Learning Simple Games with Neural Networks

Kumar Ashutosh
16D070043

Mohd Safwan
17D070047

Manas Vashistha
17D070064

***Abstract*—Simple two-player games are a common brain-teaser amongst youngsters. These games involve sequential decision-making where one player tries to outperform others through their own strategy. Automating these games requires developing an agent that takes an action based on the current state of the game. In this work, we present a Reinforcement Learning (RL) agent which can compete against various RL algorithms in two common pen-paper games – tic-tac-toe and dots & boxes. Along with the agent, we also develop a training environment for RL agents where they can compete against these algorithms and learn to play the game. We upgraded the available interfaces and add features to have human vs agent competition of the game. We have trained our agent using Deep Q-Networks which is an implementation of a variation of Q-Learning. We demonstrate the effectiveness of this approach through detailed experimentations and analysis.** [1]

***Index Terms*—tic-tac-toe, dots-and-boxes, DQN, Reinforcement Learning, XO**

I. Introduction

Tic-Tac-Toe (*XO*) is a game for two players *X* and *O* who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. After each step a player has to make a decision about where to put his next mark so that his winning chances are maximized or at least the game moves towards a draw. So, at each instant player has $[0-8]$ actions to choose from: one out of the 9 boxes to put his mark.

Dots & Boxes is again a multi-player game where each player takes turn in marking an edge from an arbitrary mxn dotted grid. The player who is successful in completing a square gets one point. All the players are free to choose any edge when it is their chance. The player who captures a square gets one point. This game is played till all the edges on the board is taken up by a player. The player with highest point wins the game.

Reinforcement Learning (RL) is typically characterized by an agent (the learner & the decision maker) and environment (surroundings which interact with the agent). For every action taken by the agent, the environment provides a reward and a new state in return. So, instead of being supervised, the agent learns the actions it should take in order to achieve the goal, on its own. This task is administered by returning a positive or negative reward to the agent by the environment. Hence, a RL problem can be modelled mathematically using Markov Decision Process (MDP).

[1]https://github.com/safwankdb/XO-Learning-Environment

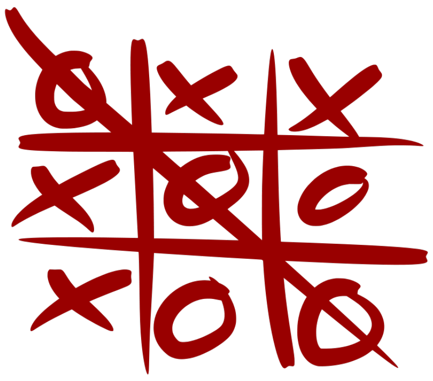


Fig. 1. Tic-Tac-Toe Game

Today, the recent advances in deep neural networks,in which several layers of nodes are used to build up progressively more abstract representations of the data,have made it possible for machine learning models to learn concepts such as object categories directly from raw sensory data. These methods utilise a range of neural network architectures, including convolutional networks (CNN) ,multilayer perceptrons and recurrent neural networks (RNN). These methods have worked really well for both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

According to Sutton and Barto (1998) the goal of reinforcement learning is to learn good policies for sequential decision problems, by optimizing a cumulative future reward signal. Q-learning (Watkins, 1989) is one of the most popular reinforcement learning algorithms, but it is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values.

Under a given policy π, the true value of an action a in state s is given by

$$Q^{\pi}(s,a) = \mathbb{E}[r^1 + \gamma r^2 + \cdot | S_0 = s, A_0 = \pi(s)]$$

where $\gamma \in [0,1]$ is discount factor. We can decide the action an agent should take from a given state s which is the action which maximizes the action value function for that state i.e. $Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$ and $\pi^*(s) = arg \max_{a \in A} Q^*(s,a)$.

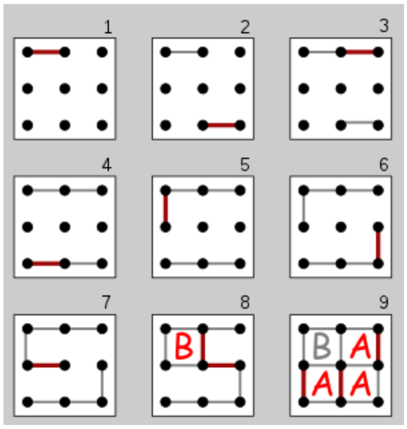


Fig. 2. Dots & Boxes Game

In a typical MDP, if there are finite number of states and actions then we can create a look up table for every state specifying the action an agent should take from that state (called the policy followed by the agent) and the next state the agent will reach after collecting the reward returned by the environment. But, in this case, the number of states is quite large (3^9) and hence it is not efficient to generate a look up table for each state. To tackle this, we try an approach where the action taken by an agent from a given state is approximated using a DNN. This method is quite common among the RL engineers and hence we try to experiment it with our game.

II. Related Work

A. Neural Fitted Q-Iteration

Martin Riedmiller [1], in his work addresses the question, "How can we exploit the positive properties of a global approximation realized in a multi-layer perceptron while avoiding the negative ones?" His idea was to provide the previous knowledge explicitly while an update was made at a new datapoint. The implementation required to store all previous experiences in terms of state-action transitions in memory. This data is then reused every time the neural Q-function is updated. He proposed an algorithm belonging to the family of fitted value iteration algorithms and referred to it as a special form of 'experience replay'.

B. Deep Q-Network

Google deepmind has been the pioneer of this research area for sometime now. Mnih et al. are able to successfully train agents to play the Atari 2600 games using deep reinforcement learning, surpassing human expert-level on multiple games [2]. Our work is motivated from this project and is a miniature version of an automated gameplay. They have approximated the Q-function for Q-learning using a deep Q-network (DQN) and have also used 'experience replay' for de-correlating the experiences. This work has inspired many other such projects. They outperform human expert on three out of seven Atari 2600 games.

C. AlphaGO Zero

AlphaGo Zero [3] was another outstanding work from Google deepmind. It was a computer program which plays the boardgame Go. It was a completely self-taught model without learning from any human-games. It defeated all its predecessors (AlphaGo Lee, AlphaGo Master etc.) in 40 days. Finally it competed along with the human experts and emerged as arguably the best Go player.

III. Analysis Pipeline

A. Game Interface

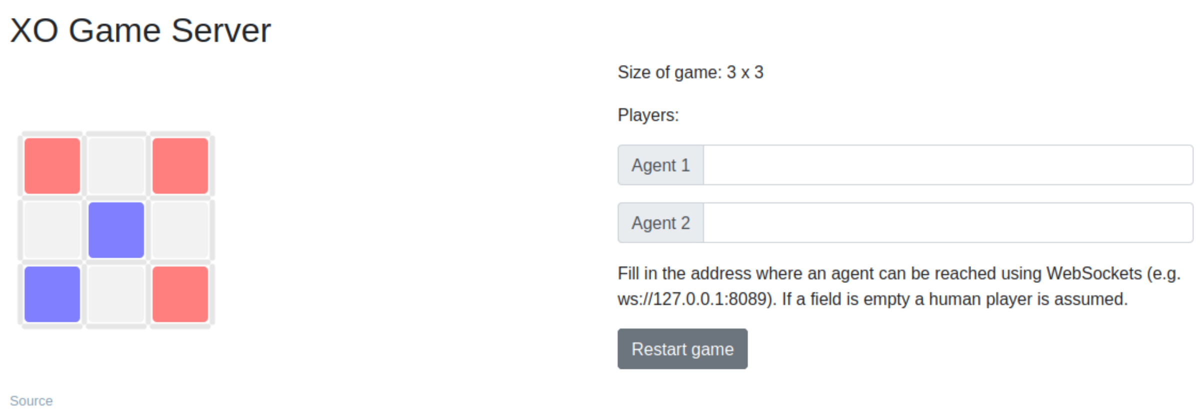
For tick-tac-toe, we develop a GUI for gameplay. The GUI can be used for both human vs human and human vs computer. The agents (or humans) communicates via web sockets. The central game server listens observes the state of the game and sends it to the participating web sockets. The corresponding agent takes turn and sends back their action to the server which updates the interface as well as the game states. For training the agent, we use a GUI-less server to repeat the game play arbitrary number of times. This GUI is an extension to a similar GUI developed for dots & boxes. An example screenshot from the GUI is shown in Figure 3.

B. MDP Formulation

The action our agent can take comes from the set $a \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. Each value corresponds to a cell (row major) in the 3×3 grid. The state at any instance can be calculated as: Each cell can have a X mark, a O mark or no mark at all. Since, there are three possibilities for each cell, if we combine these for all the 9 cells, we get a total of 3^9 possibilities (board states). Let at time instant t the board state be x_t. Then x_t combined with finite number of previous board states $(x_{t-1}, x_{t-2} \cdot)$ gives the current state. The number of previous board states becomes a hyper-parameter. Ideally, s_t should be a function of all board states from $t = 1$ but to reduce the state-space, only a finite number of board states are used.

The discount factor was set to $\gamma = 0.9$. The agent has no clue about the transition probabilities or the rewards it would get for a particular transition. Since, Q-learning is a model free approach we try to directly approximate the optimal Q-function instead of explicitly estimating the transition probabilities and rewards.

However, we need to define the rewards intrinsically so that the agent should learn in a particular way. We do it as: we know any horizontal, vertical or diagonal row has the same mark in it, the game is over. Hence we put a negative reward for losing the game or an invalid action and a positive reward only if the agent wins the game. This makes the playing agent to work like a human. Since, the maximum number of actions is five per episode, the final reward can propagate backwards easily.


XO Game Server

Size of game: 3 x 3

Players:

Agent 1

Agent 2

Fill in the address where an agent can be reached using WebSockets (e.g. ws://127.0.0.1:8089). If a field is empty a human player is assumed.

Fig. 3. GUI for the Tic-Tac-Toe Game

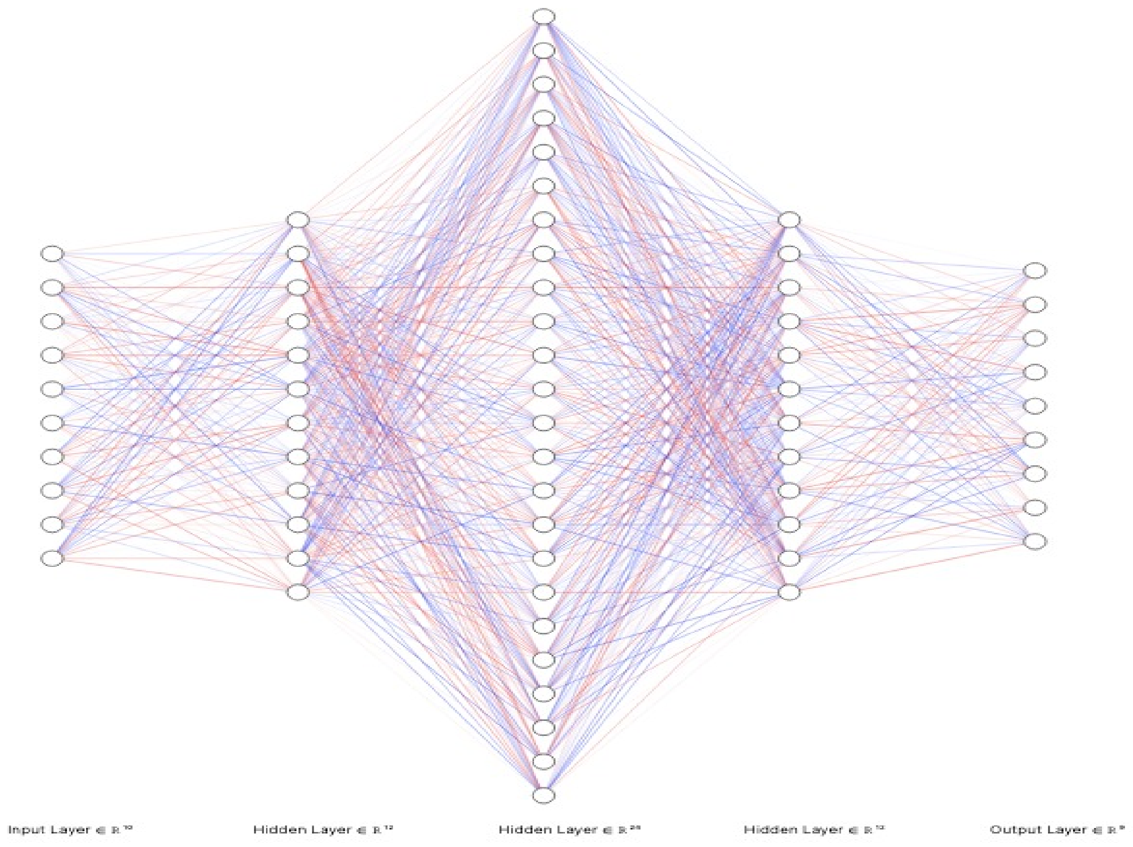

Input Layer $\in \mathbb{R}^{10}$
Hidden Layer $\in \mathbb{R}^{12}$
Hidden Layer $\in \mathbb{R}^{24}$
Hidden Layer $\in \mathbb{R}^{12}$
Output Layer $\in \mathbb{R}^{9}$

Fig. 4. Network Architecture where ReLU activation is used after every hidden layer. The input dimensions are 10, First hidden layer has dimensions 12, Second hidden layer has dimensions 24, Third hidden layer has dimensions 12 and the output layer has dimensions 9×1 representing the approximated Q-values for each action.

C. Q-Learning

Reinforcement Learning aims to maximize the expected value of the total return. Q-Learning is referred to be off-policy if we use the Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}_{s' \sim \epsilon}[r + \gamma \max_{a'} Q_i(s',a')|s,a]$$

where s' is the next state, r is the reward, ϵ is the environment and $Q_i(s,a)$ is the action value function at the ith iteration. It is proven that if we keep doing this update infinitely, it converges to the optimal action value function $\lim_{i \to \infty} Q_i(s,a) = Q^*(s,a)$, using which we can get the optimal policy $\pi^*(s)$. This iterative process becomes inefficient if we have a large number of states or actions. To avoid this, function approximation has to be used for the Q-function and this also generalizes the Q-function for the unseen states. Deep Q-Learning is one such method which is widely used for this approximation task, thanks to the universal approximation theorems [4]. A sequence of experiences, $e = (s^t, a^t, r^t, s^{t+1})$ becomes the input to the network and the network learns a Q-function which fits best for all such inputs. Experiences are analogous to datapoints and a list of experiences is called replay memory which is analogous to a dataset. Hence, the model which returns the Q-function favors actions which maximize the overall reward.

D. DQN Architecture

In the current architecture we have 3 hidden layers as shown in Figure 4. All the hidden layers are fully connected. After the hidden layers we have another fully connected layer. The output of the last fully connected layer is the Q-function approximation for each of the nine actions. There is a non-linear ReLU layer after all the hidden layers. The input is of size $9 + 1 = 10$. We assign a value from $-1, 0, 1$ to each cell. The value is decided on the basis of the mark present in the cell as: If opponent's mark, cell value is -1, if agent's mark, cell value is 1 otherwise 0. The values of all the nine cells at any time step becomes the input to the network. One extra dimension of the input represents the player who will be playing the next move. From the output we get an estimate of the Q-value for each of these nine cells to decide for the best.

E. Experience replay and stability

Q-Learning being a good approach also poses a problem. The experiences representing consecutive board states in an episode (a complete game) are often correlated. This leads to inefficient training and a poor approximation of the Q-function. Hence, we need to de-correlate these experiences. This can be done using experience replay where we store an experience (s^t, a^t, r^t, s^{t+1}) at every board state into replay memory. Due to limitations of storage, we fix a `REPLAY_SIZE` i.e. the maximum number of previous experiences that can be stored. To keep a track of actions taken by recent Q-function we constantly update the replay memory (like a queue). When updating the DQN, we sample some experiences randomly from the replay memory and add them to the training batch. Using this approach our experiences are no more correlated.

To improve the stability of the convergence of the loss functions, we clone the DQN model with parameters θ_{clone}. After every U updates to the original DQN, we update the cloned model and then this cloned model is used to compute the target y_i.

F. Training setup

In this section we describe how our work can be used as a learning environment for various RL gaming agents. We have trained our agent against various RL algorithms. These algorithms can further be used to create competitions and make other RL gaming agents learn through them. For learning, the RL agent will have to communicate through websockets which can be done easily and then learn against the desired algorithms. Also, we have designed a GUI using which human interaction with the agent is also possible quite easily.

As we mentioned that our model has been trained against different RL algorithms, next we describe these algorithms. The first algorithms is minimax search. It is a decision rule which is widely used in Artifical Intelligence and Decision Theory. The main aim is to minimize the possible loss for some worst case scenario. Here the value returned is the maximum value over all the agent's actions and over the minimum of all other player's actions. The next algorithm is $\alpha - \beta$ search. This algorithm aims to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. This is a common adversarial algorithm used for machine playing of two-player games. We train our model against these algorithms and compare the results in Figure 7 and Figure 8.

IV. Results

In this section we present the results we got from our implementation and the experiments we performed with our model.

A. Tic-tac-toe

We train and compare agents using different algorithms. The results are tabulated in Table I. Simple agent chooses action in a deterministic way. It chooses the lowest available action. Obviously, we can observe that such a strategy does not seem to do well. Next, have an agent that chooses action randomly. Finally, we have two important RL agents – MiniMax and AlphaBeta. MiniMax chooses the actions optimally and performs the best theoretically. We can also see here that MiniMax performs the best. Also, AlphaBeta outperforms Random and Simple agents by a big margin. We also graphically depict wins of our agent in the last 50 runs of the game. The results for all the agents are depicted in Figures 5, 6, 7 and 8. In all these figures, we can see an increasing number of wins by our agent.

B. Dots & Boxes

Similar to the above analysis of tick-tac-toe, we design an RL agent which is based on Q-learning, a standard RL technique. We compare this agent with a random agent that chooses edges randomly. We do on a 3x3 grid as a tradeoff between our compute power and high grid-size. We can see in this case as well that the Q-learning network performs better than the random agent. In fact, our agent is able to beat the random agent almost certainly within a few hours of training. A plot of the same is shown in Figure 9. We could not extend the board dimension due to and exponential increase in the state space.

TABLE I
PLAYER 1 IN LEFT, PLAYER 2 ON TOP. PERFORMANCE OF DIFFERENT AGENTS AGAINST EACH OTHER WITH NO SIDE SWAPPING. THE ENTRIES ARE PLAYER1 WINS - PLAYER 2 WINS - DRAWS

	MiniMax	AlphaBeta	Random	Simple
MiniMax	0-0-1000	1000-0-0	989-0-11	1000-0-0
AlphaBeta	0-1000-0	1000-0-0	859-88-53	1000-0-0
Random	0-815-185	208-598-194	582-310-108	545-427-28
Simple	0-1000-0	765-180-55	0-1000-0	1000-0-0

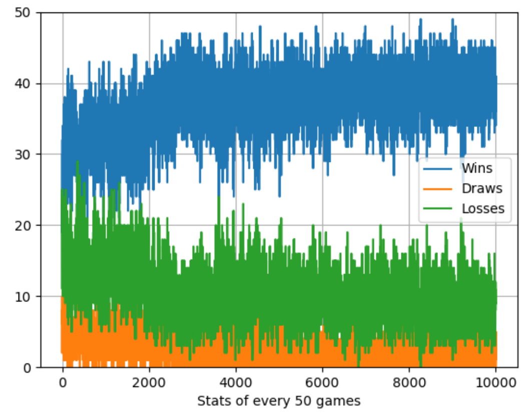


Fig. 5. vs Random Agent

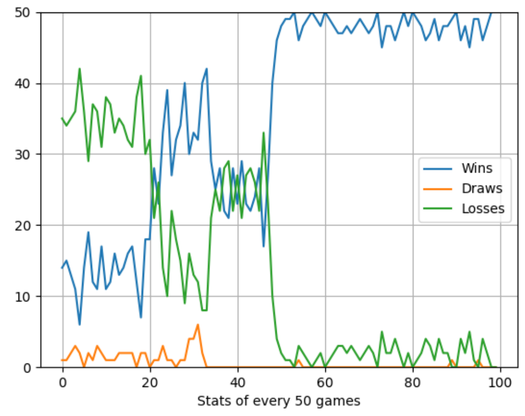


Fig. 6. vs Simple Agent

V. DISCUSSION

Finally, we review our work and discuss the qualitative outcomes of this project. We also describe some other ways which can be used to achieve similar results and are more efficient.

Traditional DQN approach trains over millions of epoch. In addition, the model is trained repeatedly in self-play mode to further improve the performance. Despite training the network for much lesser duration, we are able to obtain good level of win %. Given more number of epochs, we can outperform

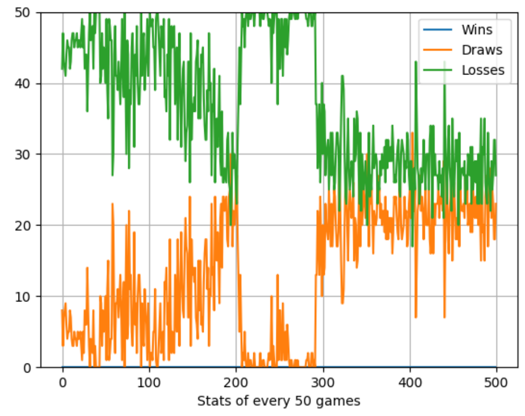


Fig. 7. vs minimax Agent

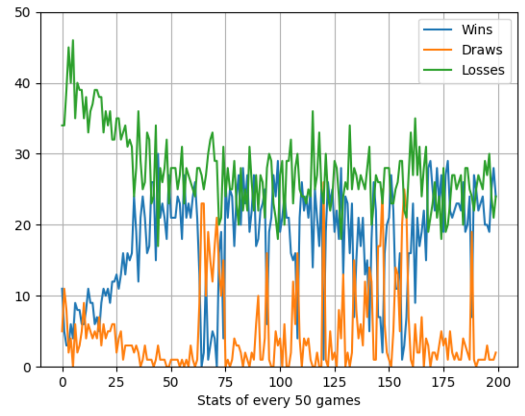


Fig. 8. vs $\alpha - \beta$ Agent

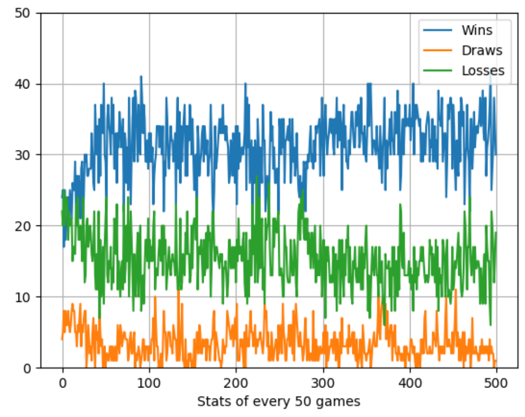


Fig. 9. Dots & Boxes : vs Random Agent

the existing approaches using our architecture and state space selection.

It is well-known that RL agents require order of magnitude more training that end-to-end supervised neural networks. This problem exacerbates when the number of states is exponentially high. In this work, we aimed to learn smaller dimension board. Given more training power and time, the same approach can be tried for a larger board-size, say for dots & boxes, a board size of 10x10 yields a state space of $O(2^{10^2})$ – even difficult for a neural network to approximate the Q values.

Finally, we observe that training RL agents are highly sensitive to the choice of hyper-parameters. This is primarily because of the sequential nature of the game. Thus, achieving the best set of hyper-parameters involves a lot of tuning.

Acknowledgment

We thank Prof. Amit Sethi and the TAs of the course EE 782 for this opportunity and assistance.

References

[1] M. Riedmiller, “Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method”. European Conference on Machine Learning, 2005.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou,D. Wierstra, and M. Riedmiller. Playing atari with deep rein-forcement learning. arXiv:1312.5602, 2013.

[3] Silver, D., Schrittwieser, J., Simonyan, K. et al. Mastering the game of Go without human knowledge. Nature 550, 354–359 (2017). https://doi.org/10.1038/nature24270.

[4] Balázs Csanád Csáji, Approximation with Artificial Neural Networks. MSc Thesis, Eötvös Loránd University (ELTE), Budapest, Hungary.

[5] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu, Asynchronous Methods for Deep Reinforcement Learning. International conference on machine learning, 2016.

[6] Hado van Hasselt and Arthur Guez and David Silver, Deep Reinforcement Learning with Double Q-learning. arXiv:1509.06461.