# Playing ATARI Breakout using Deep Q-Learning

Apoorv Malik
Oregon State University
malikap@oregonstate.edu

Ameyassh Nagarajan
Oregon State University
ameyass@oregonstate.edu

Shreya Palit
Oregon State University
palits@oregonstate.edu

## Abstract

*Building an agent that learns from high-dimensional sensory inputs is one of the challenges of Reinforcement Learning. Most agents operate on data that has been hand-labelled, implying that the performance of these agents relies on the quality of the labelled data. This project draws inspiration from the researchers at DeepMind Technologies to build a Deep Learning (DL) Agent that specializes in playing the game of ATARI Breakout, where the data for the agent is the environment of the game itself.*

## 1. Introduction

Creating a Deep Learning agent that works off raw, high-dimensional data is a difficult task. In this project, we draw inspiration from Google DeepMind's pivotal paper [1] to create a Reinforcement Learning (RL) agent that plays the ATARI Breakout. The inspiration for the RL agent comes from Markov Decision Processes (MDP's) where the agent learns the impact of a decision it takes based on the reward associated with it. In this paper, we give a brief overview of RL, MDP, and Deep Q-Networks.

### 1.1. Problem Statement

The main objective of this project is to design an AI agent that can score as high as possible in Atari Breakout. In Breakout, a layer of bricks lines the top third of the screen, and the goal is to destroy them all. A ball moves straight around the screen, bouncing off the top and two sides of the screen. When a brick is hit, the ball bounces back, and the brick is destroyed. The agent receives a reward of +1 for every brick destroyed. The agent loses a turn when the ball touches the bottom of the screen. To prevent this from happening, the agent has a horizontally movable paddle to bounce the ball upward, keeping it in play. There is a total of five turns for the agent. The episode terminates when all five turns are lost. The action space consists of four total actions: 'NOOP', 'FIRE', 'RIGHT', 'LEFT'. Given the state of the game in the form of raw pixel data, the agent needs to find the most appropriate action that will maximize the reward it will receive from the environment. The environment is completely solved when all the bricks are destroyed.

### 1.2. Approach

We have developed a solution for this problem using a deep neural network model. The model is trained to help with the Deep Q-learning algorithm. The problem consists of a high dimensional and very large state space in the form of raw pixel data. So, we have used a Deep Neural Network model, which can approximate the policy function. The policy function maps the states to the most appropriate action. Some important techniques that we have used in this solution are defined below:

- Data preprocessing: We have preprocessed the Atari Breakout frames before supplying them as input to the model.
- Deep Neural Network: This is the backbone of the solution model; it is used to predict what actions to execute given a particular state of the game.
- Deep Q-Learning: This is the algorithm that we have used to train the Deep Neural Network.
- Huber Loss: This is a special loss function used for optimization of the neural network.
- Target network: This is an improvement to the original DQN algorithm, which ensures stationary targets for the update step. This ensures a stable learning curve for the neural network.
- Experience replay: This is another improvement to the original DQN algorithm that allows the network to avoid learning from highly correlated states, which can be very inefficient.

### 1.3. Metrics

The standard evaluation metric for ATARI game play is the score returned by the game itself. The score is incremented for every successful hit on the brick by the ball. There are six rows of bricks. The color of a brick determines the points scored when the ball hits it. These are as follows: Red - 7 points  Orange - 7 points  Yellow - 4 points  Green - 4 points  Aqua - 1 point  Blue - 1 point.

The player has five total lives, and the game ends if all five lives are depleted (The player loses a life if the ball hits the bottom of the screen). The total score is calculated at the end of the game by adding all the points scored during the game. No score penalty is applicable in this environment. The reward/score system for the DQN agent will follow different score criteria. Moreover, the real performance and evaluation of the agent will be measured based on original score criteria mentioned above. The reward/score system for the agent is as follows: The rewards received by the agent (after hitting each block) is scaled between +1 and -1 (there is no negative reward in this environment). Since the scale of scores varies significantly from game to game, the positive rewards are fixed to be +1 and all negative rewards to be −1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitudes. The episode ends after a single turn is lost. However, the game resets only on true game over. This is done by DeepMind in their original solution since it helps value estimation. The total reward obtained by the agent at the end of the episode will be the total number of bricks hit (irrespective of their points), while the total points scored by the agent follow the original Breakout game's score criteria.

### 1.4. Data Exploration

There are two primary components of reinforcement learning, the environment, and the agent. The agent interacts with the environment by choosing some action (from the set of all possible actions), and the environment responds to the agent by changing its state (observation) and giving out rewards (score) to the agent.

The dataset for this project comes from the OpenAI Gym's Atari game environment. For this project, we will use the Atari Breakout environment. We will use the modified version of the original Breakout environment, designed by Google DeepMind. This new version is called "BreakoutDeterministic-v4". In this version, the agent sees and selects actions on every 4th frame instead of every frame, and its last action repeats on the skipped frames. The action space of this environment consists of four possible actions, 'NOOP', 'RIGHT', 'LEFT', and 'FIRE'. The definitions of these actions are as follows:

**'NOOP'**: No Operation; no real action is executed when this is selected.
**'RIGHT'**: Move the horizontal paddle to the right.
**'LEFT'**: Move the horizontal paddle to the left.
**'FIRE'**: Bounces the ball upward, only executable once and at the start of a new episode.

### 1.5. Exploratory Visualization

The dataset for this project will be the raw pixel data representing the game's screen. This will be the only input data that we will give to the model. The input data will vary as the agent progresses in the simulation. The model is provided a new image as input at every time step.

Figure 1: Starting image frame from Breakout.

The input that we will give to our Deep Learning Model will be the raw pixel data of the game's screen. The Atari frames are 210×160 pixel images with a 128 color palette. Providing these raw pixel data can be computationally challenging, and the algorithm may converge very slowly due to high dimensional and complex input. So, we need to apply an essential preprocessing step. The raw frames are preprocessed by first converting their RGB representation to a gray-scale representation, and then they are down-sampled to an 84×84 image. The last four frames (before the current frame) are preprocessed and then stacked together to produce a tensor of shape (1, 84, 84, 4) which will be the input to the Deep Neural Network. This stacking of previous frames allows the model to infer the velocity of the ball and be more effective at predicting the Q-Values. The frame stacking is shown in **Figure 2**.

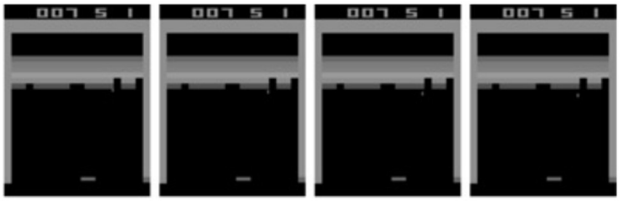

Figure 2: Four consecutive preprocessed frames from are stacked together. The resulting tensor has shape (1, 84, 84, 4).

## 2. Methodology

In this section we talk about the algorithms, techniques, network architecture and methods used to build our RL agent and train the model.

Here we have used 6 neural network layers:

1. Layer 1: Convolutional layer -> Filters = 32, Kernel size = 8, Strides = 4, Activation = Rectified Linear(relu)
2. Layer 2: Convolutional layer -> Filters = 32, Kernel size = 4, Strides = 2, Activation = Rectified Linear(relu)
3. Layer 3: Convolutional layer -> Filters = 32, Kernel size = 3, Strides = 1, Activation = Rectified Linear(relu)
4. Layer 4: Dense -> Units = 256
5. Layer 5: Dense -> Units = 128
6. Layer 6: Dense -> Units = 4

We have used L2 regularization in the Dense layers. The final (last) layer has four outputs corresponding to the Q-values of each action, which relates to the probability of taking those actions in a given state.

### 2.1 Algorithms and Techniques

Recall that in Q-Learning we use Q-table to store the Q-values. But if the number of states is very large, this approach will be inefficient. Therefore, the original Q-Learning algorithm fails for high state spaces. In Deep Q-Learning (DQN), instead of storing the Q-values in a table, we will use neural networks to approximate the policy function, which maps each state to its Q-values. This is depicted in **Figure 3** below.

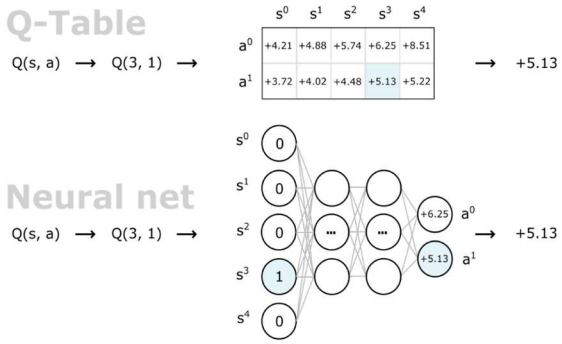


Figure 3: Q-Table vs Neural Network

To train the Deep Neural Network model, we will use the DQN algorithm. The network consists of convolutional layers that can extract information from the image data, followed by fully connected layers. With the use of both convolutional and fully connected layers, the Deep Neural Network can approximate the Q-values for a given state. The output layer is a fully connected linear layer with a single output (representing the Q-value) for each of the valid actions. This network is visualized in **Figure 4**.

The Deep Q-learning algorithm is implemented in this project with several improvements. These are Experience Replay, Target Networks, and Huber Loss. These improvements allow the algorithm to converge faster and efficiently. These improvements are explained in the later sections.

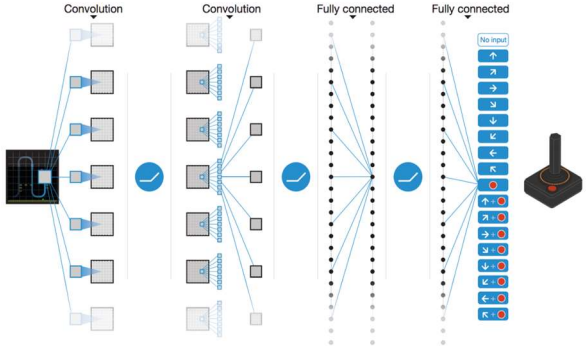


Figure 4: A visual representation of the network.

The pseudocode for the DQN algorithm is shown below.

**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
**For** episode = 1, $M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t = 1$,T **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**

### 2.2 Experience Replay

The neural network's weights will not be updated after every episode. We will use Experience Replay to store the various experiences encountered in the game. Experience replay will store the observation tuples (state,

action, reward, next state, done) that are seen during the gameplay, in a memory. After a fixed number of episodes, the model will sample a batch (of fixed size) of random experiences from the memory and use them to perform gradient descent and update steps on the network.

The reason why experience replay is helpful has to do with the fact that in reinforcement learning, successive states are highly similar. This means that there is a significant risk that the network will completely forget about what it's like to be in a state that it hasn't seen in a while. Replaying experience prevents this by still showing old frames to the network. It will also fix the issue of correlation between consecutive experiences, that can make the model incline towards choosing one kind of action. Hence, experience replay will ensure that the model has a stable learning rate and will prevent it from diverging.

### 2.3 Target Network

The Target network is another neural network with the same architecture as the main one. We update this network after every 'n' episode. Recall, that the Q-values are estimated by the following formula:

$$Q(s,a) = r + maxa(Q(s',a'))$$

Equation 1

We use the target network to predict the targets for improving our model, more specifically, we use it to predict the future Q-values, i.e. Q(s', a'). And use the main network for predicting the current Q-values. Therefore, the loss function is computed by taking the difference of the predictions of the two networks. This is shown below.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

Equation 2

Here, $\theta_i^-$ is the target network and $\theta_i$ is the main network, we synchronize both the networks after every n episode. The use of target network makes the algorithm more stable compared to the standard Q-learning, where an update that increases $Q(s_t, a_t)$ often also increase $Q(s_t + 1, a)$ for all $a$ and hence also increases the target $y_j$, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters adds a delay between two consecutive $Q$ function updates. This assures that the target $y_j$ is stable, making divergence or oscillations much more unlikely.

Eventually, we will converge the two networks so that they are the same, but we want the network that we use to predict future Q-values to be more stable than the network that we actively update every single step. We update parameters of the target network to be same as the main network every *C* iteration. **Figure 5** shows the functionality of the target network.

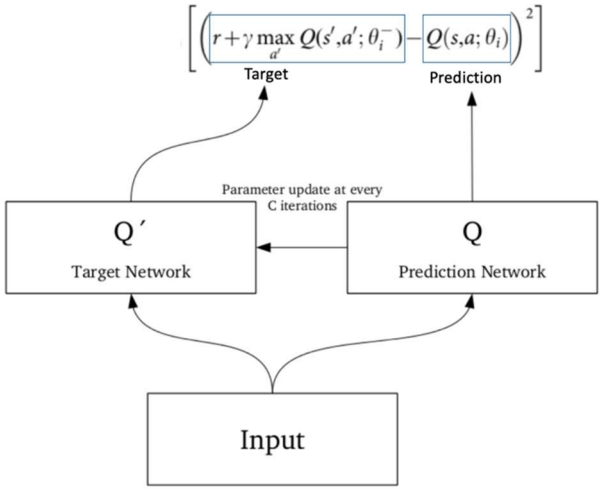


Figure 5: A visual representation of the network.

### 2.4 Huber Loss

We will use Huber Loss as a loss function for our Neural Network.

$$\mathrm{L}_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

Equation 3

This function is quadratic for small values of 'a', and linear for large values, with equal values and slopes of the different sections at the two points where $|a| = \delta$. The variable $\boldsymbol{a}$ often refers to the residuals, that is, the difference between the observed and predicted values, so the former expression can be expanded to:

$$\mathrm{L}_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \le \delta, \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Equation 4

Here, $y - f(x)$ is the difference between the true value and the predicted value. The introduction of Huber loss allows less dramatic changes, which often hurt RL.

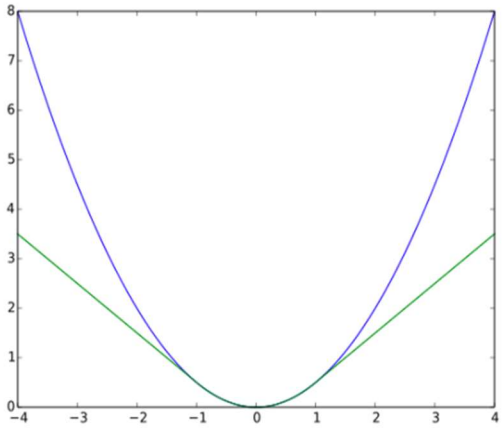


Figure 6: Green curve is the Huber Loss and Blue curve is the squared loss error.

# 3 Experiments

In this section we discuss the experiments we have used to evaluate our models and the results we achieved using these experiments.

## 3.1 Benchmarks

For the benchmark, we will use the following scores achieved by various algorithms and human-level performance. The DQN model is expected to beat all these scores by a significant margin. The highest score in these benchmarks is that of a human. This score would be our threshold for measuring the agent's performance. The human performance for this game is reported in section 5.3 of the paper by DeepMind Technologies [1]. **Table 1** shows the result of various systems that played the game.

| Planner | Score |
|---|---|
| Random Policy | 1.2 |
| Human | 31 |
| Policy Gradients | 10.5 |
| MCTS Depth 10 (50 Samples) | 2.7 |
| MCTS Depth 10 (150 Samples) | 3.0 |
| Wt. MCTS Depth 10 (150 Samples) | 6.1 |

Table 1: Benchmarks

## 3.2 Hyperparameters

The following hyperparameters are used to train the model.

| Hyperparameter | Value | Description |
|---|---|---|
| Minibatch Size | 32 | Number of training cases over which each stochastic gradient descent (SGD) update is computed. |
| Replay Memory Size | 1000000 | SGD updates are sampled from this number of most recent frames. |
| Agent History Length | 4 | The number of most recent frames experienced by the agent that are given as input to the Q network |
| Discount Factor | 0.95 | Discount factor gamma used in Q-learning update |
| Action Repeat | 4 | Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame. |
| Update Frequency | 4 | The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates. |
| Learning Rate | 0.00025 | The learning rate used by RMS Prop. |
| Decay factor (rho) | 0.95 | "rho" is the decay factor, or the exponentially weighted average over the square of the gradients. |

| Hyperparameter | Value | Description |
|---|---|---|
| Initial Epsilon | 1 | Initial value of epsilon in epsilon-greedy exploration. |
| Final Epsilon | 0.01 | Final value of epsilon in epsilon-greedy exploration. |
| Epsilon Decay | 0.9995 | Factor by which epsilon is decayed after each episode. |
| Replay Start Size | 32 | A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory. |
| No-op Max | 30 | Maximum number of "do nothing" actions to be performed by the agent at start of a new episode. |
| Target Update Frequency | 15 | Number of episodes after which the target network's weights are updated and synchronized with the weights of the main network. |

Table 2: Hyperparameter Values

### 3.3 Training and Results

We trained our model in 3 separate training sessions. The rolling mean of the scores achieved during the multiple training sessions are plotted below. It is important to note here that the scores achieved by the agent are scaled. The score is calculated at the end of an episode and the episode is terminated when a single life/ chance is lost. For example, a score of 5 means that the agent has successfully hit 5 bricks (regardless of their points) in a single life/chance.

• In the first training session, the model was trained for about 32k episodes. The model obtained a max average score of 3.6 (over 100 episodes) and max score of 14. The plot *scores vs episodes* is shown in **Figure 7**. It can be seen that scores are increasing at a nice rate, as the number of episodes increases.

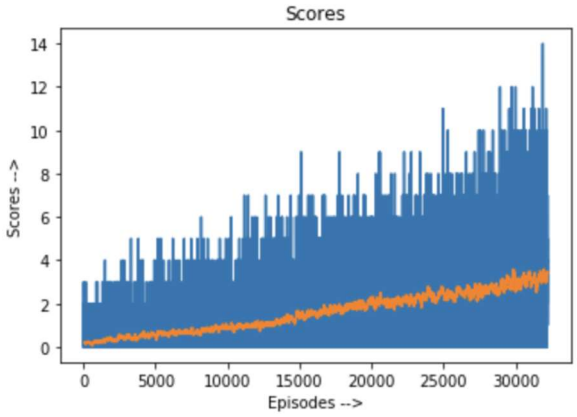


Figure 7: Scores (rolling mean) vs Episodes (1st training session).

• In the second training session, the model was trained for about 25k episodes. The agent attained a max average score of 6.89 (over 100 episodes) and max score of 30. This is almost double the score that was achieved in the first training session. The model has made significant progress in this training session. The plot *scores vs episodes* are shown in **Figure 8**.

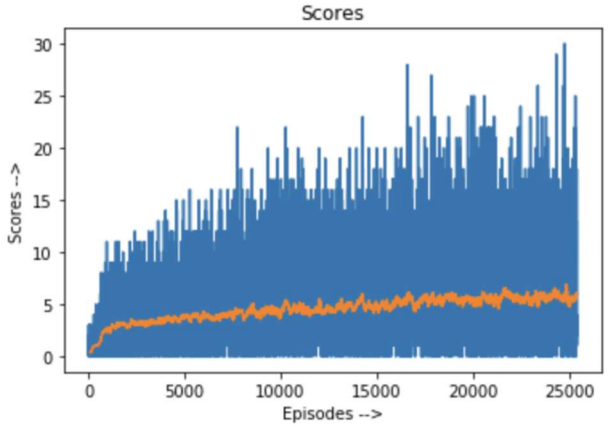


Figure 8: Scores (rolling mean) vs Episodes (2nd training session).

• In the third and the last training session, the model was trained for about 10k episodes. The model attained a max average score of 7.56 (over 100 episodes) and max score of 35. Here, only a slight improvement can be seen in contrast

to the previous training sessions. The plot *scores vs episodes* is shown in **Figure 9**.

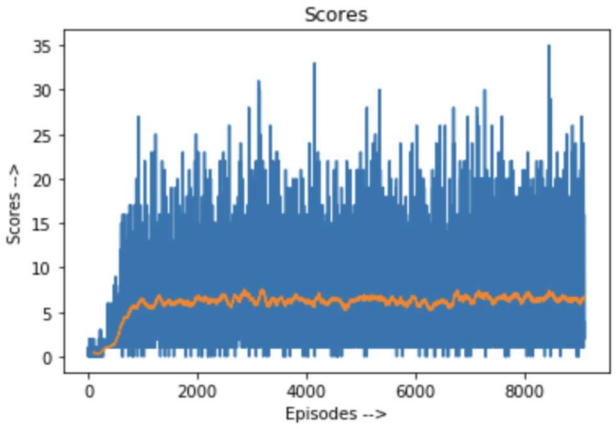


Figure 9: Scores (rolling mean) vs Episodes (3rd training session).

It seems that the algorithm has converged at this point. It is clear from the plot that the scores are not increasing further. Moreover, an increase in the agent's performance may be possible by resetting the epsilon (exploration rate) to a higher value and then training the model for more episodes. This may allow the agent to discover a special strategy (more about it in Section 4) to play this game, which may make it score 200+ points in the game!

After 100+ hours of training, the agent is able to achieve a score of 83 in the Atari Breakout video game. This is certainly a good score. The screenshot of the final state of the game (before the game terminates) is shown below.

Figure 10: Agent's performance

# 4 Conclusion

In summary, with the use of Deep Q-learning algorithm with experience replay and target network, the RL model is able to achieve a very good score in Atari Breakout. The basic preprocessing step is required for this problem, which mainly includes stacking and scaling of image frames. Also, to train the agent efficiently, *scaling of rewards* is necessary along with *frame skipping* and *episodic lives*. The model has clearly beaten all the thresholds specified and performs better than an average human at this game. The most important aspect of this project was the adoption of improvements to the DQN algorithm which includes Experience replay, Huber loss, and Target Network. This allowed the agent's score to increase significantly. The model was trained on our personal computer. The training process was done in multiple sessions over a course of 5 days, using the M1 Pro processor. The model was trained for about 70k episodes in total.

## 4.1 Improvements

Clearly, there is room for improvement for this project. Google DeepMind's solution for this problem, achieved a much higher score. A higher score of 418.5 can be achieved using Double DQN [2] using a Dueling DQN also yields a better score of 345.3 [2]. This is much higher than our model's score. These new architectures: Double DQN and Dueling DQN are clearly better than our current implementation. Also, it is important to note that DeepMind's [1] solution network was trained for 10 days, while our network was trained for about 5 days. A higher score may be achieved by improving the hyper-parameters and amending the current deep neural network architecture. Increasing the training time will help to achieve a higher score. The best strategy to score high in this game is to dig a tunnel at one side by aiming the ball towards it, as shown in **Figure 11.**

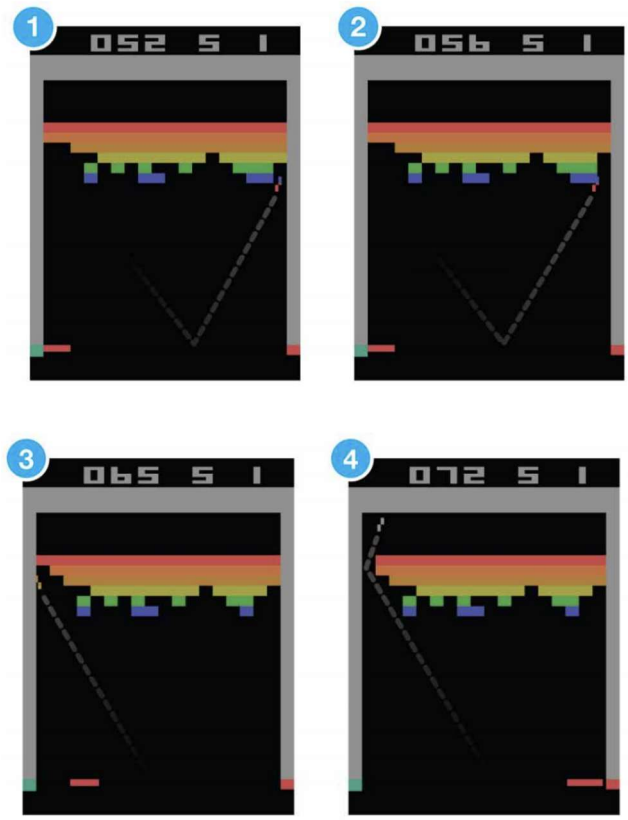


Figure 11: Efficient strategy to play Breakout.

In the DeepMind's [1] solution, at one point the agent starts to aim directly at the sides to dig a tunnel and hit the blocks from above; this strategy causes the score to increase rapidly.

In our solution, the agent did not discover this strategy yet. But with some tweaks, improvements and more training, the agent could be able to exploit this strategy and perform even better!

# 5 References

[1] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning." DeepMind Technologies. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[2] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," arxiv.org https://arxiv.org/abs/1511.06581