Deep Reinforcement learning on Atari Games

Othman SBAI
othman.sbai@eleves.enpc.fr

El Amine CHERRAT
el-amine.cherrat@polytechnique.edu

Abstract

For our project for Reinforcement Learning course, we studied the use of deep neural networks to resolve RL Problems. We based our approach on the 2015 Nature paper published by Mnih et al [9]. We first formalize the problem before giving the basic deep Q-learning algorithm followed by the architecture of the neural network. We then present improvements introduced by the *DeepMind* team in their algorithm. Finally, we present our numerical results obtained with open source code from the same team.

1 Introduction

The aim of reinforcement learning is to train an agent to discover the policy that maximizes the agent's performance in terms of the discounted future reward, while interacting with a certain environment, receiving only a reward signal. The agent can take actions in a set of possible actions based on a policy that maps each state with actions to take.

The discounted future reward is written as, where γ is the discount factor:

$$V_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Q-learning is one reinforcement learning algorithm which estimates the Q (as quality) function of a couple (state, action) with samples during the learning phase, through an iterative approximation based on the Bellman equation.

For environments with large state space and action space, Q-function can be approximated with different approximators such as neural networks, decision trees, nearest neighbors. Neural networks are the most investigated across reinforcement learning community, researchers have been trying to train neural networks to learn policies through Q learning algorithm by interacting with an environment [1],[7],[12], however this has been a challenging task for various reasons: reinforcement learning algorithms must learn policies from scalar rewards which are usually delayed in time, also the states can be very correlated.

Deep Q-Leraning is one among many RL algorithms that can be adapted with deep learning function approximation. Thanks to recent improvement of computation power and infrastructure (powerful GPUs, AWS), deep learning libraries, and new reinforcement learning environment platforms and toolkits for training and evaluating RL algorithms such as openAI Gym and Universe, Arcade Learning Environment for Atari games, TorchCraft, Deepmind Lab etc ... Many ideas and research are being carried out in this topic of learning artificial agents from games for decision taking under uncertainty.

One application field of Deep Q-Networks consists of agent playing a game. We will formalize the problem in the next chapter.

2 Formalization of the Reinforcement Learning Problem

We will use a deep Q network to train our agent, but we have to specify first the parameters that define our reinforcement learning problem. This approach can be generalized to all Atari 2600 games. Next, we will specify details of the environment where the agent takes actions but we have to specify how do we encode a single observation based on the current screen.

Set of states

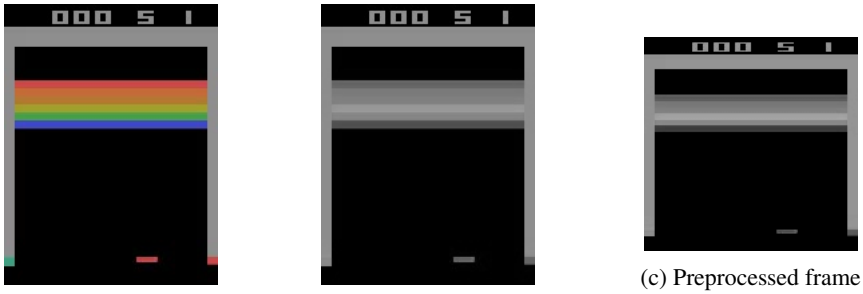
Atari 2600 frames are of size `210x160` pixels with a `128-color`. Working directly with this data of size `210x160x3`, without processing, can be very demanding in terms of computation and memory requirements. The *DeepMind* team did a preprocessing to the original Atari 2600 frames in order to reduce the input dimensionality. This processing will be denoted by ϕ.

Let's denote by $\mathbf{x}_t$ the current screen at time t. To encode a single frame we take the maximum value for each pixel colour value over the frame being encoded and the previous frame. Second, we then extract the Y channel, also known as luminance, from the RGB frame and rescale it to `84x84`.

If (R, G, B) are the RGB parameters of the rescaled frame, then the preprocessed frame is :

$$\phi(\mathbf{x}_t) = 0.21 \cdot \max(R(\mathbf{x}_t), R(\mathbf{x}_{t-1})) + 0.71 \cdot \max(G(\mathbf{x}_t), G(\mathbf{x}_{t-1})) + 0.07 \cdot \max(B(\mathbf{x}_t), B(\mathbf{x}_{t-1}))$$

The terminal state $\mathbf{x}_T$ corresponds to the end of the game.



(a) Original frame (b) Before rescale (c) Preprocessed frame

Figure 1: Preprocessing Atari 2600 frames of the `BREAKOUT` game

Set of possible actions

The Atari 2600 gamepad is a combination of joystick and a play button. Actions that can be taken by our agent are 8 possible moves of joystick, play button, combinations of play button and one joystick move. The agent can also choose to do nothing. Then, the number of possible actions is $8 + 1 + 8 + 1 = 18$ as we can see below (we denote the play button by $\odot$). However in the environment used in the code, each game has a specific number of possible actions (breakout: 4).

Table 1: Set of actions

No input	$\leftarrow$	$\nwarrow$
$\uparrow$	$\nearrow$	$\rightarrow$
$\searrow$	$\downarrow$	$\swarrow$
$\odot$	$\odot + \leftarrow$	$\odot + \nwarrow$
$\odot + \uparrow$	$\odot + \nearrow$	$\odot + \rightarrow$
$\odot + \searrow$	$\odot + \downarrow$	$\odot + \swarrow$

Set of rewards

At each time step t, the agent receives reward $\mathbf{r}_t \in \{0, 1\}$ while the screen displays cumulative reward since the beginning of episode.

Our problem now is to train agent to find optimal play policy in order to maximize cumulative reward.

3 Using Deep Q-Learning to find optimal policy

A Q-Learning approach

A policy π is a behaviour function that returns action $\mathbf{a} = \pi(\mathbf{x})$ given an observation $\mathbf{x}$. The aim of problem to maximize the expected cumulative reward over the episode $\mathbb{E}(\sum_{t=1}^{T} \gamma^{t-1} r_t)$. To do so, we define the Q function that outputs the expected total reward from state $\mathbf{x}$ and action $\mathbf{a}$ under policy π :

$$Q^{\pi}(x, a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | x, a] = \mathbb{E}_{x', a'}[r + \gamma Q^{\pi}(x', a') | s, a]$$

The optimal state value function verifies $Q^*(x, a) = \mathbb{E}_{x'}[r + \gamma \max_{a'} Q^*(x', a') | x, a]$ for all observations $\mathbf{x}$ and possible actions $\mathbf{a}$. State $\mathbf{x}'$ is new state when taken action $\mathbf{a}$ from state $\mathbf{x}$. We will try then to minimize the objective function defined as the mean squared error :

$$L(\pi) = \frac{1}{2} \mathbb{E}_{s,a,s'} \left[\left((r + \gamma \max_{a'} Q^{\pi}(x', a')) - Q^{\pi}(x, a) \right)^2 \right]$$

A Neural Network to approximate Q

In order to approximate Q^*, we will build a neural network with parameters θ. The neural network defines a policy π such that : $Q(x, a, \theta) \approx Q^{\pi}(x, a)$. The best action to take is computed by a forward pass through the network with current state as input. During training phase, we will change parameters of network θ in order to minimize the objective function by stochastic gradient descent using:

$$\frac{\partial L(\theta)}{\partial \theta} = \mathbb{E}_{x,a,x'} \left[\left((r + \gamma \max_{a'} Q(x', a', \theta)) - Q(x, a, \theta) \right) \frac{\partial Q(x, a, \theta)}{\partial \theta} \right]$$

Basic Deep Q-Learning algorithm

In the basic Deep Q-Learning algorithm, we start by randomly initializing the weights of the network θ, which is a random initialization of the state action function Q. During training phase, agent plays M episodes of the game. We denote by T the terminal state when the game is over, each episode has duration T in time. The score over the episode is $\sum_{t=1}^{T} r_t$ and in our approach we will try to maximize the expected cumulative reward $\mathbb{E}(\sum_{t=1}^{T} \gamma^{t-1} r_t)$, with $\gamma \in]0, 1[$, to ensure the convergence of the Q-Learning algorithm to optimal policy. At each episode, the agent, in state s_t, chooses to play the action a_t that maximizes $Q(\phi(s_t), a; \theta)$. After receiving reward r_t, the agent moves to next screen x_{t+1} and next state $s_{t+1} = s_t, a_t, r_t, x_{t+1}$. The agent uses a second time the network to get $r_t + \max_{a'} Q(\phi(s_{t+1}), a'; \theta)$ and sets it to the target value of action a_t. For all other actions, the target values are those predicted by network previously which makes the error equal to 0 for these outputs.

Algorithm 1 Basic Deep Q-Learning

Initialize action value function Q with random weights θ
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(x_1)$
  for $t = 1, T$ **do**
    Do a feedforward pass for s_t to get predicted $Q(\phi(s_t), a; \theta)$ for all actions a
    Execute action $a_t = \text{argmax}\, Q(\phi(s_t), a; \theta)$ and observe reward r_t and next state x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Do a feedforward pass for s_{t+1} and calculate $r_t + \max_{a'} Q(\phi(s_{t+1}), a'; \theta)$
    Set target value $y_j = r_t + \max_{a'} Q(\phi(s_{t+1}), a'; \theta)$ for $a_j = a_t$
    Set target value $y_j = Q(\phi(s_t), a; \theta)$ for $a_j \neq a_t$
    Perform a gradient descent step on $(y_j - Q(\phi(s_t), a_j; \theta))^2$ with respect to parameters θ
  end for
end for

4 Details about the neural network

Advantages of using a neural network

Neural networks have proven that they can give very high representation of the data. Using them to solve a reinforcement learning shows that they can learn how to play efficiently a game without any specific information. Even if we will test the convergence of the algorithm on the specific game : `BREAKOUT`, the network is able to run on different games. This is a very important point because the `BREAKOUT` game can be easily represented with parameters such as direction of the ball and location of elements, but we won't need this interpretation. The network can play different games and switch between them easily. We can say that in this case that the agent is able to learn how to play in general and not a specific game in particular.

The *DeepMind* team represented the Q-Learning problem with a neural network that takes the current state as input and output the corresponding action-value function values for all actions with one forward pass through the network. The agent will play the action that maximizes those values.

Input and output of the neural network

To produce the input of the neural network, the function ϕ applies the preprocessing to the m most recent frames and stacks them to produce the input to the Q-function, in which $m = 4$, although the algorithm is robust to different values of m (for example, 3 or 5).

The preprocessing map ϕ produce the input to the neural network of size `84x84x4` :

$$\phi(\mathbf{s}_t) = (\phi(\mathbf{x}_{t-3}), \phi(\mathbf{x}_{t-2}), \phi(\mathbf{x}_{t-1}), \phi(\mathbf{x}_t))$$

The output of the neural network of size `18` based on the number of possible actions.

Architecture of the neural network

The neural network, described in [9] and designed by the *DeepMind* team, is a convolutional neural network. Convolutional neural networks are very efficient with image recognition as shown in [5] because they can extract high representation of structured data. The deep Q-Network will extract features that give information to agent such as location of different elements.

The network's architecture is made up with input reshaping layer and output fully connected layer to reduce the size to action space and intermediate convolution and rectifier layers as shown in figure (2).

As explained before, the input layer inputs the preprocessed most recent four frames to for the first convolutional layer. This layer convolves 32 filters of size 8×8 with stride 4, which means that these layers extract and output 32 subframes of size $\frac{2(84-4)}{8} \times \frac{2(84-4)}{8} = 20 \times 20$. This following process is then repeated with two other convolutional layers. The second layer convolves 64 filters of size 4×4 with stride 2 and output 64 subframes of size $\frac{2(20-2)}{4} \times \frac{2(20-2)}{4} = 9 \times 9$. Then, the last layer convolves 64 filters of size 3×3 with stride 1 and output 64 subframes of size 7×7. This layer is fully connected to linear layer that reduces size of input to 512 units. The output layer is then fully connected to the 512 units and output the predicted action values for all 18 possible actions.

```
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) -> (10) -> (11) -> output]
  (1): nn.Reshape(4x84x84)
  (2): nn.SpatialConvolution(4 -> 32, 8x8, 4,4, 1,1)
  (3): nn.Rectifier
  (4): nn.SpatialConvolution(32 -> 64, 4x4, 2,2)
  (5): nn.Rectifier
  (6): nn.SpatialConvolution(64 -> 64, 3x3)
  (7): nn.Rectifier
  (8): nn.Reshape(3136)
  (9): nn.Linear(3136 -> 512)
  (10): nn.Rectifier
  (11): nn.Linear(512 -> 4)
}
```

Figure 2: Network architecture by *DeepMind* team

5 Improving the basic Deep Q-Learning Algorithm

The Basic Deep Q-Learning algorithm aims at estimating the action-value function but it turns out that in practice it may face different problems :

- Action value function Q is initialized randomly. At first iterations of the algorithm, using exploitation will give random results as well.
- Correlations between states when playing one episode causes the algorithm to diverge.
- Estimation of value function Q with neural networks may diverge since small updates of network parameters may lead to complete change of the network's policy.

Using Exploration-Exploitation

To resolve the exploration-exploitation dilemma, the *DeepMind* team used a classical ϵ-greedy policy. The agent chooses with probability ϵ a random action, with ϵ decreasing linearly from 1 to 0.1 between the first millions frames and then fixed to $\epsilon = 0.1$. The algorithm is making totally random choices in the beginning, since weights are randomly initialized and there is no useful exploitation.

Using experience replay

In order to remove existing correlations between samples when optimizing network parameters, we should perform gradient descent using samples given by different episodes. Using the experience replay method allows removing previous correlation and improves the convergence of algorithm [7],[10].

At the beginning of training phase, the agent creates and initializes a relay memory with finite capacity. During iterations, the agent stores each transition of states, perceived reward and performed action $(\phi_t, a_t, r_t, \phi_{t+1})$ in the replay memory D. Instead of using only the most recent transition to update network parameters θ, the agent samples a random mini batch of transitions to perform a stochastic gradient descent.

Using target action-value function

In order to remove oscillations or divergence in estimation of the action value function, the *DeepMind* team introduced the target action-value function Q^- with parameters θ^-. Instead of updating network parameters θ at each iteration, the agent will limit this update to periodic steps and uses Q^- to estimate actual target values.

Based on previous improvements, the *DeepMind* team introduced the following algorithm :

Algorithm 2 Improved Deep Q-Learning

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q^- with random weights $\theta^- = \theta$
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(x_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \text{argmax}\, Q(\phi(s_t), a; \theta)$
    Execute action a_t and observe reward r_t and next state 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 memory D
    Sample random mini-batch size of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set target value $y_j = r_j$ for $a_j = a_t$ if episode terminates at $j + 1$
    Set target value $y_j = r_j + \max_{a'} Q^-(\phi(s_{j+1}), a'; \theta^-)$ otherwise
    Perform a gradient descent step on $(y_j - Q(\phi(s_t), a_j; \theta))^2$ with respect to parameters θ
    Every C steps, update $Q^- = Q$
  end for
end for

6 Numerical results

Experimental setup

We use the code provided by Google *DeepMind* team that was used for the experiments carried out in the Nature paper [9]. It is a Lua based code which makes use of Torch deep learning library for GPU computing speed-up.

We trained the agent on GPU with a single Nvidia GTX1060 on Ubuntu, the code provided contains a script that sets up the dependencies of the project.

At each step, the agent perceives the state from the screen, the reward signal and whether the state is terminal and then preprocesses the state, stores the transition in the replay memory, selects and action and performs Q learning updates at fixed step interval.

We used without modification the original code to obtain results when testing on the `BREAKOUT` game.

Table 1: Training parameters

Mini-batch size	32
Replay Memory size N	10^6
Discount factor γ	0.99
Target network update frequency C	10^5
Learning rate	2.5×10^{-4}

We run the training loop for 50.10^6 game steps, and test the performance of the agent at each 250.10^3 game steps for another 125.10^3 steps. On average, training for a million steps takes about 4 hours and testing for a million steps takes about 2.75 hours with the specified hardware setup. Training on a CPU would take 30 times more.

Importance of using replay memory

As explained in the paper [9], the training is performed on mini batch samples from the replay memory. The default setup was sampling one time 32 transitions and performing Q learning backward pass to change network's weights. We used our trained network after 5.10^6 steps and let it train setting the exploration probability directly to 0.1.

Using a pre-trained network means starting with an empty replay memory, which considerably lowers the performance of the training, as we can see in the figure (3). We can clearly see the lower performance of training starting from the same weights.

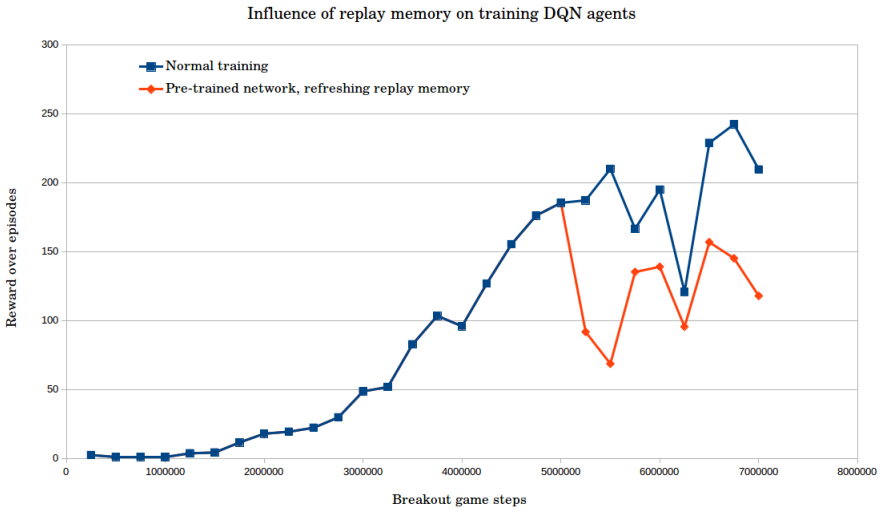
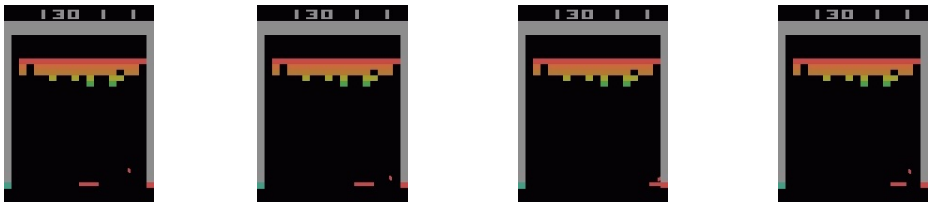


Figure 3: Influence of using replay memory on training DQN networks

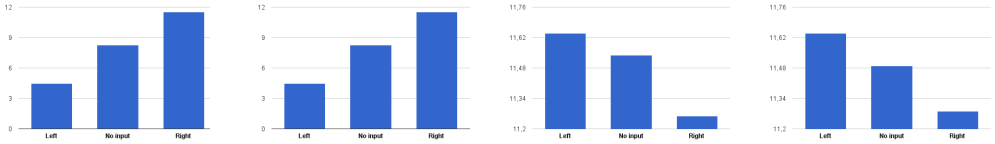
Testing on BREAKOUT game

After training the algorithm to play the BREAKOUT game, we tested it to obtain the following results.



(a) Frame at time t (b) Frame at time t+2 (c) Frame at time t+4 (d) Frame at time t+6

Figure 4: Screenshots of BREAKOUT game



(a) Go right (b) Go right (c) Go left (d) Go left

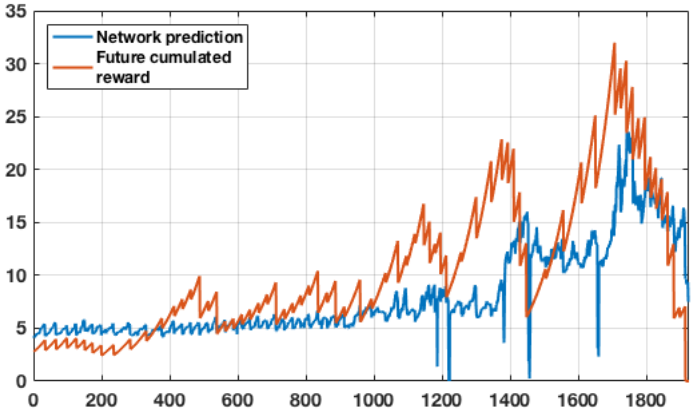
Figure 5: Corresponding estimated action-value function values

We tried to understand first how the agent makes the choices. To do so, we took some screenshots starting from a random state and we analyzed the corresponding prediction of Q-values for all possible actions $\in \{\leftarrow, \text{No input}, \rightarrow\}$ (Figure 4). We can see that agent predicts at each state action-value function values of the actual state. In frame (a), we can see that the agent understood that the ball is moving to the right and chooses the action that maximizes predicted Q values by the network.

Future cumulative reward prediction

We compare the network prediction of the best Q value of the current state versus the real value that is yielded after the game finishes. We observe that the network's prediction in blue are very correlated with the real reward obtained shown in orange.

$$Q^{\pi}(x,a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots | x, a] \quad vs \quad \frac{1}{\gamma^t} \sum_{t' \geq t} \gamma^{t'} r_{t'}$$



7 Conclusion

Neural nets for Reinforcement Learning

In this document, we have studied the use of neural networks to resolve Reinforcement Learning problems. As we have seen, the main strength of these algorithms is that they do not need any features and can directly extract them from their input. In fact, classical reinforcement learning algorithms needs handcrafted features related to the problem and their performance depends heavily on the quality of these features. The algorithm presented in this study can be applied to all ATARI 2600 games.

By testing our algorithm on the BREAKOUT and using the provided interface, we could see during training phase how the agent learns to play and maximize its score. Improvements introduced by the *DeepMind* team such as experience replay and periodical update of network parameters shows that it is possible for the agent to play different games without any adjustment of the neural network.

Limitations of the model

The previous studied model has several limitations due to its high spatial and temporal complexity. The training phase of one game can take up to one week using GPU and it needs lot of resources since size of inputs and network layers can be very demanding. BREAKOUT game properties are not complex and are based on simple curves and shapes. Even though, the algorithm couldn't converge quickly to the optimal policy. Applying the previous approach to more general problems may fail if the environment is too complex.

Future work

Emergence of deep reinforcement learning is a big step for Artificial Intelligence research field. After using neural network on ATARI 2600, the *DeepMind* used this new approach to create AlphaGO. Beating highest ranking GO players has proven the strength of Deep Reinforcement Learning.

Since the publication of studied article [9], many improvements to Deep Reinforcement Learning have been proposed :

- **Double Q-Learning :** (van Hasselt et al.,2015), another team from DeepMind showed in their publication [14] that the pevious deep Q-Learning algorithm has tendancy to overestimate action values for some games. The mean idea of the algorithm is to evaluate two action values functions by choosing randomly at each iteration which one to update.
- **Dueling network architecture :** (Wang et al., 2015), also a team from *DeepMind*, introduced another method to resolve the previous reinforcement learning problem [15]. The main idea is using two neural networks to estimate two separate parameters : one for the state value function and one for the state-dependent action advantage function. This method give better results for environments with many similar-valued actions.
- **Extension to continuous action space :** (Lillicrap et al., 2016) applied, in [6], Deep Q-Learning to continuous space actions. The hypothesis of independence between samples don't stand anymore and they changed their approach so that their algorithm could resolve 20 physical problems.

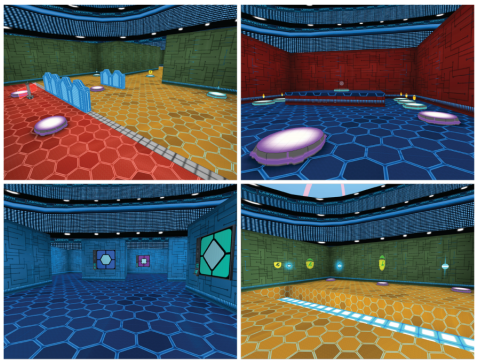


Figure 6: 3D environment for Reinforcement Learning Problems by DeepMind

References

[1] Charles W Anderson. Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 103–114, 1987.

[2] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

[3] Andrej Karpathy. Deep reinforcement learning: Pong from pixels. `http://karpathy.github.io/2016/05/31/rl/`, 2016.

[4] Akshay Krishnamurthy, Alekh Agarwal, and John Langford. Pac reinforcement learning with rich observations. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 1840–1848. Curran Associates, Inc., 2016.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[6] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[7] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[10] Joseph O'Neill, Barty Pleydell-Bouverie, David Dupret, and Jozsef Csicsvari. Play it again: reactivation of waking experience and memory. *Trends in neurosciences*, 33(5):220–229, 2010.

[11] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[12] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction, 1998.

[13] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

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

[15] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.