# PLAYING ATARI WITH DEEP REINFORCEMENT LEARNING

Ayan Abhiranya Singh

# Introduction

This paper was authored by the following team at DeepMind Technologies: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller.

The goal of this paper was to learn game-control policies from high-dimensional sensory input using reinforcement learning.

The proposed model is a convolutional neural network (CNN). The input is the raw pixels of a frame of the game and the output is a value function estimating future rewards.

## Example Inputs for the model

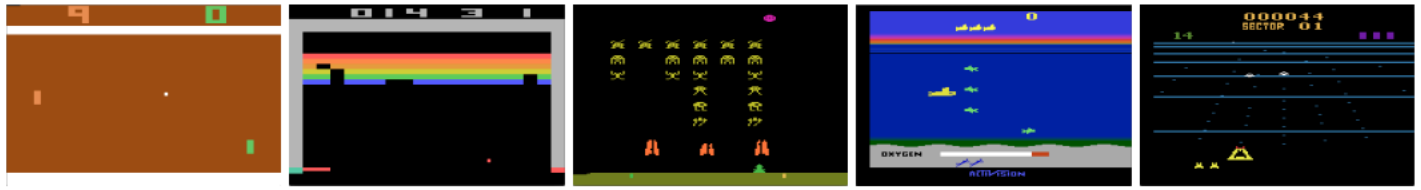


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

- The games above are implemented in The Arcade Learning Environment (ALE).
- The goal is to build a neural network agent that receives the high dimensional visual input (210 × 160 RGB video at 60Hz) shown above and performs a set of actions that were designed to be difficult for humans players.
- The expectation is that a single NN architecture is able to learn and play all the games above.

# TD-Gammon

TD-Gammon is a program that used a multi-layer perceptron with one hidden layer to estimate a value function using a reinforcement learning algorithm like Q-learning.

Best known success story of reinforcement learning being used to play video games.

## Deep Q-learning

- The approach used by the authors of the paper is deep q-learning, which connects the reinforcement learning algorithm to a deep neural network which operates directly on RGB images.

## Preprocessing

- Processing raw Atari frames, which are 210 × 160 pixel images with a 128 color palette, can be computationally demanding.
- To reduce dimensionality, frames are converted from RGB to gray-scale and down-sampling it to a 110×84 image.
- The final input representation is obtained by cropping an 84 × 84 region of the image that roughly captures the playing area. The final cropping stage is only required because we use the GPU implementation of 2D convolutions, which expects square inputs.

## Proposed Neural Network Architecture

- The input to the neural network consists is an 84 × 84 × 4 image.
- The first hidden layer convolves 16 8 × 8 filters with stride 4 with the input image and applies a rectifier nonlinearity.
- The second hidden layer convolves 32 4 × 4 filters with stride 2, again followed by a rectifier nonlinearity.
- The final hidden layer is fully-connected and consists of 256 rectifier units.
- The output layer is a fully connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games considered.

# The Stochastic Environment

- The agent interacts with the environment, E, which is stochastic (cannot be determined completely by the agent).
- At each time-step the agent selects an action from the set of legal game actions, A = {1, . . . , K}. The action is passed to the emulator and modifies its internal state and the game score.
- The agent knows nothing about E, and only observes an image $x_t \in R^d$ from the emulator, which is a vector of raw pixel values representing the current screen.
- Based on the change in game score, the agent receives a reward $r_t$ .
- Note that in general the game score may depend on the whole prior sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed.

# Sequences as Markov Decision Processes

- Since the task is partially observed, any single frame cannot provide enough information about the current game state and thus sequences of frames are considered.
- All sequences in the emulator are assumed to terminate in a finite number of time-steps. This gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state.
- The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes future rewards.
- Assuming that future rewards are discounted by a factor $\gamma$ at each time step, the discounted future return at time t, $\mathrm{R}_t = \sum_{t'}^{T} \gamma^{t'-t} r_{t'}$

## Optimal Q-function – the Bellman Equation

- The optimal action-value function Q*(s, a) as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a:

$$\boldsymbol{Q^*(s,a) = \max_{\pi} E[R_t | s_t = s, a_t = a, \pi]},$$ where $\pi$ is a policy mapping sequences to actions.

- The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following 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′ maximizing the expected value of $\mathrm{r} + \boldsymbol{\gamma\, Q^*(s,a)}$ :

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

## Loss function and stochastic gradient descent

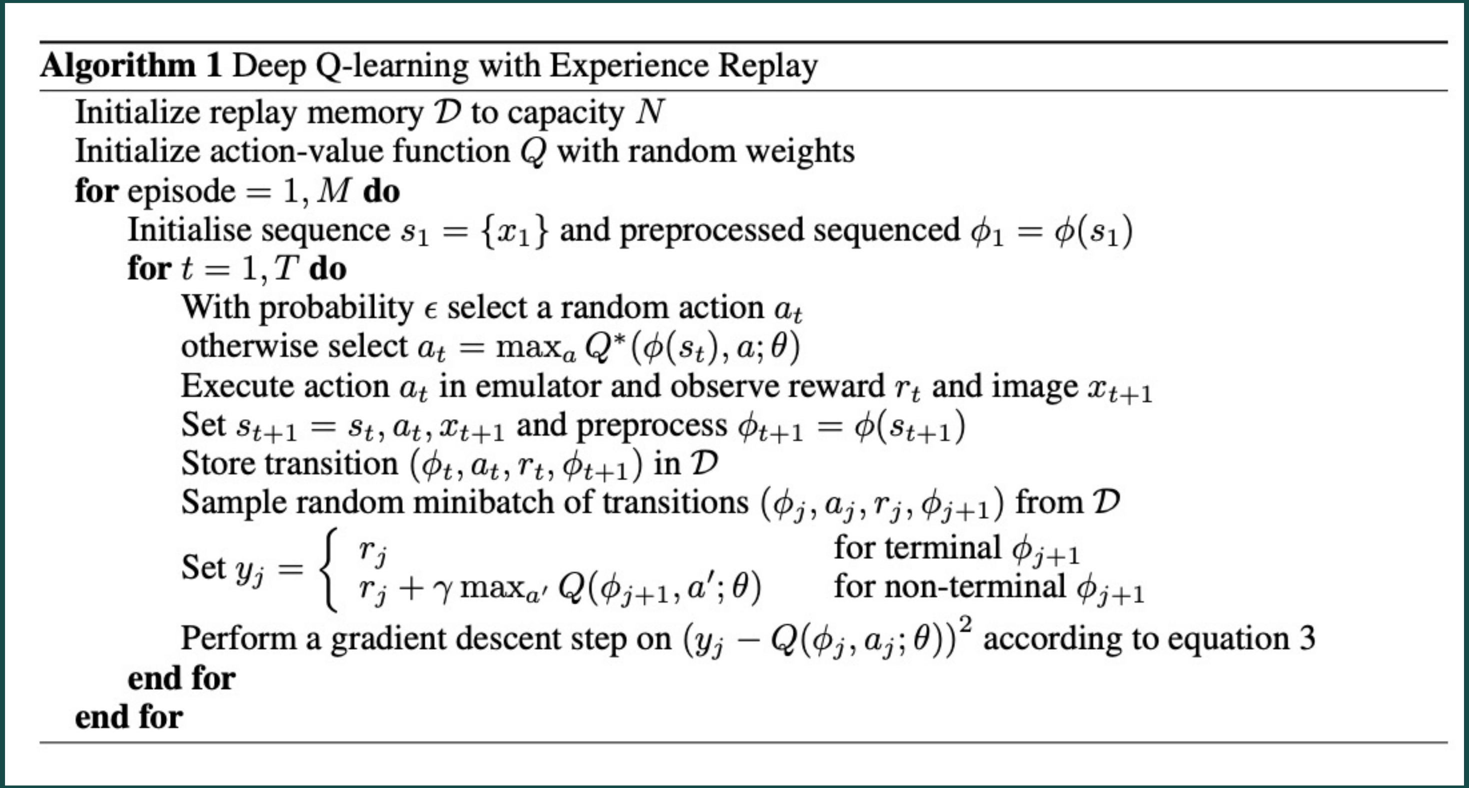
- A Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i:

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

- Differentiating the loss function with respect to the weights we arrive at the following gradient:

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

# Deep Q-learning with Experience Replay (Algorithm)



**Algorithm 1** Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        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 $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

# Experiments: Average Reward per Episode

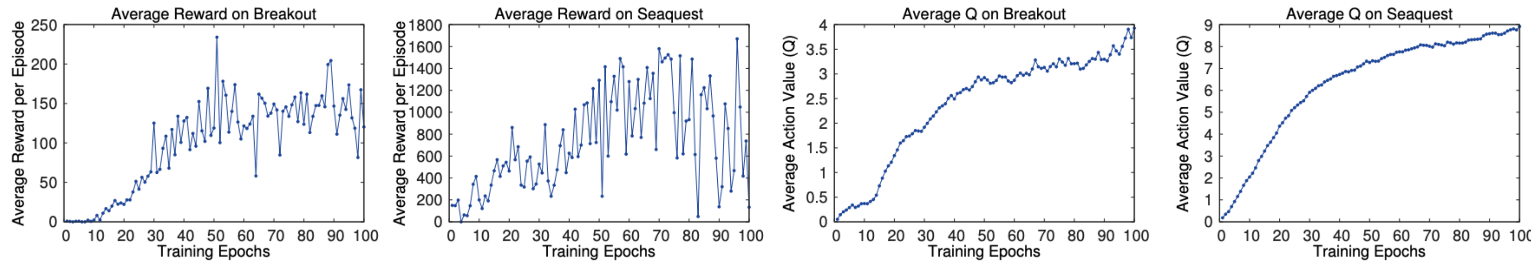


Figure 2: The two plots on the left show average reward per episode on Breakout and Seaquest respectively during training. The statistics were computed by running an $\epsilon$-greedy policy with $\epsilon = 0.05$ for 10000 steps. The two plots on the right show the average maximum predicted action-value of a held out set of states on Breakout and Seaquest respectively. One epoch corresponds to 50000 minibatch weight updates or roughly 30 minutes of training time.

## Experiments: predicted Q-values on Seaquest

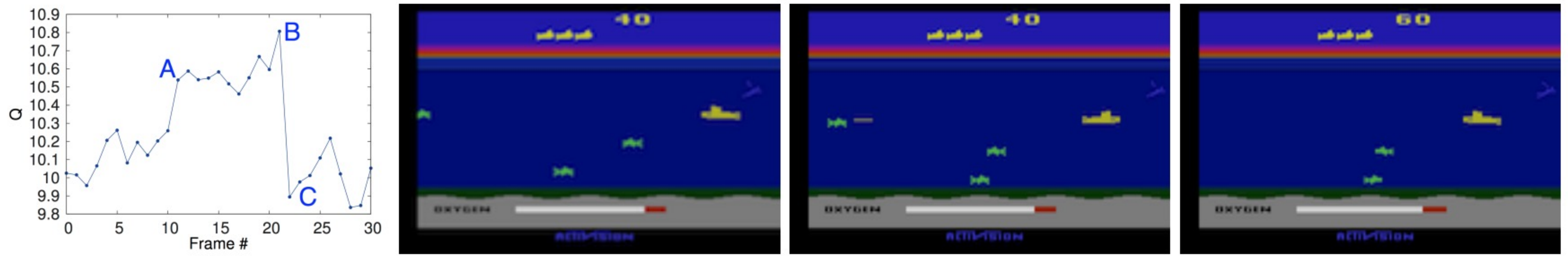
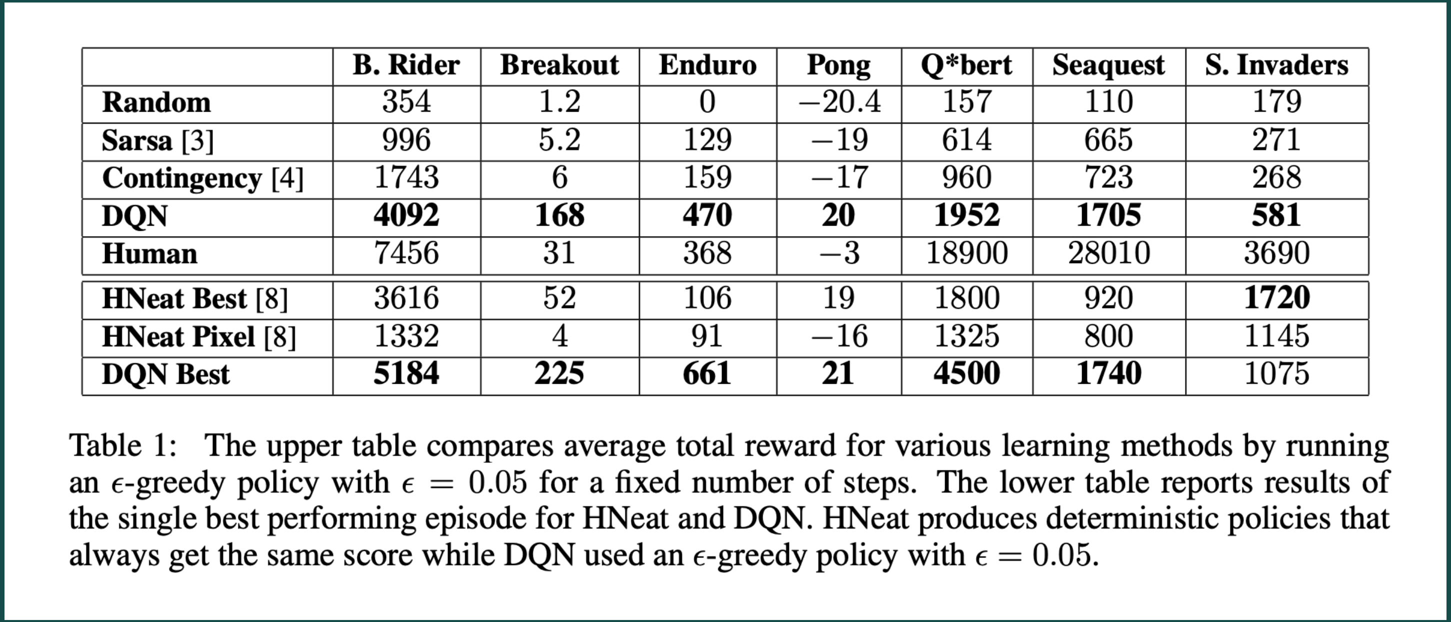


Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

## Experiment: Reward comparison

- The HNeat Best score reflects the results obtained by using a hand-engineered object detector algorithm that outputs the locations and types of objects on the Atari screen.
- Convolutional networks trained with this approach are referred to as Deep Q-Networks (DQN).



| | **B. Rider** | **Breakout** | **Enduro** | **Pong** | **Q*bert** | **Seaquest** | **S. Invaders** |
|---|---|---|---|---|---|---|---|
| **Random** | 354 | 1.2 | 0 | −20.4 | 157 | 110 | 179 |
| **Sarsa** [3] | 996 | 5.2 | 129 | −19 | 614 | 665 | 271 |
| **Contingency** [4] | 1743 | 6 | 159 | −17 | 960 | 723 | 268 |
| **DQN** | **4092** | **168** | **470** | **20** | **1952** | **1705** | **581** |
| **Human** | 7456 | 31 | 368 | −3 | 18900 | 28010 | 3690 |
| **HNeat Best** [8] | 3616 | 52 | 106 | 19 | 1800 | 920 | **1720** |
| **HNeat Pixel** [8] | 1332 | 4 | 91 | −16 | 1325 | 800 | 1145 |
| **DQN Best** | **5184** | **225** | **661** | **21** | **4500** | **1740** | 1075 |

Table 1: The upper table compares average total reward for various learning methods by running an $\epsilon$-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an $\epsilon$-greedy policy with $\epsilon = 0.05$.

## Conclusion

- The proposed deep learning model, using only raw pixels as input, produces superhuman results in six of the seven games it was tested on, utilizing the same architecture.
- The Deep Q-Network successfully verifies the viability of online Q-learning that combines stochastic minibatch updates with experience replay memory to ease the training of deep networks for reinforcement learning.