

# Applications of Deep RL in arcade game environments

## Abstract

We use recent advances in training deep neural networks to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games. We demonstrated that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previously discovered algorithms according to the research paper “**Playing Atari with Deep Reinforcement Learning**”. As a result, we were able to achieve an average score level comparable to that of a professional human games tester in Space invaders using the Space invaders-v4 Open AI gm, with the same algorithm, network architecture, and hyperparameters.

## Challenges Faced:

Understanding the way to convert a paper to code, reviewing multiple complex research papers, watching some YouTube tutorials to help learn the fundamentals of Deep Learning and also the dive from RL to Deep RL which enabled us to make and better the algorithms, choosing a suitable atari game.

## Introduction

“**Reinforcement learning** is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward.” -(Wikipedia). The agents learn by interacting with the environment. Agent encounters a state, decides an action to perform and receives a reward by performing the action, and moves to another state in the environment.

The input for the agent is the current state, and the agent maintains a Q-function based on which the agent makes decisions accordingly. So, the agent should initially identify the current state. If we provide numerical data, the agent can understand the state and perform an action accordingly. But in a real-world scenario, the agents get the **input through high-dimensional** sensory inputs, images, signals, etc., and the agent has to derive representations from these inputs. The agent has to identify the state, and after performing appropriate action, it has to update the Q-function values. To accomplish all this, we can use the “Deep Learning” techniques to directly understand the high-dimensional inputs and compute the Q-function values. “**Deep neural networks**” can be used to derive data from high-dimensional data such as images. If we know Q-values, we can run a Reinforcement Learning (RL) algorithm, which decides the actions. This combination of Deep Learning and Reinforcement learning is known as “**Deep Reinforcement Learning**.”

The research paper “Human-level control through deep reinforcement learning” has proposed a Deep reinforcement learning algorithm called the “**Deep Q Network (DQN)**,” where a deep neural network is used to learn policies directly from high-dimensional input. We will be using a “**Deep Convolutional network**,” which uses hierarchical layers of filters that derive useful correlations and data from the images. This network is used to find or approximate the action-value function or the Q-function  $Q(s,a)$ . From the Q-function values, we can choose a greedy action that is the Q-function’s optimal value. The general equation of the optimal Q-value is shown below.

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a, \pi]$$

As we calculate the Q-values using a deep neural network, a nonlinear function approximator, the RL algorithm may become unstable as the network is stochastic. If we use this in Q-function, then the stochastic values can lead to wrong decisions as the decisions are very sensitive to the action-values of the Q-function. The RL agent may also diverge or learn wrong as the observations are correlated. To overcome this, the paper used two tricks. One is “experience replay,” which shuffles the observations and removes the correlations. Another is to maintain two networks. One network updates iteratively, and the other is just the copy of the first network but updated or copied periodically, which acts as a target network.

## DQN

Deep Q-Network (DQN) is a deep reinforcement learning algorithm that uses deep neural networks to estimate Q-function. The neural network used is a deep convolutional network consisting of three convolution layers and two linear layers. ReLU activation function was used. The neural network takes high-dimensional input or, in simple words, the state representation as input and estimates the Q-function. The neural network computes the Q-values for all the possible actions from the input, which is the current state.

The input image needs to be preprocessed to remove flickering and reduce input dimensionality and some preprocessing required for the atari environment. After understanding the state, the actions are selected based on an e-greedy policy based on  $Q(s,a)$ . The action is performed on the emulator, and we receive the output image, which is again preprocessed and stored as a state representative. To perform experience replay, we have to store all these experiences: the state, action, reward, and observation pairs. Experience  $e$  can be represented as  $e_t = (s_t, a_t, r_t, s_{t+1})$ . While training the agent, we randomly pick a sample of experiences and update the Q-function. By doing this, we remove the correlation between the experiences, and we can send a batch of experiences as training data for the deep neural networks.

In short, the **deep Q network calculates the action values  $Q(s,a,\theta)$**  where  $\theta$  are the parameters of the network. The DQN maps the high-dimensional input to a vector of action values. If we consider the input dimension as ‘n’ and the number of actions is ‘m’. Then the DQN maps the n-dimensional input to m-dimensional output, which is the action-values for ‘m’ actions.

## Neural network logic and estimation of Q-values:

We take a sample of experiences and send them through the network and optimize the network by minimizing the loss. The loss function is the difference between the predicted Q-value and the target Q-value. The predicted Q-value is obtained by passing the state as input to the neural network, and the target Q-value is reward+(gamma\*optimal Q-value). The loss function is shown in the figure below.

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

Where  $\theta_i$  are the parameters of the Q-network at  $i$ th iteration and  $\theta_i^-$  are the parameters of the second network or the target network. We iteratively update  $\theta_i$  values(weights) of the first network using the deep learning methods. And also, periodically updating the  $\theta_i^-$ , weights of the second neural network equal to the first one. This second neural network acts as a target network. Thus we optimize the target neural network to predict the actions based on the optimal Q-value. The network's final output can be considered the Q-values, the parameter for selecting the action.

## Pseudocode of the algorithm:

### 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  $\epsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$


 e-greedy action

        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  $D$

 experience replay

        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  $\hat{Q} = Q$

 updating the target network parameters

**End For**

**End For**

## Implementation:

Tensorflow ,Keras for creating and training neural network

PyTorch as well for same purposes

OpenAI Gym Atari for creating SpaceInvaders environment

Wrappers for processing gym environment from Common Baselines library

**The implementation code - vanilla dqn**

<https://github.com/nicknochnack/KerasRL-OpenAI-Atari-SpaceInvadersv0>

## Results

```
Episode 1: reward: 320.000, steps: 724
Episode 2: reward: 225.000, steps: 569
Episode 3: reward: 400.000, steps: 1061
Episode 4: reward: 310.000, steps: 887
Episode 5: reward: 215.000, steps: 1074
Episode 6: reward: 230.000, steps: 827
Episode 7: reward: 385.000, steps: 1145
Episode 8: reward: 245.000, steps: 717
Episode 9: reward: 185.000, steps: 545
Episode 10: reward: 110.000, steps: 461
262.5
```

## Improvement

### Policy:

Our Epsilon Greedy policy is linearly annealed , i.e. the value of Epsilon changes as the training goes on, it starts at 1 and decays to 0.1 over 1m steps and remains constant after that.

### Pre-Processing And Frame Stacking:

The Observation space outputted by Gym's environment is 210x160 and is color i.e. 3 channels. This gives much unnecessary information which is not needed and slows the training considerably. Therefore we have converted the environment to grayscale and cropped out a 84x84 square which contains all required elements.

In addition , we have implemented frame-stacking, the agent considers the last 4 frames while observing, this is done as vectors like laser movement, movement of spaceships and speed cannot be determined by a single frame.

## The problem with DQN:

The above DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain. In basic Q-learning, the optimal policy of the Agent is always to choose the best action in any given state. The assumption behind the idea is that the best action has the maximum expected Q-value. However, the Agent knows nothing about the environment, in the beginning, it needs to estimate  $Q(s, a)$  at first and update them at each iteration. Such Q-values have lots of noise, and we are never sure whether the action with maximum expected/estimated Q-value is really the best one. Unfortunately, the best action often has smaller Q-values compared to the non-optimal actions in most cases. According to the optimal policy in basic Q-Learning, the Agent tends to take the non-optimal action in any given state only because it has the maximum Q-value. Such a problem is called the overestimation of action value (Q-value). The noisy estimated Q-values caused due to overestimation lead to large positive biases in the updating procedure of the Q-value. This is because the loss function discussed above depends on the current  $Q(s, a)$ , which is very noisy.

Overoptimistic value estimates are not necessarily a problem in and of themselves. Suppose all values would be uniformly higher i.e., equally overestimated. In that case, the relative action preferences are preserved, and we would not expect the resulting policy to be any worse since these noises don't impact the difference between the  $Q(s', a)$  and  $Q(s, a)$ .

## The solution to overestimation - Double Deep Q-Learning:

The max operator in DQN(mentioned below) uses the same values both to select and to evaluate an action :

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-)$$

This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation, which is the idea behind Double Q-learning.

In the original Double Q-learning algorithm, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights,  $\theta$ , and  $\theta'$ . For each update, one set of weights is used to determine the greedy policy and the other to determine its value.

For a clear comparison, we can first untangle the selection and evaluation in Q-learning and rewrite its target as :

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}} Q(S_{t+1}, a; \theta_t); \theta_t)$$

The Double Q-learning error can then be written as :

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}} Q(S_{t+1}, a; \theta_t); \theta'_t)$$

Notice that the selection of the action, in the argmax, is still due to the online weights  $\theta_t$ . This means that, as in Q-learning, we are still estimating the value of the greedy policy according to the current values, as defined by  $\theta_t$ . However, we use the second set of weights  $\theta'_t$  to fairly evaluate the value of this policy. This second set of weights can be updated symmetrically by switching the roles of  $\theta$  and  $\theta_t$ .

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function without introducing additional networks. However, we can evaluate the greedy policy according to the online network, but using the target network to estimate its value leads to the Double DQN algorithm. Its update is the same as for DQN, but replacing the target  $Y_t$  DQN with

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}} Q(S_{t+1}, a; \theta_t), \theta_t^-)$$

In comparison to Double Q-learning, the weights of the second network  $\theta'_t$  are replaced with the weights of the target network  $\theta_t^-$  -- for evaluating the current greedy policy. The update to the target network stays unchanged from DQN and remains a periodic copy of the online network. This version of Double DQN is perhaps the minimal possible change to DQN towards Double Q-learning. The goal is to get most of the benefit of Double Q-learning while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead.

## Implementation of DDQN:

<https://drive.google.com/drive/folders/17Px4Q0jPYZpnSq2HpalCncbEyzVdndHi?usp=sharing>

The pre-processing step, deep neural network architecture, is done by pytorch instead. Added an experience buffer of size 1m for usage by experience replay. The only change is in the calculation of the 'y' value or the expected reward, two networks are used , one online and one target. In DQN, the action which has maximum state-action value was selected, and 'y' was set as reward + gamma \* Q(s,a,θ). So here, the action is chosen using the same network, and the return is calculated using the same. But in DDQN, the action is chosen based on the maximum action value of the first network (which we here considered as another network other than the target network). Still, the return is calculated using the target network. In DDQN, 'y' is equal to reward +( gamma \* Q(s,a,θ - ) ). The network is trained similar to the DQN, where a sample is taken and performed training. But here, the expected return or the 'y' value is different, which is the 'y' based on the Double-Q learning algorithm. The loss function is the Smooth-L1(huber loss) of the target value 'y' and predicted value Q(s,a,θ).

## Pseudocode of the algorithm:

```

Initialize replay memory  $D$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights  $\theta$ 
Initialize target action-value function  $\bar{Q}$  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  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ 
        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  $D$ 
         $a\_max = \operatorname{argmax}_a Q(\phi_{j+1}, a; \theta)$ 
        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma Q(\phi_{j+1}, a\_max; \theta^-) & \text{otherwise} \end{cases}$ 
        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  $\bar{Q} = Q$ 
    End For
End For

```

Annotations:

- $\Rightarrow$  e-greedy action (points to the selection of  $a_t$ )
- $\Rightarrow$  experience replay (points to the sampling from  $D$ )
- } Double-Q learning (bracketed around the  $y_j$  calculation)
- $\Rightarrow$  updating the target network parameters (points to the reset of  $\bar{Q}$ )

```

GAMMA=0.99
BATCH_SIZE=32
BUFFER_SIZE=int(1e6)
MIN_REPLAY_SIZE=50000
EPSILON_START=1
EPSILON_END=0.1
EPSILON_DECAY=int(1e6)
NUM_ENVS = 4
TARGET_UPDATE_FREQ=10000 // NUM_ENVS
LR = 1e-4

```

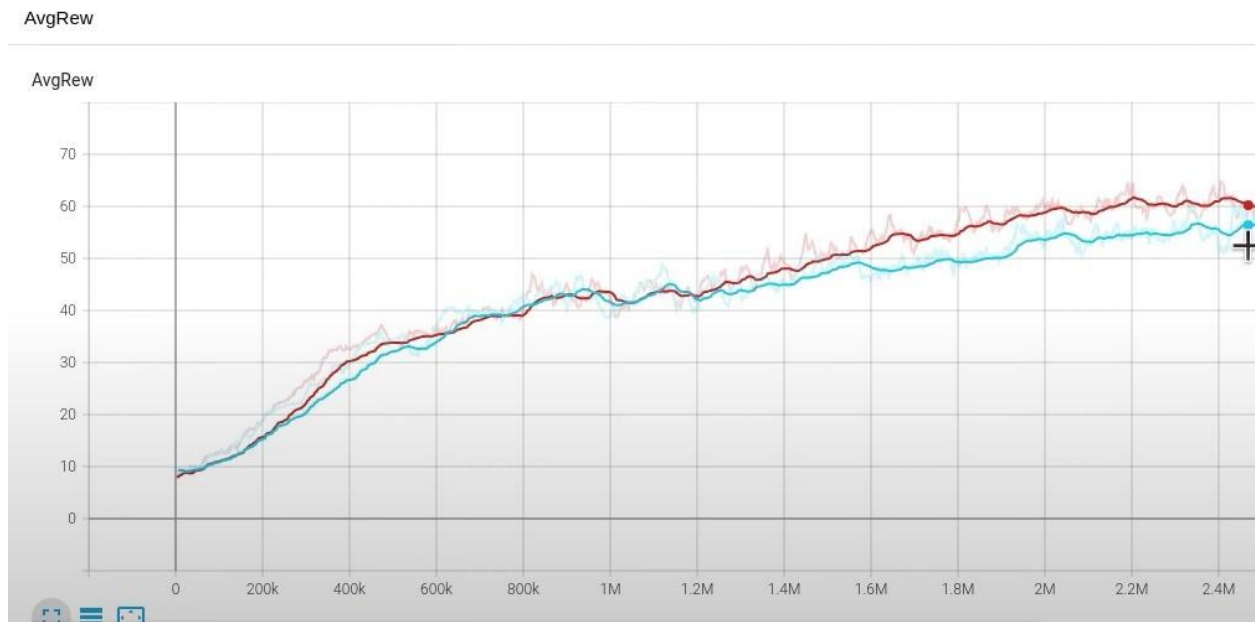
## Results

```

Episode 1: reward: 560.000
Episode 2: reward: 415.000
Episode 3: reward: 580.000
Episode 4: reward: 415.000
Episode 5: reward: 720.000
Episode 6: reward: 1090.000
Episode 7: reward: 720.000
Episode 8: reward: 685.000
Episode 9: reward: 415.000
Episode 10: reward: 575.000
617.5

```





**DQN(blue) vs DDQN(red)** both with optimizations implemented

- Here AvgRew is not score agent but gives an idea of it for comparison purposes

## Observation

In DQN, an agent actively goes out of its way to target the mothership ignoring other ships, which causes the round to end once they reach the ground. However, in DDQN, it has learned to prioritize and hit it in between hitting other ships.

It also learned to prioritize ships nearing the ground as they would end the episode, reducing score.

These two were important learning steps caused by experience replay and target network that enabled the agent to increase score drastically.

Double DQN also oscillates much less than DQN and is more likely to converge to a value due to usage of two networks.

## Conclusion

In arcade/atari game environments, the agent can only see the images which are high-dimensional vectors. The agent has to understand the state, choose appropriate action, perform an action, and understand the result state. So, the agents have to understand the high-dimensional image and calculate the Q-values for all possible actions. This is made possible by the use of deep neural networks, which take high-dimensional input like images and give the vector of Q-values for all possible actions. We can then choose appropriate actions based on the Q-values. The agent learns by playing and training the neural network using these experiences. After rigorous training, the neural network can now accurately calculate the Q-values for new input, and the agent can perform the optimal action. The Agent performs well



with an average score of 617.5 which doesn't beat the best human performance but is still remarkable. This is due to the nature of the game itself( Space Invaders ) which requires the agent to form long term choices, unlike breakout or pong. Another problem was computational and time constraints as these networks require long amounts of time to train(16 hours for 750,000 iterations)

The DQN agent had overcome a few challenges of using the non-linear neural network approximators using experience replay and usage of two networks. But, there were a few problems in DQN, which were overestimating the Q-values and initial noise. We came up with the idea of using Double Q-learning instead of Q-learning to overcome these problems. Learning is much more stable with Double DQN, suggesting that the cause for these instabilities is in fact, Q-learning's overoptimism. The Double DQN agent performed better than the DQN agent. Double DQN does not just produce more accurate value estimates but also better policies.

## Contribution of group members

Name	ID	Contribution
Vansh Agrawal	2021A7PS2998H	Searched for improvement and wrote a few parts of the report.
Aryan Saluja	2021A7PS2947H	Implemented the code and recorded observations
Guthula Baladitya	2019A7PS0067H	Collected research papers and wrote few parts of the report
Vundavalli Harshavardhan Chowdary	2019A7PS0044H	Explained about the implementation (DQN) and improvement (DDQN) in the report
Chodiseti Rajesh	2019A7PS0093H	Searched for improvement and wrote a few parts of the report.
Madepalli Balu Pavan	2019A7PS0061H	Implementation of the paper and searched for an improvement