(5) The probability that $\max_a Q_t(s, a) \leq x$ is equal to the probability that $\epsilon_a \leq x$ for all a. Because the estimation errors are independent,
$P(\max_a \epsilon_a \leq x) = P(X_1 \leq x \wedge X_2 \leq x \wedge \cdots \wedge_m \leq x) = \prod_{a=1}^{m} P(\epsilon_a \leq x)$

$$P(\epsilon_a \leq x) = \begin{cases} 0 & 0 \text{ if } x \leq -1 \\ 1 & 1 \text{ if } x \geq 1 \\ \frac{1+x}{2} & \text{otherwise} \end{cases}$$

and hence

$$P(\max_a \epsilon_a \leq x) = \begin{cases} 0 & 0 \text{ if } x \leq -1 \\ 1 & 1 \text{ if } x \geq 1 \\ (\frac{1+x}{2})^m & \text{otherwise} \end{cases}$$

This gives us the CDF of the random variable $\max_a \epsilon_a$
We can get the PDF of $\max_a \epsilon_a$ by taking the derivative as follows,
$f_{max}(x) = \frac{d}{dx} P(\max_a \epsilon_a \leq x) = \frac{m}{2}(\frac{1+x}{2})^{m-1}$.

Hence, this gives us
$\mathbb{E}[\max_a \epsilon_a] = \int_{-1}^{1} x f_{max}(x) dx = \int_{-1}^{1} x \cdot \frac{m}{2}(\frac{1+x}{2})^{m-1} = \frac{m-1}{m+1}$