Playing Pong with Deep Reinforcement Learning

Project for the AG 'Theory and Implementation of Deep Learning Algorithms'

Michael Krebsbach: mkrebsbach@web.de
University of Würzburg, Supervisor: Prof. Dr. Alfio Borzi.

October 10, 2021

Abstract

Deep Q-learning has emerged in 2013 and constitutes a powerful unsupervised reinforcement learning algorithm that can learn to behave in an environment by interacting with it autonomously. A deep Q-learning algorithm is implemented and used to train two competing neural networks on the game *Pong*. Furthermore all theory and code used for the algorithm are presented, explained here and can be used to directly compete with the neural networks. This algorithm can, with little modifications, be used to interact also in other environments and with different network architectures. On the game Pong this approach leads to both neural networks playing on a human-level.

1 Introduction

The term *machine learning* (ML) was first coined in 1959 by Arthur Samuel [1]. He used it to describe his algorithm that plays the game *Checkers* and is supposed to show a behavior that would be called learning in humans, i.e. improve its performance with experience. Since then the field of machine learning has been a heavily investigated field of research with plentiful of interesting developments.
One of the approaches that nowadays are grouped under ML are *neural networks* (NNs), which date back to 1943 [2]. Originally they were developed to model the human brain by imitating neurons and interconnecting them in a similar manner as in a real brain. The network can be trained by tuning those interconnections, the so called weights. With the improvement of computer chips and emerging computational resources it became possible to increase the number of neurons that constitute the NN, leading to networks made up of multiple layers of neurons, so called *deep neural networks*.
The most important tool used for training such (deep) neural networks is the famous back propagation algorithm rediscovered by David Rumelhart in the 1980s [3]. It is used for so called *gradient descend* which is supposed to optimize the performance of the network by minimizing a *loss function*. A mayor disadvantage of neural networks is that they require a very large amount of data samples for gradient descend. For supervised learning there has to exist a label for each data point in the respective data set. The labeling is often done by a human supervisor which is a tedious process. Alternatively, many algorithms for unsupervised learning exist which do not require human supervision.
One example that is often used for unsupervised learning is *reinforcement learning* (RL). Therefore an agent that is controlled e.g. by a neural network interacts with an environment (for example a game). The agent can take some actions in the environment which changes accordingly and gives a reward back to the agent. The goal of the learning algorithm is then to maximize the reward. Also 70 years after Arthur Samuel first coined the term machine learning it is still used to play games. However, 70 years of research led to many developments and improvements compared to the original Checkers algorithm. Even though it is often used to play games like Atari games [4], Chess or Go [5], the idea can also be used in real-life scenarios like autonomous driving [6], robots [7] or optimization of chemical reactions [8].
This report describes the project *Playing Pong with Deep Reinforcement Learning* in which two *deep Q-networks* (DQNs) are implemented according to the 2013 paper *Playing Atari with Deep Reinforcement Learning* [9]. Using the algorithm to play *Pong* is merely a proof of concept, however, the same algorithm can in principle be scaled arbitrarily to interact also in more challenging environments.
The report first focuses on theory of reinforcement and deep Q-learning in Sec. 2. In a second step all important components of the implementation are discussed in Sec. 3, followed by the evaluation of some results in Sec. 4. Sec. 5 gives a short conclusion of the project. Lastly the code implemented in the project is given in appendix A. It can also be found on GitHub: `https://github.com/MKrebsbach/PlayingPongDQN`.

2 Deep Q-Learning

DQN algorithms are one sort of reinforcement learning. RL algorithms usually follow a simple scheme presented in Fig. 1 in which an agent (usually controlled by a neural network) interacts with an environment. For each discrete time step t the environment is in a state which is an element of the state space ($s_t \in S$) and is fed into the agent. Out of a set of possible actions $\mathcal{A}$ the agent chooses an action $a_t \in \mathcal{A}$ according to its policy $\pi : S \to \mathcal{A}$. This action is executed by the environment which changes into state s_{t+1} according to the rules of the enviroment and feeds a reward r_t back to A which should award good situations (e.g. winning) and penalize bad situations (e.g. loosing). This can be iterated until a final time step T is reached. The aim of the algorithm is to find the policy π that maximizes the future reward

$$R_t = \sum_{\tau=t}^{T} \gamma^{\tau-t} r_\tau \tag{1}$$

where $\gamma \in [0, 1]$ is a discount factor. This can be done using the so called *action-value function*

$$Q^*(s_t, a_t) = \max_\pi \mathbb{E}\left[R_t | s_t, a_t, \pi\right] \tag{2}$$

which satisfies the *Bellman equation*

$$Q^*(s_t, a_t) = \mathbb{E}\left[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) | s_t, a_t\right]. \tag{3}$$

Depending on the challenge, the environment might show stochastic behavior which is why the expectation value $\mathbb{E}$ has to be calculated. However, since this report focuses on the game Pong which is deterministic, it is not necessary to compute an expectation value $\mathbb{E}$ which will be ignored in the following. In words, this function returns the future return which can be obtained by, from time t on, always choosing the optimal policy which thus maximizes the future reward. This action-value function Q^* can be approximated by a neural network $Q(s, a; \theta)$ with weights θ. It takes the state s as an input and outputs the approximation of $Q^*(s, a)$ for all actions a that can possibly be taken in the next step. By choosing the action with the highest estimate of the action-value function for the current state the agent tries to use the optimal policy.
In order to train this network a couple of obstacles have to be resolved:

1. Replay memory and sampling
   As mentioned above, the training of neural networks usually requires a large data set. Also the *i.i.d.* hypothesis, that the data set contains *independent* samples which are *identically distributed* over the whole range of possible samples should be satisfied. Deep reinforcement learning is supposed to learn by playing a game without a given data set but by interacting with the environment. One game, however, consists of a sequence of highly correlated states which are neither independent nor identically distributed. This problem can be resolved by saving every game that was played so far in a *replay memory* from which samples can be drawn from independent games and thus fulfill the i.i.d. hypothesis. Eventually the size of the replay memory should be fixed to avoid exceeding memory usage.

2. Exploration vs. exploitation
   In order to make sure that the memory is indeed filled by samples that are identically distributed over all possible states of the game, the exploration vs. exploitation dilemma has to be dealt with. At one point in the learning process the agent has to learn to perform actions that result in a high positive reward. It might be possible that this restricts its movement to a small part of all possible actions and this memory is filled by games that do not represent the whole distribution of possible action-state pairs very well. However, even higher reward might be achievable using strategies not present in this memory. This can be resolved by choosing a random action with probability ϵ and the one with the highest value $Q(s, a; \theta)$ with probability $1 - \epsilon$. By decreasing ϵ with the number of games, that have been played the memory is filled with identically distributed samples first (it explores the game) before the agent exploits that for some specific actions the reward is maximized.

3. Target network
   The algorithm used for this DQN is an unsupervised learning algorithm. Thus no external supervisor is present to label the data points in the memory. However, this task can be done by the network

Figure 1: Scheme describing the main components of Reinforcement learning.

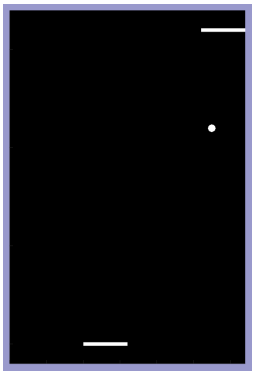

Figure 2: A screenshot from the game Pong.

itself, or rather by an old version of the network, the *target network*. By keeping an old set of weights θ_{Target}, for example from a game in which the agent performed well, an artificial supervisor is created, which should be updated only rarely, e.g. every few thousand games.

Now that these problems have been dealt with the actual learning can start. First a sample has to be randomly drawn from the memory. One sample contains the following information: s_t, a_t, r_t, s_{t+1} and, whether the game has ended during this sequence. If it has ended, then the label for this data point is simply $y = r_t = Q^*(s_t, a_t)$ which is the true value of $Q^*(s_t, a_t)$ since the game ends after this action. Otherwise it can be determined using the target network

$$y = r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_{\text{Target}}) \tag{4}$$

which is only an estimation of $Q^*(s_t, a_t)$. Note the similarity to the Bellman equation Eq. (3). Now that this data point is labeled a loss function can be calculated

$$L(\theta) = (y_i - Q(s_t, a_t; \theta))^2 \tag{5}$$

and the weights can be updated according to the stochastic gradient descend

$$\nabla_\theta L(\theta) = -(y_i - Q(s_t, a_t; \theta)) \nabla_\theta Q(s_t, a_t; \theta) \tag{6}$$

$$\theta_{\text{New}} = \theta - \mu \nabla_\theta L(\theta) \tag{7}$$

where μ is the learning rate. $\nabla_\theta Q(s_t, a_t; \theta)$ is the standard gradient for neural networks and depends on the network architecture. For a minibatch gradient descend this has to be repeated for many randomly chosen samples from the replay memory. For an average game duration of $\hat{T}$ a minibatch gradient descend steps of minibatch size $\hat{T}$ or even small multiples of $\hat{T}$ can be performed. The labels y for each sample are not saved but calculated every time a sample is drawn from the memory because the target network itself is also updated.
Even though there is no theoretical proof (yet) that, using this update scheme, $Q(s, a; \theta)$ converges to $Q^*(s, a)$, it does lead to agents that perform well in the example of the game Pong which can be seen in the following sections.

3 Implementation

This section contains a step-by-step description of the most important building blocks required when implementing a DQN for playing Pong. The full code that is executable with Matlab and Octave can be found in the appendix.

3.1 Pong Engine

The first important component of RL is an environment defining the problem that has to be solved. This report focuses on the game Pong. The engine is adjusted from [10]. A screenshot from the game can be

seen in Fig. 2. The game consists of a four sided field containing a ball that can move around freely but in a linear fashion. Once it hits one of the side walls it is deflected and the horizontal movement changes direction. Two movable blocks, one at the bottom and on at the top of the field can also deflect the ball and prevent it from touching the lower or upper side. Each block is controlled by an agent which can move it left or right or stay still. The game ends if the ball hits the lower or upper wall.
The reward the environment assigns to an action in a certain state determines the goal of the agents. For example each agent is getting a positive reward if it hits the ball with the block and a negative if it doesn't which results in loosing the game. Maximizing the reward thus leads to a game that does not end by one player loosing. By tuning the reward function it is also possible to try to assign different strategies to the agent. For example a cooperative game could be achieved by also rewarding each player positively if the opposite agent is also able to get the ball. This could result in both agents playing *easy to get* balls by for example aiming for the opposite block. On the other hand penalties, or negative rewards for every opposite deflection and a large reward for winning the game could lead to a more competitive game. However, such reward shaping might be more interesting for more challenging games than Pong.
The relevant game parameters are: ball position (x,y coordinates), ball velocity (x,y), upper block position (x) and lower block position (x). For each time step the Pong environment takes these parameters (the current state of the game) as well as the actions of each agent as input. It then performs one time step by changing the ball position according to the velocity and the blocks according to the actions. Lastly it checks whether the ball hits a wall or a block and calculates the reward. Then it returns the state of the game for the next time step as well as the information if the game has ended after this move or not.
Now that the game is set up the agents have to be defined.

3.2 Agents

Each agent is implemented as an independent neural network. In principle there is no restriction on the architecture of these neural networks. However, for a game as simple as Pong a neural network with one hidden layer suffices. It should be mentioned that in this implementation the neural networks directly take the game parameters specified above as input. Alternatively it is also possible to use screenshots of the field as high-dimensional input of the neural network. In order to be possible for the neural network to infer information about the dynamic of the game at least two consecutive screenshots would then have to be used. In this case it might be advisable to add convolutional layers to the neural networks. This would requires more memory and computational resources while at the same time using the same learning algorithm. In order to keep all computations as short as possible this is not implemented in this project. However, many examples in the literature show that such an increase in complexity can be handled by the algorithm (see e.g. [4]).
The output of each agent consists of three numbers: the estimated value of Q^* for the three possible actions *left, right* and *stay still*. The chosen action is then simply the one with maximal result. A visual representation of the competing agents in the game is presented in Fig. 3. The game field was rotated by 90° clockwise.

3.3 Deep Q-learning

The heart of the project is the DQN algorithm. It consists of four main steps:

1. Step 1: Initialization
   First the agents (named Agent1 (lower block) and Agent2 (upper block)), the memory and the learning parameters have to be initialized. In this case the agents were chosen to be neural networks with one hidden layer of 21 nodes each. ReLU was chosen as activation function after the hidden layer. This can be changed to e.g. a sigmoid function in the files *ActivationF.m* and *DerActivationF.m*. Also due to the *Exploration-vs-Exploitation trade off* the memory can immediately be filled by a number of games played with randomly chosen actions. Additionally these random games can later be used as a benchmark. If the agents are able to outperform the random games w.r.t. the average length of the games or the reward gained during the game, this is a good indication that the agents have indeed learned something.

2. Step 2: Main training loop:
   The main loop specifies how many games should be played, also referred to as *episodes*. For each episode the parameter ϵ is decreased given that it has not already reached a minimal value. Then one game is played (see Step 3: Game loop) and saved to the memory. If the game resulted in a new

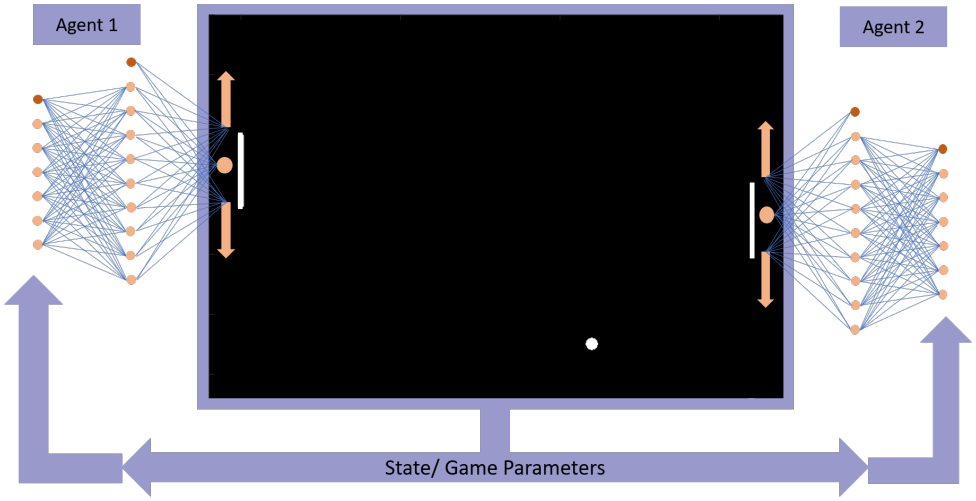

Figure 3: Representation of the two agents controlled by a neural network each which play the game Pong. For optical purposes the game field is rotated by 90° compared to the game in *Pong.m*.

high score for one of the agents, then the current network parameters of this agent can be saved to later become the next target network. Once every few thousand episodes the target can be updated using this record-breaking next target network. Of course other ways of choosing the next target network are also possible. After that the agents can be trained using minibatch gradient descend according to Eq. (7). Since by Eq. (4) a label for each sample from the memory can be calculated the gradient descend algorithm can be implemented just like in a supervised learning algorithm. In order to keep a balance between the agents it might be advisable to train only the agent that has lost the last game.

3. Step 3: Game loop
   For one game first the game parameters are initialized randomly, but in such a way that it is possible for the agents to get the ball. It is also possible to start each game with the same initial parameters. However, this would drastically decrease the diversity of states saved to the memory and samples from the memory might not be identically distributed anymore. For each agent a random action is chosen with probability ϵ. Otherwise the state is fed into the neural network of the agent which outputs the estimations for Q^* for each possible action. The action that promises a high future reward and, given that it is a good estimate of Q^*, leads to the optimal policy is then chosen. This is repeated until the game is over (one agents has lost) or a maximal time is reached.

4. Step 4: Test phase
   As a last step ϵ can be set to zero and games entirely played by the agents can be compared to the random games played in Step 1. From this comparison it should become clear whether the agents improved their game play w.r.t. random agents.

4 Evaluation

The code presented in the appendix is compatible with both *Octave* and *Matlab*. However, since Matlab is much faster it is possible to train the network for more episodes while not taking up too much time. Also the random seeds are initialized differently. Thus at the beginning of the script *RunDQN.m* the user has to specify whether Octave or Matlab is used.
By running *RunDQN.m* the DQN is set up and trained as described in Sec. 3. *TestRun.m* can be used to display 5 games played by the agents trained before. The variable *Random* should be set to *true* for random games and to *false* for agent-played games. Lastly *Pong.m* can be used to play one game against Agent2 (upper player). The lower block can be moved using the keys *a* (left) and *d* (right).
This section presents the result obtained when executing the code in Matlab and using the specified seed. In general it is not very easy to evaluate the progress of the agents, especially during the game. Unlike for supervised learning the loss function cannot be used for performance evaluation since the labels estimated with the target network are flawed. This lack of a good indication for the progress renders hyperparameter

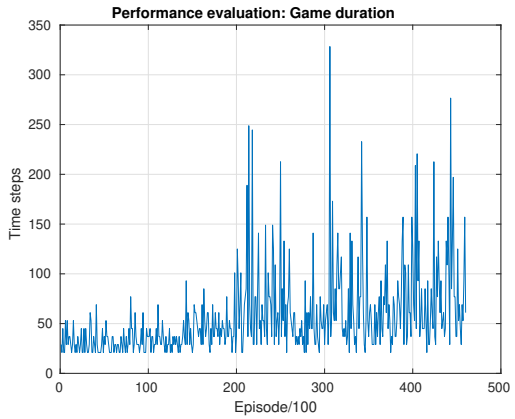

Figure 4: Performance evaluation of the agents during training. Every 100 episodes 5 games are played by the agent networks. The average game duration of these 5 games are plotted over in total 46000 episodes.

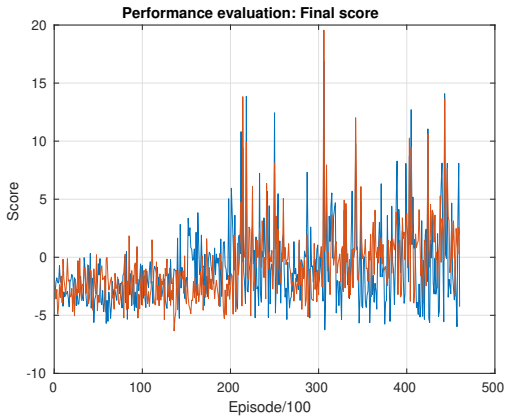

Figure 5: Performance evaluation of the agents during training. Every 100 episodes 5 games are played by the agent networks. The average game reward of Agent1 (blue) and Agent2 (red) of these 5 games are plotted over in total 46000 episodes.

tuning difficult. Two quantities that represent the progress reasonably well are the total score as well as the total runtime of the game. However, both only increase once both agents are already able the frequently get the ball. Since at first most of the games are played randomly (depending on ϵ) these games cannot be chosen when evaluating the agents. Therefore a few games with $\epsilon = 0$ have to be played every once in a while to keep track of the progress. Furthermore it should be kept track of which player is winning more often. It is not unlikely that one player is already playing fairly well while the other one always looses. Also in this case the total runtime and score do not represent the ability of the agents very well. At the same time learning can take quite a long time (especially with Octave, Matlab works faster) which renders hyperparameter tuning strategies like grid- or random-search impractical.
Due to these obstacles the main challenge in tuning the DQN is to find suitable learning parameters. They include: learning Rate μ, number of episodes, discount rate γ, minibatch size and the frequency for updating the Target networks.
The parameters specified in the code are optimized in such a way that for the given seed the agents learn to play the game relatively fast in order to be able to quickly demonstrate the capabilities of the DQN algorithm. However, this makes the learning algorithm sensitive to the initial conditions and for other random seeds other learning parameters might be better.
Even though these obstacles make learning difficult the DQN shows remarkably good results. To see this the evaluation quantities described above are plotted below. Therefore every 100th episode 5 games are played by the agents with $\epsilon = 0$ and the resulting mean time and mean reward for each player are saved. This can be seen in Figs. 4 and 5. While this does not look like a huge progress it should be kept in mind that not the current agent but only the best agent so far is important since in the end not the final agent but the best agent is chosen as a result. The later is always saved in NextTarget1 and NextTarget2. The same plots but with games played by NextTarget1 and NextTarget2 can be seen in Figs. 6 and 7. They behave step-like since all games are played with the same weights of the neural networks until a new record of Agents1 or 2 updates NextTarget1 or 2 respectively. Both plots seem to be *cut of* at time step 400 or a reward of ≈ 23. This is because the total runtime of the game is restricted to 400. The agents thus consistently reach (for some some versions of NextTarget) the best possible score.

Lastly Fig. 9 shows why it is important to keep track of which player usually wins on a different example of learning parameters. NextTarget1 (blue) has a reward which is consistently higher than that of NextTarget2 (red). This means that Agent1 has a record that is consistently able to get the ball while Agent2 looses almost always. This of course restricts the length of the game even though NextTarget1 is already reasonably well at playing. This could not have been detected by only looking at the average Time shown in Fig. 8.
After training the balanced agents are used to play 5000 games against each other with $\epsilon = 0$. In these games they reached on average rewards of about ≈ 23 and game times of ≈ 372 which is very close to the best possible result. Again the maximal game time was set to 400. Compared to random games, which on average take ≈ 35 time steps and result in a reward of ≈ -3, this is a large improvement. Thus the agents

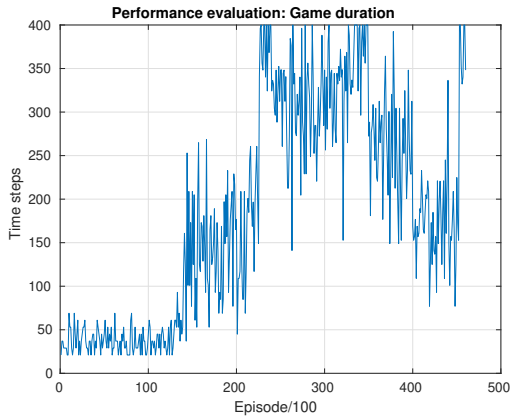

Figure 6: Performance evaluation of the next targets, i.e. the best performing agents so far during training. Every 100 episodes 5 games are played by the NextTarget networks. The average game duration of these 5 games are plotted over in total 46000 episodes.

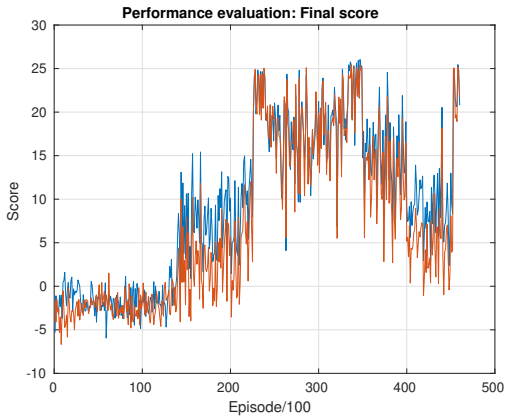

Figure 7: Performance evaluation of the next targets, i.e. the best performing agents so far during training. Every 100 episodes 5 games are played by the NextTarget networks. The average game reward of NextTarget1 (blue) and NextTarget2 (red) of these 5 games are plotted over in total 46000 episodes.

have learned to play the game very well. This achievement can be most impressively observed by directly playing against Agent2 which can be done following the instructions in the appendix. For the learning parameters specified in the code the agent wins a majority of games against a human counterpart.

5 Conclusion

In this project a DQN algorithm has been implemented which can be used to train two agents (i.e. two independent neural networks) to play the game Pong against each other. Therefore the agents play the game and save the experience into a *replay memory* which poses as a data set for training the neural networks. However, remarkably the DQN algorithm does not require a supervisor, but instead uses an old version of itself to label the data. An updating scheme derived from the Bellman equation is the used to train both agents.

With this approach agents have been trained that were able to consistently reach the maximal game time of 400 time steps, which corresponds to 5 ball deflections each. This is a significant improvement in comparison to random agents which on average deflects the ball less than once. When directly competing with a human opponent the agents were able to play on a similar, if not superior, level than the human player.

These results can be observed most impressively by directly playing against one of the players which can be done following the instructions in the appendix.

It needs to be emphasized that the DQN algorithm described in this report does not know any of the internal mechanisms of the game. With very little changes the same algorithm could be used to learn all sorts of different environment, e.g. other game or also real-life scenarios. Also the algorithm does not depend on the network architecture which could be exchanged with any other.

Finally, the DQN algorithm is an unsupervised learning algorithm that can train an agent to behave in a variety of environments (given that there exists a reward function) without any environment-specific information or any supervision or initial input. For the game Pong this project implemented a DQN algorithm that trains two neural networks to play Pong on a human level. This is achieved entirely by self-play without any supervision.

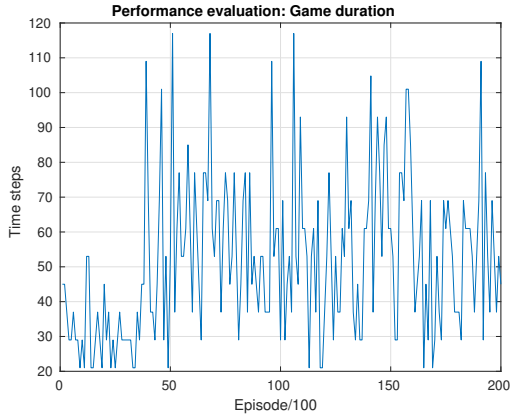

Figure 8: Performance evaluation of unbalanced agents during training. Every 100 episodes 5 games are played by the NextTarget networks. The average game duration of these 5 games are plotted over in total 20000 episodes.

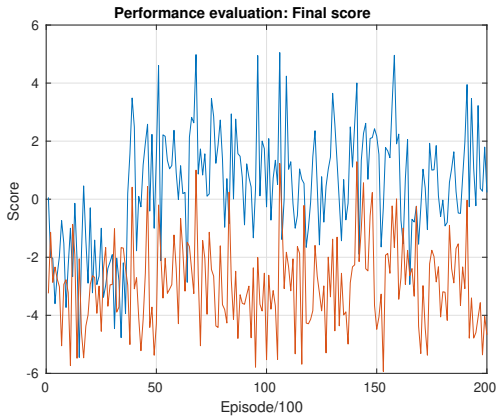

Figure 9: Performance evaluation of unbalanced agents during training. Every 100 episodes 5 games are played by the NextTarget networks. The average game reward of NextTarget1 (blue) and NextTarget2 (red) of these 5 games are plotted over in total 20000 episodes.

References

[1] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[2] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[3] D. E. Rumelhart, G. E. Hinton, and E. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[4] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[5] D. Silver et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[6] J. Chen, B. Yuan, and M. Tomizuka. Model-free deep reinforcement learning for urban autonomous driving. pages 2765–2771, 2019.

[7] T. Haarnoja et al. Learning to Walk via Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1812.11103, December 2018.

[8] Z. Zhou, X. Li, and R. N. Zare. Optimizing chemical reactions with deep reinforcement learning. *ACS Central Science*, 3(12):1337–1344, 2017. PMID: 29296675.

[9] V. Mnih et al. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1312.5602, December 2013.

[10] S. Henderson. Pong. `https://www.mathworks.com/matlabcentral/fileexchange/69833-pong/`, 2021. [Online; accessed 15-July-2021].

Appendix A Matlab/Octave Code

This Appendix lists all Matlab/Octave files required to train two agents (two-layered neural networks) to play the game Pong against each other using deep-Q learning according to this report. Each file should be saved under the filename specified in the first line of each file. Once all files are prepared the file *RunDQN.m* can be used to setup and train the agents. This can take a while depending on the learning parameters and if Matlab or Octave is used. Remember that at the beginning of *RunDQN* it has to be specified whether Octave or Matlab is used. Also the agents are only saved to the *workspace* (Matlab) or the *variable editor* (Octave) and not to a file. Thus when starting the program *RunDQN* or after clearing the *workspace/variable editor RunDQN* always has to be executed first. By executing *TestRun.m* 5 games played by the agents that have just been trained can be watched. By setting *Random* to true alternative random agents can be watched as a benchmark for the performance. Lastly *Pong.m* can be used to play the game against Agent2.
The reward function defined in the environment *PongNextStep.m* is constructed in a way that is described as *cooperative* above.
The different files are presented next in an alphabetical order.

```
% ActivationF.m
function x = ActivationF(x)

  % ReLU
  x = max(0, x);

  % Sigmoid
  % x = 1 ./ (1 + exp(-x))
end
```

```
% DerActivationF.m
function x = DerActivationF(x)

  % ReLU
  x = (x > 0);

  % Sigmoid
  % x = x .* (1-x);
end
```

```
% DQN.m
function out = DQN(Agent, PongVariables, IntermediateOutput)

  if nargin < 3
      IntermediateOutput = false;
  end

  % Normalize Input
  PongVariables = PongVariables ./ [32; 64; 1.5; 1.5; 32; 32];

  % Add Bias
  PongVariables(end + 1) = 1;

  % Layer 1
  v1 = Agent.W1 * PongVariables;
  y1 = ActivationF(v1);
  y1(end + 1) = 1;              % Add Bias
```

```
  % Layer 2
  v2 = Agent.W2 * y1;

  if IntermediateOutput
    out = struct();
    out.y1 = y1;
    out.v2 = v2;
  else
    out = v2;
  end

end
```

```
% EpisodeToFigure.m
function EpisodeToFigure(Episode)

  % ------------------------------ Setup
      ------------------------------------------------------ %
  global fieldHeight
  global fieldWidth

  fieldHeight = 64;
  fieldWidth = 32;

  PongVariables = Episode(1).PongVariables;

  % figure
  pongFigure = figure('color', [.6 .6 .8],...
  'units','normalized','position',[.1 .1 .3 .8]);

  % axes
  pongAxes   = axes('color', 'black', 'XLim', [0, fieldWidth],...
       'YLim',[-4,fieldHeight+4], 'position', [.05 .05 .9 .9]);
  xticklabels([]);
  yticklabels([]);

  % ball
  pongBall     = line(PongVariables(1), PongVariables(2) ,...
   'marker','.', 'markersize',25, 'color','white');

  % lower block
  pongBlock1 = line([PongVariables(5) - 3, PongVariables(5) + 3], [0 0],...
   'linewidth',4,'color','white');

  % upper block
  pongBlock2 = line([PongVariables(6) - 3, PongVariables(6) + 3],...
                    [fieldHeight fieldHeight], 'linewidth',4,'color','white
                        ');


  for eps = 2:length(Episode)
    PongVariables = Episode(eps).PongVariables;

    set(pongBall, 'XData', PongVariables(1), 'YData', PongVariables(2))

    set(pongBlock1, 'XData', [PongVariables(5) - 3, ...        % update lower
        block
                              PongVariables(5) + 3])
```

```
    set(pongBlock2, 'XData', [PongVariables(6) - 3, ...        % update upper
        block
                              PongVariables(6) + 3])

    pause(.03);

  end
  pause(0.1)
  close
end
```

```
% Gradient.m
function [dW1, dW2] = Gradient(Agent, Target, ...
                               x1, action, reward, x2, ...
                               game_end, Discount)

  % ---------- Calculate y from Bellman Equation ---------- %

  if game_end
    y       = reward;
  else
    out     = DQN(Target, x2, false);
    QTarget = max(out);
    y       = reward + Discount * QTarget;
  end

  % Agent's estimate of Q (and intermediate steps out.y1, out,v2)
  out         = DQN(Agent, x1, true);


  % ---------- Backpropagation ---------- %

  e2          = zeros([3,1]);
  e2(action)  = y - out.v2(action);
  delta2      = e2;

  e1          = Agent.W2(:,1:end-1)' * delta2;
  delta1      = DerActivationF(out.y1(1:end-1)) .* e1;

  % Normalize Input like in DQN
  x1          = x1 ./ [32; 64; 1.5; 1.5; 32; 32];

  % Add Bias like in DQN.
  x1(end +1) = 1;

  % Gradients
  dW1         = delta1 * x1';
  dW2         = delta2 * out.y1';


end
```

```
% InitializeAgent.m
function Agent = InitializeAgent(L1)

  if nargin < 1
```

```
    L1 = 21;
  end

  Agent    = struct();
  Agent.W1 = rand([L1 7])*2 - 1;
  Agent.W2 = rand([3 L1+1])*2 - 1;

end
```

```
% InitializePong.m
function PongVariables = InitializePong()

  PongVariables = [randi([5 28]);      ... % PongBallPos X
                   32;                 ... % PongBallPos Y
                   rand([1,])*3-1.5;   ... % PongBallVel X
                   randi(2)*3-4.5;     ... % PongBallVel Y
                   randi([5 28]);      ... % PongBlockCenter 1 (Lower)
                   randi([5 28])       ... % PongBlockCenter 2 (Upper)
                   ];

end
```

```
% InitializeTarget.m
function Target = InitializeTarget(L1)

  if nargin < 1
      L1 = 21;
  end

  Target    = struct();
  Target.W1 = zeros([L1 7]);
  Target.W2 = zeros([3 L1+1]);

endfunction
```

```
% keyboardFunction.m
function action1 = keyboardFunction(figure, event)
  global PongBlockCenter1
  global fieldWidth

  switch event.Key
    case 'a'
      if PongBlockCenter1 - 5 > 0
        PongBlockCenter1 = PongBlockCenter1 - 2;
      end
    case 'd'
      if fieldWidth - PongBlockCenter1 - 5 > 0
        PongBlockCenter1 = PongBlockCenter1 + 2;
      end
  end
end
```

```
% MinibatchSGD.m
function Agent = MinibatchSGD(Agent, AgentName, Target, LearningRate,
    Discount, MinibatchSize, Regularization)
```

```
  if nargin < 6
    MinibatchSize = 64;
  end

  if nargin < 7
    Regularization = 0;
  end

  dW1 = zeros(size(Agent.W1));
  dW2 = zeros(size(Agent.W2));


  % ------------------------ Collect MinibatchSize Gradients
      ------------------------ %

  for i = 1:MinibatchSize

    % Sample Random instance from Memory
    [x1, action1, action2, reward1, reward2, x2, game_end] =
        SampleFromMemory();

    % Train only one of the agents (specified by Agent Name)
    if AgentName == 1
      action = action1;
      reward = reward1;
    elseif AgentName == 2
      action = action2;
      reward = reward2;
    end

    % Calculate Gradient
    [dW1temp, dW2temp] = Gradient(Agent, Target, ...
                                  x1, action, reward, x2, ...
                                  game_end, Discount);

    % Add Gradient
    dW1 = dW1 + dW1temp;
    dW2 = dW2 + dW2temp;

  end


  % ------------------------ Update Agent
      ------------------------------------------ %

  Agent.W1 = LearningRate / MinibatchSize * dW1 + (1- Regularization) *
      Agent.W1;
  Agent.W2 = LearningRate / MinibatchSize * dW2 + (1- Regularization) *
      Agent.W2;

end
```

```
% Pong.m
% This Program lets you play the game "Pong" against a Neural Network
% The code for this environment was copied from Silas Henderson:
% https://de.mathworks.com/matlabcentral/fileexchange/69833-pong/
% and adjusted for these purposes
```

```
clc; close all;

if exist("Agent2", "var")
  disp ('Use existing Agent')
else
  Agent2 = load('Agent2_10x10_Competitive_2L_260.txt').Agent2
end

global fieldHeight;
global fieldWidth;

fieldHeight = 64;
fieldWidth  = 32;

global PongBlockCenter1;
TestReward1 = 0;
TestReward2 = 0;


% ------------------------------ Setup
    ------------------------------------------------------ %

% Setup Variables
PongVariables = InitializePong();
PongVariables(4) = 1.5; % Initial Movement should go up
PongBallPos = [PongVariables(1) PongVariables(2)];
PongBallVel = [PongVariables(3) PongVariables(4)];
PongBlockCenter1 = PongVariables(5);
PongBlockCenter2 = PongVariables(6);

% Figure
PongFigure = figure('color', [.6 .6 .8],...
'KeyPressFcn', @keyboardFunction,...
'units','normalized','position',[.1 .1 .3 .8]);

% Axes
pongAxes    = axes('color', 'black',...
 'XLim', [0 fieldWidth], 'YLim',[-4,fieldHeight+4], 'position', [.05 .05 .9
     .9]);
xticklabels([]);
yticklabels([]);

% Ball
PongBall    = line(PongBallPos(1),PongBallPos(2),...
 'marker','.', 'markersize',25, 'color','white');

% Lower Block
PongBlock1 = line([PongBlockCenter1 - 3, PongBlockCenter1 + 3], [0 0],...
 'linewidth',4,'color','white');

% Upper block
PongBlock2 = line([PongBlockCenter2 - 3, PongBlockCenter2 + 3], ...
                  [fieldHeight fieldHeight], 'linewidth',4,'color','white')
                      ;
```

```
% ------------------------ Game Loop
    ------------------------------------------------------ %

t = 1;
game_end = false;

while not (game_end)

  % Agent 2
  out = DQN(Agent2, PongVariables, false);
  [Q2, action2] = max(out);

  PongVariables(5) = PongBlockCenter1;


  % ---------------------- Perform one step in Game
     ------------------------------- %

  [PongVariables, game_end, reward1, reward2] ...
              = PongNextStep(PongVariables, 1, action2);

  PongBallPos = [PongVariables(1) PongVariables(2)];
  PongBallVel = [PongVariables(3) PongVariables(4)];
  PongBlockCenter2 = PongVariables(6);

  PongVariables(5) = PongBlockCenter1;

  % Keep Track of Total Reward
  TestReward1 = TestReward1 + reward1;
  TestReward2 = TestReward2 + reward2;

  set(PongBall, 'XData', PongBallPos(1), 'YData', PongBallPos(2)) % Update
      Ball

  set(PongBlock1, 'XData', [PongBlockCenter1 - 3, ...      % Update lower
      block
                            PongBlockCenter1 + 3])

  set(PongBlock2, 'XData', [PongBlockCenter2 - 3, ...      % Update upper
      block
                            PongBlockCenter2 + 3])


  pause(.043);

  % raise running index
  t = t + 1;

end

close;
```

```
% PongNextStep.m
function [PongVariables, game_end, reward1, reward2] = ...
                PongNextStep(PongVariables, action1, action2)

  global fieldHeight
  global fieldWidth
```

```
game_end = false;
reward1 = 0;
reward2 = 0;

% Extract Pong Parameters
pongBallPos = [PongVariables(1) PongVariables(2)];
pongBallVel = [PongVariables(3) PongVariables(4)];
pongBlockCenter1 = PongVariables(5);
pongBlockCenter2 = PongVariables(6);

% Perform action1 and action2
% 2 = Left, 3 = Right, else = Pause
switch action1
  case 2
    if pongBlockCenter1 - 1 > 3 + 1
      pongBlockCenter1 = pongBlockCenter1 - 2;
    end
  case 3
    if fieldWidth - pongBlockCenter1 > 3 + 1
      pongBlockCenter1 = pongBlockCenter1 + 2;
    end
end

 switch action2
  case 2
    if pongBlockCenter2 - 1 > 3 + 1
      pongBlockCenter2 = pongBlockCenter2 - 2;
    end
  case 3
    if fieldWidth - pongBlockCenter2 > 3 + 1
      pongBlockCenter2 = pongBlockCenter2 + 2;
    end
end

% Move Pong Ball
pongBallPos = pongBallPos + pongBallVel;

% Side Bounce Check Left
if pongBallPos(1) - 1 <= 3 && pongBallVel(1) < 0
  pongBallVel(1) = - pongBallVel(1);
end

% Side Bounce Check Right
if fieldWidth - pongBallPos(1) <= 3 && pongBallVel(1) > 0
  pongBallVel(1) = - pongBallVel(1);
end

% Lower Block Check
if pongBallPos(2) - 1 < 2.5 && pongBallVel(2) < 0
  if abs(pongBallPos(1) - pongBlockCenter1) <= 3.5
    pongBallVel(2) = - pongBallVel(2);
    pongBallVel(1) = pongBallVel(1) + (pongBallPos(1) - pongBlockCenter1)
        /2;
    pongBallVel(1) = sign(pongBallVel(1)) * min(abs(pongBallVel(1)), 1.5)
        ;
    reward1 = reward1 + 3 + 1.5 - abs(pongBallPos(1) - pongBlockCenter1)
        /2;
```

```
      reward2 = reward2 + 1;
    else
      game_end = true;
      reward1 = reward1 - (3 + abs(pongBallPos(1) - pongBlockCenter1)/5);
      reward2 = reward2 - 1;
    end
  end

  % Upper Block Check
  if fieldHeight - pongBallPos(2) < 2.5 && pongBallVel(2) > 0
    if abs(pongBallPos(1) - pongBlockCenter2) <= 3.5
      pongBallVel(2) = - pongBallVel(2);
      pongBallVel(1) = pongBallVel(1) + (pongBallPos(1) - pongBlockCenter2)
          /2;
      pongBallVel(1) = sign(pongBallVel(1)) * min(abs(pongBallVel(1)), 1.5)
          ;
      reward1 = reward1 + 1;
      reward2 = reward2 + 3 + 1.5 - abs(pongBallPos(1) - pongBlockCenter2)
          /2;
    else
      game_end = true;
      reward1 = reward1 - 1;
      reward2 = reward2 - (3 + abs(pongBallPos(1) - pongBlockCenter2)/5);
    end
  end

  % Update PongVariables
  PongVariables = [pongBallPos(1); pongBallPos(2); pongBallVel(1);
      pongBallVel(2);pongBlockCenter1; pongBlockCenter2];

end
```

```
% RandomAgent.m
function [idx, t, R1, R2] = RandomAgent(idx, MaxMemory, MaxRuntime)

  global Memory

  % Initialize Pong
  Memory(idx).PongVariables = InitializePong();
  Memory(idx).game_end = false;

  t = 1;
  while t < MaxRuntime && not (Memory(idx).game_end)

    % Choose Random Action
    Memory(idx).action1 = randi(3);
    Memory(idx).action2 = randi(3);

    [Memory(mod(idx, MaxMemory) + 1).PongVariables, ...
     Memory(mod(idx, MaxMemory) + 1).game_end, ...
     Memory(idx).reward1, ...
     Memory(idx).reward2] = PongNextStep(Memory(idx).PongVariables, ...
                                         Memory(idx).action1, ...
                                         Memory(idx).action2);

    % raise running indices
    t = t + 1;
    idx = mod(idx, MaxMemory) + 1;
```

```
  end
  R1 = Memory(idx-1).reward1;
  R2 = Memory(idx-1).reward2;
  idx = mod(idx, MaxMemory) + 1;

end
```

```
% RunDQN.m
clear; clc; close all;
tic;

% First specify if you use Octave (true) or Matlab (false)
Octave = false;

% Set State for Random Generator
if Octave
  rand('state', 895647); % 895647
else
  rng(895647);
end

global fieldHeight
global fieldWidth
global Memory

% Fix size of the Pong Field
fieldHeight = 64;
fieldWidth  = 32;

% ------------------------ Step 1: Initializazion ------------------------ %
% ------------------------ 1. Setup Learning Parameters ------------------ %

if Octave
  StartLearning  = 1600 ;      % # of Random Episodes before Learning
      starts
  NEpisodes      = 20000;      % Octave is much slower but this works as
      well (at least on Windows?)
  LearningRate   = 6e-1 ;
  MaxMemory      = 1e4  ;      % Size of the Memory (Again smaller for
      Octave)
  UpdateTarget   = 4001 ;      % Periode with which Target is updated
else
  StartLearning  = 5000 ;     % # of Random Episodes before Learning starts
  NEpisodes      = 46000;
  LearningRate   = 4e-1 ;
  MaxMemory      = 1e5  ;     % Size of the Memory
  UpdateTarget   = 10001;     % Periode with which Target is updated
end

Discount       = 0.95 ;      % Discount in Bellman Equation
MinibatchSize  = 64   ;
MaxRuntime     = 400  ;      % Maximal Runtime of Pong
Regularization = 0e-5 ;


% Setup Epsilon Greedy
Epsilon        = 1;
```

```
EpsilonDel      = 1e-4;
EpsilonMin      = 0.05;

% Setup Memory
Memory          = struct();
idx             = 1;          % Memory index


% ------------------------ 2. Initialize Agents and Targets ------------------ %

NHidden = 21;

Agent1  = InitializeAgent(NHidden);
Agent2  = InitializeAgent(NHidden);

Target1 = Agent1; %InitializeTarget(NHidden);
Target2 = Agent2; %InitializeTarget(NHidden);


% ------------------------ 3. Fill Memory with Random Games ------------------ %

disp('
    ********************************************************************')
disp('********************      Random Phase
    ***************************')
disp('
    ********************************************************************')

TRandom  = [];
R1Random = [];
R2Random = [];
for RandomEpisode = 1:StartLearning

  % RandomAgent plays one round of pong
  % All Actions are chosen randomly
  % Everything is saved into Memory

  [idx, t, R1, R2] = RandomAgent(idx, MaxMemory, MaxRuntime);
  TRandom(RandomEpisode)  = t;
  R1Random(RandomEpisode) = R1;
  R2Random(RandomEpisode) = R2;

  if mod(RandomEpisode, 100) == 0
    disp('################################')
    RandomEpisode
    meanTime = mean(TRandom(end-99:end))
  end

end


% ------------------------ Step 2: Training ---------------------------------- %

disp('
    ********************************************************************')
disp('********************      Training Phase
    ***************************')
disp('
```

```
    **********************************************************************')

% Some Variables and Lists for later
Looser      = 2;
Winner      = [];
Time        = [];
WinnerStat  = [];
MaxReward1  = -10;
MaxReward2  = -10;
DevTime     = [];
DevReward1  = [];
DevReward2  = [];

% ---------------------- 1. Main Training Loop ---------------------- %

for Episode = 1:NEpisodes

  % Setup Epsilon
  Epsilon = max( EpsilonMin, Epsilon - EpsilonDel);

  % Initialize Pong Parameters
  Memory(idx).PongVariables = InitializePong();
  Memory(idx).game_end = false;

  % Game Loop;
  t = 1;
  TotalReward1 = 0;
  TotalReward2 = 0;


  % ---------------------- Step 3: Game Loop ---------------------- %

  while t < MaxRuntime && not (Memory(idx).game_end)

    % Agent 1
    % With Probability Epislon choose Random Action
    % Otherwise Agent1 chooses Action
    if rand([1,]) < Epsilon
      Memory(idx).action1 = randi(3);
    else
      out = DQN(Agent1, Memory(idx).PongVariables, false);
      [Q1, Memory(idx).action1] = max(out);
      Memory(idx).Q1 = Q1;
    end

    % Agent 2
    % With Probability Epislon choose Random Action
    % Otherwise Agent2 chooses Action
    if rand([1,]) < Epsilon
      Memory(idx).action2 = randi(3);
    else
      out = DQN(Agent2, Memory(idx).PongVariables, false);
      [Q2, Memory(idx).action2] = max(out);
      Memory(idx).Q2 = Q2;
    end
```

```
  % ---------------------- Perform one step in Game ---------------------- %
  % Save new game parameters and rewards in Memory
  [Memory(mod(idx, MaxMemory) + 1).PongVariables, ...
   Memory(mod(idx, MaxMemory) + 1).game_end, ...
   Memory(idx).reward1, ...
   Memory(idx).reward2] = PongNextStep(Memory(idx).PongVariables, ...
                                       Memory(idx).action1, ...
                                       Memory(idx).action2);

  % Keep Track of Total Reward
  TotalReward1 = TotalReward1 + Memory(idx).reward1;
  TotalReward2 = TotalReward2 + Memory(idx).reward2;

  % The looser of this round is trained next round
  if Memory(mod(idx, MaxMemory) + 1).game_end
    if Memory(idx).reward1 < Memory(idx).reward2
      Looser = 1;
      Winner(end+1) = 2;
      Time(end+1) = t+1;
    else
      Looser = 2;
      Winner(end+1) = 1;
      Time(end+1) = t+1;
    end
    WinnerStat(end+1) = Winner(end);
  end

  % raise running indices
  t = t + 1;
  idx = mod(idx, MaxMemory) + 1;

end

idx = mod(idx, MaxMemory) + 1;

% If new Reward record save this Weights for next Target update
if TotalReward1 >= MaxReward1
  MaxReward1 = TotalReward1;
  NextTarget1 = Agent1;
end
if TotalReward2 >= MaxReward2
  MaxReward2 = TotalReward2;
  NextTarget2 = Agent2;
end

if mod(Episode, UpdateTarget) == 0
  disp('**************************************************')
  disp('***************** UPDATE TARGETS *****************')
  disp('**************************************************')
  Target1 = NextTarget1;
  Target2 = NextTarget2;
end

% ------------------------ Minibatch Training ------------------------ %
if Looser == 1
  Agent1 = MinibatchSGD(Agent1, 1, Target1, LearningRate, Discount, ...
                        MinibatchSize, Regularization);
else
```

```
    Agent2 = MinibatchSGD(Agent2, 2, Target2, LearningRate, Discount, ...
                          MinibatchSize, Regularization);
  end


  if mod(Episode, 100) == 0
    TestTime = [];
    TestReward1 = [];
    TestReward2 = [];
    for i = 1:5
      [TestTime(i), TestReward1(i), TestReward2(i)] = RunOneGame(
          NextTarget1, NextTarget2, MaxRuntime);
    end
    disp ('##################################################')
    meanTestTime    = mean(TestTime)
    meanDevReward1 = mean(TestReward1)
    meanDevReward2 = mean(TestReward2)
    Episode
    DevTime(end +1)     = meanTestTime;
    DevReward1(end +1) = meanDevReward1;
    DevReward2(end +1) = meanDevReward2;
    Epsilon
    meanWinner = mean(WinnerStat)
    WinnerStat = [];
  end

end


% ---------------------- Step 4: Test Phase ---------------------- %

disp ('
    ********************************************************************')
disp ('********************      Test Phase
    ******************************')
disp ('
    ********************************************************************')

TTest = [];
R1Test = [];
R2Test = [];

% Target is often better than Agent
Agent1 = NextTarget1;
Agent2 = NextTarget2;

for i = 1:StartLearning

  % Initialize Pong parameters
  PongVariables = InitializePong();
  game_end = false;

  t = 1;
  TotalReward1 = 0;
  TotalReward2 = 0;
  % ---------------------- One Game Loop
      ---------------------------------------- %
  while t<MaxRuntime && not (game_end)
```

```
    [Q1, action1] = max(DQN(Agent1, PongVariables));
    [Q2, action2] = max(DQN(Agent2, PongVariables));

    [PongVariables, game_end, reward1, reward2] ...
                      = PongNextStep(PongVariables, action1, action2);

    TotalReward1 = TotalReward1 + reward1;
    TotalReward2 = TotalReward2 + reward2;
    % Raise running Indices
    t = t + 1;

  end
  R1Test(i) = TotalReward1;
  R2Test(i) = TotalReward2;
  TTest(i) = t;
  if mod(i, 100) == 0
    disp ('############ Test Episode'), disp (i)
    meanTest = mean(TTest(end-99:end))
  end
end

% Compare mean Game Time of Random Loop with Test Loop
disp ('#######################################################')
disp ('Random vs DQN Agents:')
meanRandom   = mean(TRandom)
meanTest     = mean(TTest)
meanR1Random = mean(R1Random)
meanR2Random = mean(R2Random)
meanR1Test   = mean(R1Test)
meanR2Test   = mean(R2Test)

figure(1)
plot(DevTime)
xlabel('Episode/100')
ylabel('Time steps')
title('Performance evaluation: Game duration')
axis;
grid on

figure(2)
plot(DevReward1)
hold on
plot(DevReward2)
xlabel('Episode/100')
ylabel('Score')
title('Performance evaluation: Final score')
axis;
grid on

% Print total time used for training
toc;
```

```
% RunOneGame.m
function [t, TestReward1, TestReward2] = RunOneGame(Agent1, Agent2,
    MaxRuntime)
  global fieldHeight
  global fieldWidth
```

```
% Initialize Pong Parameters
PongVariables = InitializePong();
game_end = false;

% Game Loop;
t = 1;
TestReward1 = 0;
TestReward2 = 0;


% ---------------------- Game Loop
    ------------------------------------------------------ %

while t < MaxRuntime && not (game_end)

  % Agent 1
  out = DQN(Agent1, PongVariables, false);
  [Q1, action1] = max(out);

  % Agent 2
  out = DQN(Agent2, PongVariables, false);
  [Q2, action2] = max(out);


  % ---------------------- Perform one step in Game
      ------------------------------- %

  [PongVariables, game_end, reward1, reward2] ...
              = PongNextStep(PongVariables, action1, action2);

  % Keep Track of Total Reward
  TestReward1 = TestReward1 + reward1;
  TestReward2 = TestReward2 + reward2;

  % Save Winner
  if game_end
    if reward1 < reward2
      Winner = 2;
    else
      Winner = 1;
    end
  end

  % raise running index
  t = t + 1;

end
```

```
% SampleFromMemory.m
function [x1, action1, action2, reward1, reward2, x2, game_end] =
    SampleFromMemory()

  global Memory

  game_end_old = true;

  % Choose index that does not correspond to a last state
  while game_end_old
```

```
    M = length(Memory) - 1;

    id = randi(M);

    game_end_old = Memory(id).game_end;

  end

  % Extract relevant Parameters from Memory(index)
  x1       = Memory(id).PongVariables;
  action1  = Memory(id).action1;
  action2  = Memory(id).action2;
  reward1  = Memory(id).reward1;
  reward2  = Memory(id).reward2;
  x2       = Memory(id + 1).PongVariables;
  game_end = Memory(id + 1).game_end;


end
```

```
% TestRun.m
% If there is currently no variable Agent1/2 then load it from a file
if exist("Agent1", "var")
  disp ('Use Agent1 from Workspace/Variable editor')
else
  disp ('No Agent1 was found in the Workspace/Variable editor');
end

if exist("Agent2", "var")
  disp ('Use Agent2 from Workspace/Variable editor')
else
  disp ('No Agent2 was found in the Workspace/variable editor');
end

% Switch between Random Game and NN-Game
Random = false;

% Initialize lists to save performance
TTest = [];
Winner = [];

global fieldHeight
global fieldWidth

fieldHeight = 64;
fieldWidth = 32;


% ------------------------------ Main Loop
  ------------------------------ %

for i = 1:5
  TestMemory = struct();
  PongVariables = InitializePong();
  TestMemory(i).PongVariables = PongVariables;
  t = 1;
  game_end = false;
```

```
  % ---------------------------------- Game Loop
    ---------------------------------- %

  while t < 400 && not (game_end)
    [Q1, action1] = max(DQN(Agent1, PongVariables));
    [Q2, action2] = max(DQN(Agent2, PongVariables));

    if Random
      action1 = randi(3);
      action2 = randi(3);
    end

    [PongVariables, game_end, reward1, reward2] = PongNextStep(
        PongVariables, action1, action2);
    TestMemory(t).PongVariables = PongVariables;

    t = t + 1;
  end


  EpisodeToFigure(TestMemory)

  if t ~= 400
    if reward1 > reward2
      Winner(end+1) = 1;
    else
      Winner(end+1) = 2;
    end
  end

  TTest(i) = t;
  if mod(i, 100) == 0
    disp ('########## Test Episode'), disp (i)
    t
  end
end

TTest
Winner
```