CS-464 Term Project Final Report Breakout Atari™ Game with Reinforcement Learning

Group 4
Abdullah Arda Aşçı (21702748), Alim Toprak Fırat (21600587),
Atahan Yorgancı (21702349), Tuna Alikaşifoğlu (21702125)

I. Introduction

Breakout is an arcade game that was published and developed by Atari, where the player is in control of a paddle on the x-axis and tries to break down each brick by handling a ball. At each time the ball touches to a brick, that particular brick is destroyed and the player gains a point, the game is won if all the bricks are destroyed. A sample position from the game is provided in Figure 1.

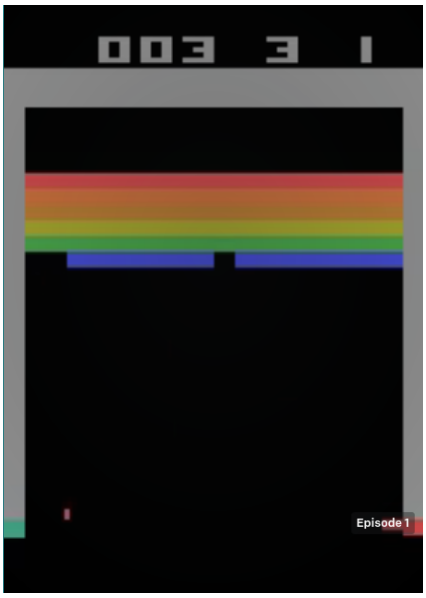


Fig. 1: Sample Position from the Breakout Atari Game [1]

In order to obtain a model that can play the game and get better results than a random agent with the use of Supervised or Unsupervised Learning, a large amount of data needs to be generated. Reinforcement learning doesn't need any data. In reinforcement learning, an agent who has a set of actions is trained in an environment using rewards or punishments considering the decisions of the agent in different situations.

In reinforcement learning, there is a *environment*, which have different states changes at each time step, $s_t \in S$, and a learning *agent* who can observe the environment and then learns from the outcomes of it's actions, $a_t \in A$. In reinforcement learning problems, the agents' decisions influence their own actions in proceeding steps, agents don't have any information of which actions to take in what situation and they try to maximize reward (or minimize punishment) by trying out different combinations in the set of actions.

One of the main challenges of RL is to build a model that can learn from the noisy and delayed environmental data. The latency between the incoming data forces training model to give a delayed input reaction and the delays add up to each other in a cascading manner. To overcome the delay problem significantly, rather than using computer vision to gather frames and analyze them one by one, a breakout game developed in such a way that all in-game data can be accessible to the model.

Another challenge is the incoming state data being highly correlated with the previously acquired ones, rather than being independent. To build a model which can perform well withing the high-dimensional correlated state space, a Deep Q-Network (DQN) is developed which combines reinforcement learning with deep neural networks. The many layered approach made it possible to build up more abstract representations of the correlated spatial data.

To be able to train our model, the action set, state space and rewards were defined for the Breakout game. The positive reward are defined as unit positive reward for every break of a brick. No negative rewards are defined. However, in order to stop the repetitive behavior, such as bouncing the ball of the wall and getting it back without breaking any bricks and exploiting the positive rewards, a random restart protocol is implemented, so that the model is pushed to explore rather than exploit.

II. Problem Description

The problem that we have is the utilization of deep reinforcement learning, with the specific model of convolutional neural networks, trained with Deep Q-Learning, whose input is raw pixel values, varying between $\{0, 1, \ldots, 255\}$, of a grayscale version of the image of the current state, and output is the most suitable action from the action space. The question that we would like to address is whether the trained agent can outperform a random agent, which takes random actions at each time step, and if so whether it can outperform a human agent that plays the game in the same settings.

III. Methods

As our project is to create a model with reinforcement learning, specifically DQN, we do not have a predefined dataset. With this in mind, our methodology can be separated into three parts.

A. Reinforcement Learning

Reinforcement Learning (RL) is a subset of Machine Learning (ML) where the aim is to teach a model, called agent, via its interactions with the surrounding environment. This method of learning requires no set of labeled or unlabeled data to be collected before the learning actually starts. Instead the agent, typically a neural network, is used for predicting the optimal action to be taken at each step based on its observation and a reward is determined by the environment which is used for training the agent. In RL, this environment is modeled as a Markov Decision Process (MDP) [2].

1) Markov Decision Process: MDP is a mathematical framework based on Markov Chains for decision making processes with inherent randomness. In Markov Decision Processes, we define:

- S: State Space (finite set),
- A: Action Space (finite set),
- A_s: Set of actions available at state s,
- $P^a_{ss'} = P_a(s, s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of action a in state s leading to state s',
- $R_a(s, s')$: Reward received after transitioning from s to s'.

2) Markov Property: For RL agents to work with MDPs we need the environment to be fully observable, meaning that state s must capture all the characteristics of the environment. In more technical terms, any state is S must satisfy the Markov Property which is defined as.

$$P[s_{t+1} \mid s_t] = P[s_{t+1} \mid s_1, \ldots, s_t] \tag{1}$$

This property essentially enables the environment to be memoryless which is required for Markov Chains and more importantly its extension MDPs.

3) Policy: The objective in an MDP is to optimize the *policy* of the decision making algorithm. Here, we define the function $\pi(s)$ that outputs the action chosen by the decision maker based on the current state s. This optimization mainly done by maximizing the cumulative reward function. This function can be expressed as:

$$G_t = \sum_{t=0}^{\infty} \gamma^t R_a(s_t, s_{t+1}), \tag{2}$$

where $0 \leq \gamma \leq 1$ is the discount factor. The equation above introduces the concept of *discount factors*. This parameter is quite important as it is one of the hyperparameters of RL training loops. It is useful for avoiding cyclic behavior and infinite returns, and representing an exponentially increasing uncertainty for the future time steps.

Moreover, the policy that maximizes the function given above is regarded as as the *optimal policy* and denoted as $\pi^*(s)$. It should be noted that this optimal policy is not necessarily unique.

4) State-Value Function: The state-value function, or just value function, is denoted by $V_\pi(s)$. It is the expected return stating from state s and following policy π of an MDP. In most applications, it is used to evaluate how good being in a state is. It is mathematically expressed as:

$$V_\pi(s) = E_\pi[G_t \mid s_t = s] \tag{3}$$

As can be seen in the equation above, calculating the cumulative reward function is required to find the value function of a state. This can be decomposed into a recursive function as the current reward plus the discounted value function of the successor by utilizing the Bellman Equation.

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma V_\pi(s_{t+1}) \mid s_t = s] \tag{4}$$

With this done, we now define the state-value function that is produced by the optimal policy π^* as the *optimal state-value function*. In mathematical terms:

$$V_*(s) = \max_\pi V_\pi(s) \tag{5}$$

5) Action Value Function: The action value function, also called the Q-function, is the expected return starting from state s, taking action a, and then following policy π. This is expressed as:

$$Q_\pi(s,a) = E_\pi[G_t \mid s_t = s, a_t = a] \tag{6}$$

Similar to the state-action function, we can also decompose this into a recursive function by redefining

$$G_t = R_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}), \tag{7}$$

and define the optimal action-value function as:

$$Q_*(s,a) = \max_\pi Q_\pi(s,a) \tag{8}$$

6) Finding the Optimal Policy: In all MDPs, three conditions are satisfied:

- An optimal policy π^* exists (not necessarily unique),
- Optimal policy achieves the optimal state-value function
$$V_{\pi^*}(s) = V_*(s) \tag{9}$$
- Optimal policy achieves the optimal action-value function
$$Q_{\pi^*}(s,a) = Q_*(s,a) \tag{10}$$

These three assumptions can be used to shows that finding the optimal policy π^* to solve the MDP can be done by maximizing over $Q_*(s,a)$ with:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } \arg\max_a Q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

This equation essentially means that finding the optimal policy can be done by following the path given by $Q_*(s,a)$ assuming that the optimal Q function is known [3].

B. Neural Networks

Artificial Neural Networks (ANNs), or simply Neural Networks (NNs), are machine learning models that are loosely based on real life neurons and their connections. Although the theoretical work for these types of models were mostly developed in mid 20$^{\text{th}}$ century, the applications of them were quite limited considering the computational power required for the necessary calculations.

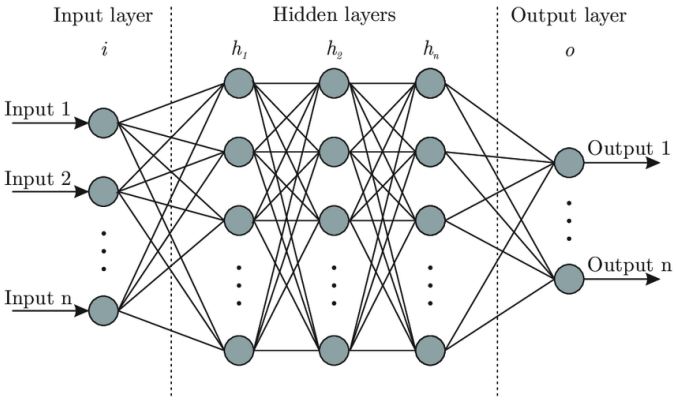


Fig. 2: A Basic ANN Structure [4]

In our project, the observation provided to the RL agent compromises of screenshots of the game's graphics. Therefore, the NN we used is made up of mainly convolutional layers for image processing with smaller fully connected layers at the end for decision making. The entire neural network used is:

1) Convolutional Layer: 32 filters,
2) ReLu Activation Layer,
3) Convolutional Layer: 64 filters,
4) ReLu Activation Layer,
5) Convolutional Layer: 64 filters,
6) ReLu Activation Layer,
7) Fully Connected Layer: 512 wide,

8) ReLu Activation Layer,
9) Fully Connected Layer: 3 wide

C. Deep Q Learning (DQN)

The standard Q learning algorithm uses the training episodes to fill up a Q table where each value corresponds to a specific action taken in a specific state. This method functions properly when the amount of state action pairs are relatively low. For example, the "Cliff Walking Problem", shown below, is simple enough that a Q table would be the feasible. In fact, the table would only have between 160 to 192 entries depending on how the environment is defined.

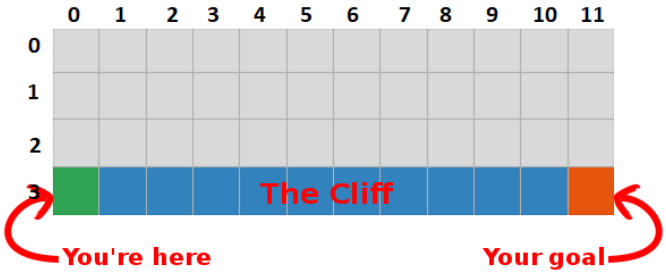


Fig. 3: Cliff Walking Problem [5]

For more complicated environments, this approach quickly loses its feasibility. In those cases, like what we have in Breakout, a neural network is used as a nonlinear function approximator to replace the table. In DQN, the nn provides a mapping from the give state s, to the Q function of all possible actions in that state $Q(s, \underline{a})$.
The training loop of our network has a few additional, but still standard, features. These are:

1) ϵ-greedy Policy: To simulate *exploration*, the action will be chosen randomly with probability ϵ,
2) Experience Replay Memory: To combat the unstable nature of deep reinforcement learning, a circular memory with a set size will be used to reduce the correlations between the sequence of observations used in every training batch.
3) Target Network: An additional *target network* is used to increase the stability of the learning.

The final learning algorithm is given in algorithm 1 in section B.

IV. Results

The agent was trained for 100000 episodes with random restarts every 20000 episodes. For evaluating the performance the agent average game length, mean Q value, mean reward, and mean loss is recorded. Results from one random restarts is shown in Figure 4 in section C. From the figures we can conclude that the agent improves its knowledge about the game nearly every episode. The graphs are not smoothed out, even tough the trajectory is clearly upwards for received reward, and as the agent explores the reward fluctuates.

V. Discussion

As previously mentioned we trained out agent for 100000 episodes with random restarts every 20000 episode. In random restarts epsilon value of the agent is set to its default starting value. Without random restarts after the agent is trained for certain number of episodes, the agent starts to overfit to samples, and performance of the agent in terms of received reward decreases. In this regard, performing random restarts provides the agent with novel samples, as it takes random actions.

While training we observed that as agent's replay memory size increases the agent is able to avoid overfitting for more number of episodes. This phenomena can be explained by the fact that we assume every memory sample is conditionally independent from each other. In theory, it is obvious that recorded transitions are dependent on each other as they are recorded consecutively. In practical terms, if we increase the memory size, and take random samples after shuffling, probability of samples being dependent can be negligible.

For increasing the agent performance, we decreased the action space by taking out the `FIRE` action. This action is only used when the agent loses a life, or at the beginning of the game. After reducing the action space, and training new agents for the new model the agent's average reward nearly doubled. This can be by the fact that `FIRE` action is ignored if the ball is not on the paddle. Trying to learn when to fire is a difficult task as this information is encoded in the number of lives displayed at the top the screen, or the ball being at the paddle.

VI. Conclusion

All in all, we successfully implemented a deep reinforcement learning architecture using convolutional neural networks, trained with Deep Q-Learning, whose input is raw pixel values, varying between $\{0, 1, \ldots, 255\}$, of a grayscale version of the image of the current state, and output is the most suitable action from the action space. As we demonstrated in our final presentation demo, our trained agent outperforms the random agent with less than an hour of training. As demonstrated, our own group members also played the game using the keyboard agent, although none of the group members can perform more than 20 in any of the trials. In this context, the trained agent outperforms every group member with the score of 28. Therefore, the answer to our initial question is yes! The trained agent with the specified model can outperform both random and human agents.

Throughout the project, we learned the basic approaches of reinforcement learning. We learned that by using CNNs and DQNs, it is possible to learn control policies directly from high-dimensional sensory input using reinforcement learning. The resultant agent is satisfactory to respond our question with "yes". In addition to basic theoretical approach, we also had the chance to interact with a practical issue that we were not expecting to face. While we were trying to adjust the exploration versus exploitation hyperparameter, ε, we faced with an issue that we later learned it is called the credit assignment problem. The credit assignment problem means that it can happen that we choose an action, and we only win or lose hundreds of actions later, leaving us with no idea of as to which of our actions led to this win or lose, thus, making it difficult to learn from our actions [3]. We learned that solely increasing the training duration does increase the performance of the agent. We learned that in order to overcome this credit assignment pitfall, using random restarts actually increases the performance of the agent, by adjusting the exploration versus exploitation problem in the direction of exploration.

As the future work of the project, performance of the agent can be increased by changing the deep neural network architecture and increasing the training duration. During our project, we have encountered with more complex approaches like double DQNs, Never Give-Up (NGU), etc. Experimentation on these approaches can be expressed as potential future work. Furthermore, these approaches can be generalized as *DeepMind*'s `Agent57` to provide trained agents for 57 atari games [6].

REFERENCES

[1] OpenAI. (Oct. 2019). "Gym," [Online]. Available: https://gym.openai.com/envs/Breakout-v0/. [Accessed: Mar. 3, 2021].

[2] L. Buşoniu, T. de Bruin, D. Tolić, J. Kober, and I. Palunko, "Reinforcement learning for control: Performance, stability, and deep approximators," *Annual Reviews in Control*, vol. 46, pp. 8–28, 2018, ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2018.09.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1367578818301184.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[4] Buseyarentekin, M. C. Örücü, Merveeekilic, and Iremnur, *Tag: Evrişimsel sinir ağları (cnn)*. [Online]. Available: https://globalaihub.com/tag/evrisimsel-sinir-aglari-cnn/.

[5] L. Vazquez, *Understanding q-learning, the cliff walking problem*, Apr. 2018. [Online]. Available: https://medium.com/@lgvaz/understanding-q-learning-the-cliff-walking-problem-80198921abbc.

[6] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," *CoRR*, vol. abs/2003.13350, 2020. arXiv: 2003.13350. [Online]. Available: https://arxiv.org/abs/2003.13350.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. DOI: 10.1038/nature14236. [Online]. Available: https://doi.org/10.1038/nature14236.

APPENDIX A
CONTRIBUTION

TABLE I: Task Sharing

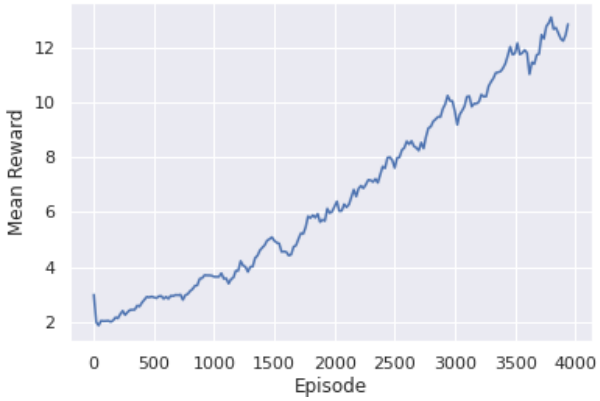
Student	Task
Abdullah Arda Aşçı	Setting OpenAI, and developing wrappers for `gym` environment. Training & testing with different hyperparameters.
Atahan Yorgancı	Background research about RL and DQN. Implementation of artificial agent. Cloud based training training & testing with different hyperparameters.
Alim Toprak Fırat	Background research about RL and Q-Learning. Training & testing with different hyperparameters.
Tuna Alikaşifoğlu	Developing CLI for running, and training using `gym` environment. Implementation of DQN. Training & testing with different

APPENDIX B
DQN ALGORITHM

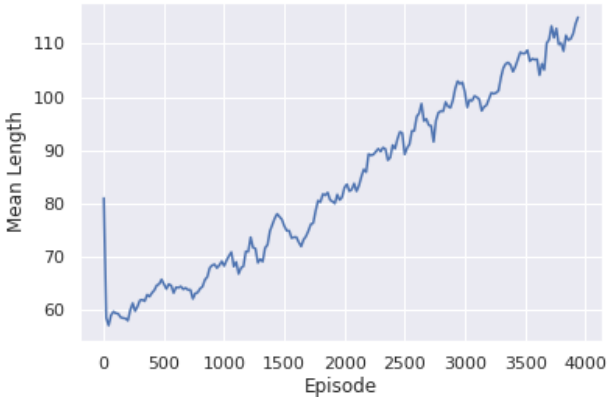
Algorithm 1: Deep Q-Learning with Experience Replay [7]

Initialize replay memory with capacity N;
Initialize action-value function Q with random weights θ;
Initialize target action-value function Q^{-} with weights $\theta^{-} = \theta$;
for *episode* = 1 *to* M **do**
  Initialize sequence $s_1 = \{x_1\}$;
  for $t = 1$ *to* T **do**
    With probability ε select a random action a_t;
    otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$;
    Execute action a_t and observe reward r_t with next state s_{t+1};
    Store the set (s_t, a_t, r_t, s_{t+1}) in memory;
    Sample minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from memory;
    if *episode terminates at step* $i + 1$ **then**
      $y_i = r_i$;
    else
      $y_i = r_i + \gamma \hat{Q}(s_{i+1}, a')$;
    end
    Perform gradient descent on $(y_i - Q(s_i, a_i))^2$ on network parameters θ;
    Every C step, reset $\hat{Q} = Q$;
  end
end

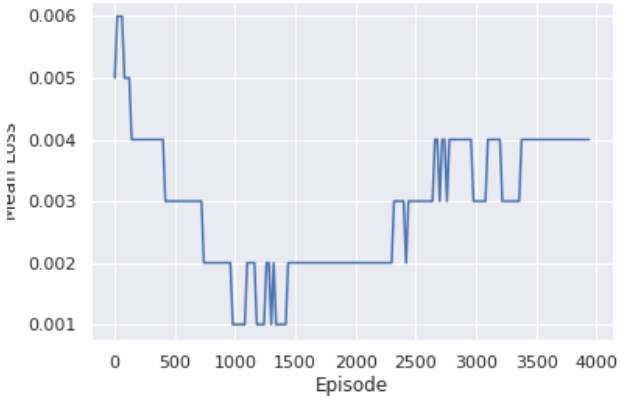
APPENDIX C
RESULT PLOTS



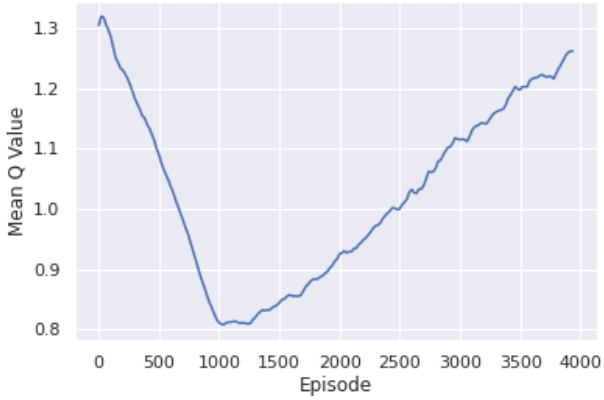
(a) Mean Reward



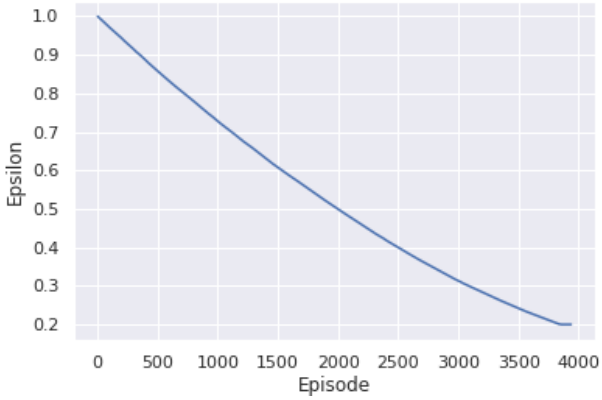
(b) Mean Length



(c) Mean Loss



(d) Mean Q Value



(e) Epsilon

Fig. 4: Change per Episode