Learning to Play Breakout Using Deep Q-Learning Networks

Gabriel Andersson and Martti Yap

***Abstract*—We cover in this report the implementation of a reinforcement learning (RL) algorithm capable of learning how to play the game *Breakout* on the Atari Learning Environment (ALE). The non-human player (agent) is given no prior information of the game and must learn from the same sensory input that a human would typically receive when playing the game.**

The aim is to reproduce previous results by optimizing the agent driven control of *Breakout* so as to surpass a typical human score. To this end, the problem is formalized by modeling it as a Markov Decision Process. We apply the celebrated Deep Q-Learning algorithm with action masking to achieve an optimal strategy.

We find our agent's average score to be just below the human benchmarks: achieving an average score of 20, approximately 65% of the human counterpart. We discuss a number of implementations that boosted agent performance, as well as further techniques that could lead to improvements in the future.

***Index Terms*—Breakout, Deep Q-learning, Markov decision process, Reinforcement learning.**

I. Introduction

The topic of artificial intelligence (AI) has become increasingly popular in the recent years and is now becoming standard technology in many fields [1]. This includes Machine Learning (ML), a subset of AI, where computer algorithms learn to perform tasks based on patterns and inference rather than explicitly programmed instructions. Common applications of this include image classifiers, where ML algorithms learn to characterise features in an image. This typically falls under the ML category of *supervised learning* — where the algorithm learns by training on data pre-labelled with its inherent features.

Reinforcement learning (RL) is another machine learning technique in which the algorithm or *agent* makes decisions in the context of an *environment* — wherein it aims to maximize a given optimality criterion. The decision or *action* taken by the agent can at every time step influence the environment. As a result, the agent has the means to influence which sequences of data are available to it, setting it apart from, for instance, *supervised learning*. A further distinction is that the data available to an RL agent is unlabelled, and received sequentially as the environment is explored.

Video games have gained popularity as applicable domains for RL implementations. Aside from being relatively well known to a wider audience, games also provide an inexpensive and risk-free environment for agents. The popularity can also be attributed to the potential they offer in terms of repeatability and concurrency, speeding up training. Lastly, you may tailor your choice of game; allowing you, for instance, to test the generality of your algorithm on similar games. One such example of the latter is [2], in which the same agent was used to learn a wide range of Atari 2600 games on the Arcade Learning Environment (ALE) [3].

A common challenge faced by RL algorithms is associating actions with long-term and short-term rewards. The difficulty lies in that the rewards and the actions that caused them can be separated by any number of time-steps. In order to learn, the agent is left with the challenge of associating actions with their corresponding rewards.

Further difficulties are met in practice when the environments are very large. Reinforcement learning algorithms like *Q-learning* [4] rely on fully exploring an environment to achieve an optimal behaviour. As a result, RL methods were difficult to successfully apply to video games prior to the addition of Deep Learning elements. Progress was made in part due to the application of function approximators in the form of neural networks (DQN). These approximations did away with the need to fully explore an environment; making larger environments like video games a feasible application for RL.

Among the first of these applications was *Deep Q-Learning* which was introduced by [5]. This new implementation does, however, have its drawbacks. We, for instance, forgo all the theoretical guarantees of convergence that algorithms like *Q-Learning* offered. There arise also issues in the repeatability of the training process. Since the results are simulation based, they may differ from one run to another.

It is therefore of interest to see if these results can be reproduced independently. With this in mind, we aim to implement a reinforcement learning agent capable of learning how to play Breakout on ALE, as done in [5]. To measure the success of the agent we will compare the average score to that of an experienced human player, as recorded in [5].

II. Background

The act of playing Breakout is a decision making problem that we formalize as a discrete-time $t \in \mathbb{N}$, finite state Markov Decision Process (MDP).

A. Definitions

We define an MDP as a tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$ where

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- P is a transition probability function, such that $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$, where $\Delta(\mathcal{S})$ is a distribution over the states.

- r is a reward function such that $r : \mathcal{S} \times \mathcal{A} \to [0, K]$ where $K > 0$ is a bounded constant.
- γ is a discount factor, $\gamma \in (0, 1)$.

We can see that all transition probabilities $P(s_{t+1}|s_t, a_t)$, fulfill the *Markov Property*. That is, the probability of the next state only depends on the current state.

We define a policy π as a sequence of distributions over actions given the amount of information available at time t. In particular, we will restrict our attention to stationary markov policies $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. We do so because it is known that *optimal policies* π_* for our class of MDPs belong to stationary Markov policies (Theorem 6.2.10 in [6]).

For a policy π we define the *state-value function* $v_\pi(s)$ as the expected cumulative discounted reward or expected *return*. That is,

$$v_\pi(s) = \mathbb{E}_s^\pi \left[\sum_{t=1}^{\infty} \gamma^t r(s_t, a_t) \right], \tag{1}$$

where we make use of the discount factor γ and introduce the notation $\mathbb{E}_s^\pi$ to be the expected value given s, following policy π. The goal of our agent is then to find the policy that satisfies the *optimal state-value function* $v_*(s) = \max_\pi v_\pi(s)$. Which can also be expressed;

$$v_*(s) = \max_a \left[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) v_*(s') \right]. \tag{2}$$

We can then in a similar fashion define the *action-value function* q_π, which describes the expected return of taking action a in state s and thereafter following policy π:

$$q_\pi(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) v_\pi(s'). \tag{3}$$

From here we introduce the *optimal action-value function* q_* as the optimal value function upon taking action a, that is

$$q_*(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) v_*(s'). \tag{4}$$

Solving the MDP is now equivalent to finding $q_*(s, a)$, since one can then adopt a policy that selects the action that yields the highest return at every time step.

B. Q-Learning

One approach of finding q_* (and thereby the optimal policy π_*), is called *Q-learning*. This is a model-free algorithm, meaning that we do not aim to estimate the transition probability function nor the rewards (P, r) of the MDP. The *Q-learning* algorithm is given in Algorithm 1.

The algorithm starts by initializing q arbitrarily (line 1.) and then applying a policy that selects an action at each given time step t. Adopting a policy that always selects $a_t = \text{argmax}_a (q(s_t, a))$, may initially seem advantageous. However, this greatly limits the explored state-action space and leads to over-reliance on the initial states. A common solution to this exploration problem is to use the ϵ-*greedy* policy given in the Q-learning algorithm (line 6.–11.). The convergence of

Algorithm 1: Q-learning algorithm

1: Define a convergence tolerance δ
2: Initialize $q(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
3: Define ϵ-*greedy* constant $\epsilon \in (0, 1)$
4: Receive initial state s_0
5: **while** $||q_t(s, a) - q_{t+1}(s, a)|| > \delta$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$ **do**
6: sample $\rho \sim U(0, 1)$
7: **if** $\rho < \epsilon$ **then**
8: select a random action uniformly $a_t \sim \mathcal{A}$
9: **else**
10: select $a_t = \text{argmax}_a (q(s_t, a))$.
11: **end if**
12: Do action a_t and observe $r(s_t, a_t)$ and s_{t+1}
$q_{t+1}(s_t, a_t) \leftarrow q_t(s_t, a_t) +$
13: $\alpha \left[r(s_t, a_t) + \gamma \max_a q_t(s_{t+1}, a) - q_t(s_t, a_t) \right]$
14: $t \leftarrow t + 1$
15: **end while**

q towards q_* is then assured [4] p. 220, when applying the step

$$q_{t+1}(s_t, a_t) \leftarrow q_t(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma \max_a q_t(s_{t+1}, a) - q_t(s_t, a_t) \right].$$

Here we introduce the learning rate $\alpha \in (0, 1)$, which corresponds to how large learning steps we take at each iteration, i.e. how much impact new information has on our q-function. It makes use of *Temporal difference (TD)-learning* [7] Chp. 6, meaning that we at every time step consider $r(s_t, a_t) + \gamma \max_a q(s_{t+1}, a)$ to be the prediction of our future return. Thus we rely on the fact that we are more certain about the future returns at the next state s_{t+1} than we are at s_t.

In order to achieve convergence of the q function, every state-action q-value needs to be updated. The algorithm thus relies on the possibility of visiting and learning from every state-action pair. We could, in theory, use this algorithm and thereby formulate an optimal strategy for Breakout. In practice, this is however infeasible. The state space is much too large and therefore a different approach is necessary.

C. Deep Q-Learning

To handle very large state spaces, we introduce a function approximator $\hat{q}_\theta = \hat{q}_\theta(s, a; \theta)$, where $\theta \in \mathbb{R}^M$ is a vector that parameterises the function approximator. These parameters are in this case the weights in a neural network. This is used in the *Deep Q-learning* algorithm which otherwise is similar to Q-learning, where we at time t use the TD-target $r(s_t, a_t) + \gamma \max_a \hat{q}_\theta(s_{t+1}, a)$. To improve the approximation $\hat{q}_\theta$ we define a *Loss Function* $\mathcal{L}(\theta)$ and attempt to minimize it,

$$\mathcal{L}(\theta) = \mathbb{E} \left[\left(r(s_t, a_t) + \gamma \max_a \hat{q}_\theta(s_{t+1}, a) - \hat{q}_\theta(s_t, a_t) \right)^2 \right].$$

The minimization is done by taking the semi-gradient of $\mathcal{L}(\theta)$ with respect to θ. That is,

$$\nabla_\theta \mathcal{L}(\theta) = \mathbb{E}\left[2\left(r + \gamma \max_a \hat{q}_\theta(s_{t+1}, a) - \hat{q}_\theta(s_t, a_t)\right)\nabla_\theta \hat{q}_\theta(s_t, a_t)\right]. \tag{5}$$

Estimating the expected value can be computationally demanding, the mean of $r + \gamma \max_a \hat{q}_\theta(s_{t+1}, a) - \hat{q}_\theta(s_t, a_t)\nabla_\theta \hat{q}_\theta(s_t, a_t)$ has to be calculated for every accessible state-action pair. It is often more favorable to optimize the loss function using stochastic gradient descent (SGD). Optimizing the loss function is then done using the gradient above without the expected value.

As introduced in Section I; this method has its drawbacks. By using the state s_{t+1} as our prediction at s_t we introduce a bias in our trajectory towards q_*. We also have a strong correlation between the samples if we perform the SGD on consecutive frames. To avoid these correlations, we introduce a concept formulated in [5] as *Replay Memory* $\mathcal{D}$. The transitions (s_t, a_t, r_t, s_{t+1}) are stored in $\mathcal{D}$, and the SGD is carried out using subsets (batches) of the replay memory, $(s_j, a_j, r_j, s_{j+1}) \sim \mathcal{D}$. This breaks the correlation and biases, reducing oscillations and other unwanted behavior in the SGD. Replay memory also enables the agent to learn from the same transitions multiple times — leading to greater data efficiency [5].

In order for SGD to converge, the target must be fixed. Using temporal difference: the target in (5) depends on the parameters that we are updating at every SGD step. We thus have a moving target, which more or less ensures that the SGD will not converge. The solution to this is to compute the TD-targets with respect to fixed parameters Θ. These are then updated to the current parameters θ after a specified number (C) of learning steps. This yields a periodically stationary target for the SGD,

$$\nabla_\theta \mathcal{L}(\theta) = 2\left(r + \gamma \max_a \hat{q}_\Theta(s_{t+1}, a) - \hat{q}_\theta(s_t, a_t)\right)\nabla_\theta \hat{q}_\theta(s_t, a_t).$$

The full Deep Q-learning algorithm as described in [2] is given in Algorithm 2. We note that it is always possible to split the time horizon of the MDP into episodes, which in the case of Breakout would correspond to one full game.

Algorithm 2: Deep Q-learning algorithm

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function $\hat{q}_\theta$ with random weights θ
Initialize target action-value function $\hat{q}_\Theta$, with weights $\Theta = \theta$
for each episode j **do**
  Initialize state s_0
  for $t = 1, \ldots, T_j$ in episode j **do**
    sample $p \sim U(0, 1)$
    if $p < \epsilon$ **then**
      select a random action uniformly $a_t \sim \mathcal{A}$
    else
      select $a_t = \text{argmax}_a(\hat{q}_\theta(s_t, a))$.
    end if
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random batch (s_j, a_j, r_j, s_{j+1}) uniformly from $\mathcal{D}$
    Set $y_j \leftarrow \begin{cases} r_j, & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_a \hat{q}_\Theta(s_{j+1}, a), & \text{otherwise} \end{cases}$
    Perform an SGD step on $(y_j - \hat{q}_\theta(s_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps update $\hat{q}_\Theta \leftarrow \hat{q}_\theta$
  end for
end for

III. Method

A. Environment and Game Characteristics

In order for the agent to interact with our chosen game environment we make use of the Open AI GYM library [8], which emulates the games with no prior assumptions about the controlling agent [8]. GYM is in turn built around the Arcade-Learning-Environment (ALE) [3]. In a given time step t the emulator accepts an action a_t out of the set of game-legal actions $\mathcal{A}$ and returns a reward along with a frame f_t, a 210×160 RGB image. Points (or rewards) are assigned by GYM in the same way as any human player would receive them. Points are received when the ball touches a block, removing it from the game after rebounding the ball back towards the pedal that the user controls (Fig. 1). There are six rows of bricks yielding more points the higher they are placed.

B. Preprocessing

To reduce the dimensionality of our network we preprocess the 210×160 RGB image by converting to greyscale and down-sampling. Lastly, we normalise the brightness between $[0, 1]$ and ensure the score is cropped out; preventing our neural network from learning from the in-game text, see Fig. 1. We denote this preprocessing as ϕ such that $\phi(f_t) = x_t$, resulting in a 84×84 pixel image.

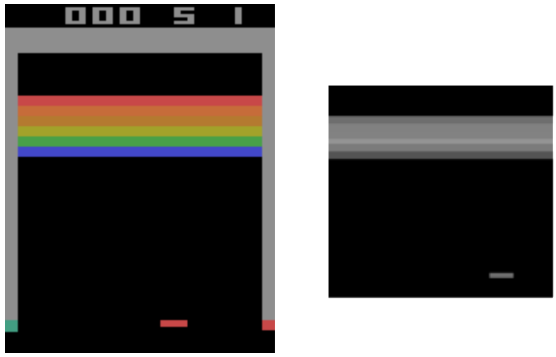

Fig. 1. The preprocessing filter applied to a frame in Breakout

Once preprocessed we stack four consecutive frames to construct the state $s_t = (x_{t-3}, x_{t-2}, x_{t-1}, x_t)$, which we

assume to fulfill the Markov property since it captures both direction and speed of in-game objects. New frames are then added by the *first in first out* (FIFO) principle. Our motivation for these choices lies in the earlier successes of [2], [5] which employ the same technique.

C. Neural Network

As covered earlier, we make use of a neural network to approximate the q-function. This network takes as input a state s_t and returns a q-value for each action available to our agent at time t. One could alternatively pass state-action tuples to the network, returning a scalar q-value. However, this requires a separate forward pass through the network for each action, increasing the computational cost linearly with the number of actions available.

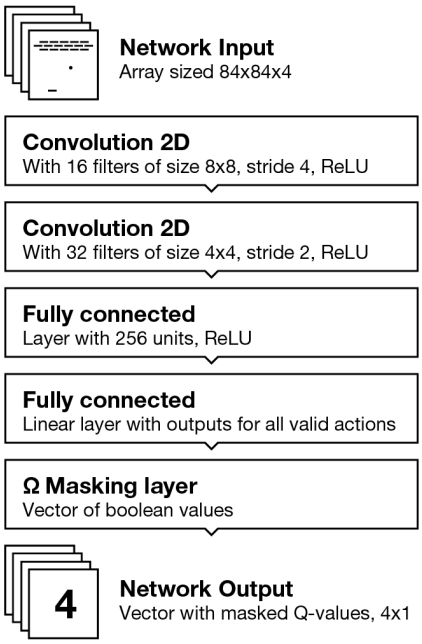

Fig. 2. Our network consists of two convolutional networks followed by two fully connected layers, and a masking feature, Ω. The mask is applied via element-wise multiplication, and is a vector of boolean elements with size $\mathcal{A}$. The hidden layers' parameters are taken from [5].

We adopt instead a similar neural network architecture to the ones used in [5], with the addition of a masking feature Ω. The network consists of two convolutional layers, the first of which takes our state s_t as input. This layer consists of 16 8×8 filters with a stride of 4 and a rectified linear unit (ReLU) activation function. The following layer convolves 32 4×4 filters with stride set at 2 and the same activation function. The final hidden layer is fully connected with 256 neurons and ReLU activation, followed by a fully connected output layer with one output for each possible action. This final outcome is then element-wise multiplied with Ω, a vector of boolean values ω_i of the same size as $\mathcal{A}$. During predictions the mask is set as $\omega_i = 1, \forall i$. During training we set $\omega_k = 1$ and $\omega_i = 0$ for $i \neq k$, where k corresponds to the action taken in that iteration.

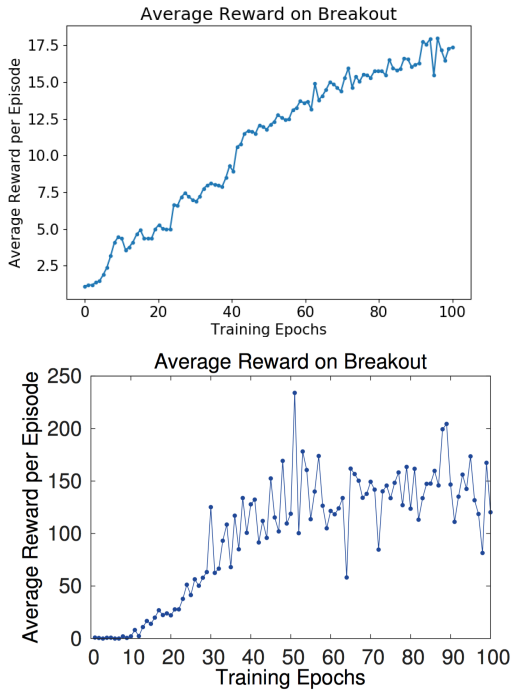

Fig. 3. Our results of average game score during training. Compared to the results of [5] (bottom). One training epoch corresponds to 50,000 learning batches to the neural network.

The minimization of the loss function $\mathcal{L}(\theta)$ is done using *RMSProp*, an SGD method including momentum [9]. We also bound the gradient between $[-1, 1]$ (*clip-norm*), reducing the likelihood of SGD diverging. This could, however, reduce the speed of convergence. Finally, the initial values of the parameters θ are set using a normal distribution $N(0, \sqrt{2/n})$ in the two convolutional layers. For the fully connected layer, the parameters are initialized using a uniform distribution $[-k, k]$, where $k = \sqrt{6/(n+m)}$. Here n is the number of input units and m the number of output units in the weight tensors [10]. For a full list of hyperparameters see Appendix.

IV. Results

Training our algorithm for 5,000,000 learning batches (100 training epochs) yielded the results shown in Fig. 3 and 4. These have been plotted side by side with the state of the art results recorded from [5].

After training our agent we applied an ϵ-greedy policy with $\epsilon = 0.05$. We record the result of this in Fig. 5 together with the corresponding scores of a random-action agent, as well as a typical human score as listed in [5].

Type of Player	Average score
Our agent*	20
Agent in [5]	168
Random agent	1.2
Human player	31

Fig. 5. *Average scores after 5 million learning iterations. We compare this to: a random-action agent, a typical human score and agent as recorded in [5]. Our average is calculated over one epoch.

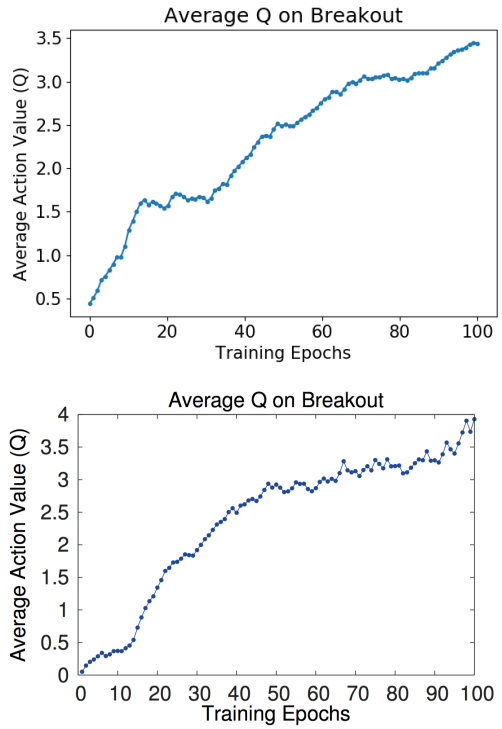

Fig. 4. Our results of average $\max_a \hat{q}$ during training (top). These are evaluated on a set of 5000 states sampled prior to training. Comparable to the results of [5] (bottom). One training epoch corresponds to 50,000 learning batches to the neural network.

V. Discussion and Analysis

A. Performance

By reviewing Fig. 3 we can be certain that our agent is, in fact, learning how to play Breakout. The learning rate is however comparatively slower than that achieved by [5]. After training for 5,000,000 learning iterations and using an ϵ-greedy policy with $\epsilon = 0.05$, our agent achieved a mean score of 20 per episode. Comparatively [5] averaged approximately 168. We note however that our predicted q-values throughout training are very similar to [5], as shown in Fig. 4. This leads us to believe that the differing results are not a consequence of our implementation of the Deep Q-learning algorithm.

We suspect that the Neural network architecture is in part to blame. There are a number of ambiguities in the original paper [5]. For instance the minimization of the cost function can be complemented with techniques like *gradient clipping* which limits the maximum values of the gradients. Here [5] does not explicitly mention if this is applied, whereas a later paper on the topic does [2]. We chose to apply the error clipping as mentioned in the latter.

The definition of terminal states is another aspect that could affect the result. The end of a game (loss of five lives) is how the terminal state is predefined in ALE and chosen in this paper. One could, however, choose the loss of one to be the terminal state definition, which most likely would affect the final result.

We also make use of the predefined action space in ALE, that is, $\mathcal{A}$ =['NOOP', 'FIRE', 'RIGHT', 'LEFT']. Where 'NOOP' corresponds to performing no actions, and 'FIRE' is how to restart after losing a life. One could therefore implement a 'FIRE'-action after every life loss and thereby remove it from the actions available to the agent. This would then result in an action space with a smaller dimension — possibly speeding up the training of the agent. We are unsure how this was implemented in [5], as it appears they do not make use of the GYM library.

We discovered several such subtleties in our review of [5] and resorted to our best judgment or, when possible, the parameters used in [2]. This is in part due to the computation time involved in experimenting with hyperparameters.

B. Implementation

Correlating the changes in our hyper-parameters or implementations with the performance of the code proved difficult. The computational time required to produce any reliable results was for our hardware in the time scale of several days, making an iterative process impossible with our time constraints. Instead, we made several changes at a time, relying heavily on the parameters that have been shown to be effective.

Despite this, we were able to identify a few factors that sped up our code significantly. The addition of a masking layer meant we could skip a forward pass through the network, speeding up calculations. We also vectorized our code with *Numpy arrays*, which have been heavily pre-optimized and are significantly faster than native python data structures, particularity if the code is run over many threads in a GPU.

Adapting our definition of what constitutes our Markov state also improved the agent's learning rate. When two different states had overlapping frames, that is $s_{t+k} = (x_{t-3+4k}, x_{t-2+4k}, x_{t-1+4k}, x_{t+4k})$ and $s_{t+k+1} = (x_{t-2+4k}, x_{t-1+4k}, x_{t+4k}, x_{t+1+4k})$ the agent's learning rate was extremely slow. This was improved by redefining the states such that $s_{t+k} = (x_{t-3+4k}, x_{t-2+4k}, x_{t-1+4k}, x_{t+4k})$ and $s_{t+k+1} = (x_{t+1+4k}, x_{t+1+4k}, x_{t+3+4k}, x_{t+4+4k})$. We suspect this was similarly applied in [5], however it is not explicitly stated.

C. Future adaptations

Occasionally new techniques emerge in RL that are shown to improve the agent's performance, such as the following.

Double Q-learning: Double Q-learning attempts to counteract any overestimation of the action value predictions. It does this by introducing a second q-function, used to decouple the action-selection from the q-value prediction. The TD-target is then estimated using both networks,

$$r(s_t, a_t) + \alpha\, \underbrace{q\left(s_{t+1}, \text{argmax}_a\, \underbrace{q\left(s_{t+1}, a\right)}_{\text{current network}}\right)}_{\text{target network}}.$$

This is also known as *actor critic* and improves the estimation of q-values — resulting in a more stable training [11].

Prioritized Experience Replay: We currently make use of *Replay memory* to break the TD-target correlations. Sampled transitions are stored in the memory uniformly, regardless of their significance. Prioritized Experience Replay (PER) exploits that the RL agent learns more efficiently from some transitions than from others. PER introduces a non-uniform probability distribution in the replay memory, based on the significance of the stored transitions. This biases the agent to learn more selectively from those, improving the learning rate [12].

Parallel Threads: The efficiency of the DQN algorithm could be improved by running several game environments in parallel. Each instance of the environment has a respective actor, which explores the state-action space independently. This method does not make use of a replay memory. The algorithm relies instead on the aforementioned actors to decorrelate the transitions sent to the neural network. This kind of multi-threaded architecture can be efficiently applied to powerful machines with multiple cores or GPUs, increasing the performance of the algorithm [13].

VI. Conclusion

In this paper we implement a reinforcement learning algorithm with the aim to reproduce the result in [5]. After training our agent learns to play Breakout just beneath the level of a typical human, with an average score of 20 compared to the human score of 31 and a result of 168 achieved by [5].

We discuss and attribute a few probable causes for this discrepancy, primarily in the form of ambiguities in the network configuration of [5]. Due to the heavy computational load of training the agent; an iterative and experimental process was not possible.

Finally, we discuss what improvements could be made to better the agent in the future. We introduce a number of possible techniques from recent studies: *Double Q-learning*, *Prioritized Experience Replay*, and *Parallel Threads*.

Appendix
Hyperparameters

Acknowledgment

The authors would like to thank their supervisors Alessio Russo and Damianos Tranos for their invaluable support and guidance throughout the project.

References

[1] Y. Shoham, R. Perrault, E. Brynjolfsson, J. Clark, J. Manyika, J. C. Niebles, T. Lyons, J. Etchemendy, B. Grosz, and Z. Bauer. (2018) The ai index 2018 annual report. AI Index Steering Committee, Human-Centered AI Initiative. Stanford University, Stanford, CA. Accessed Feb. 2019. [Online]. Available: http://cdn.aiindex.org/2018/AI%20Index%202018%20Annual%20Report.pdf

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[3] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, Jun. 2013.

[4] C. J. C. H. Watkins, "Learning from delayed rewards," *King's College*, May 1989. [Online]. Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, Feb. 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[6] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. New Jersey: John Wiley & Sons, 2014.

[7] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA: MIT Press, 1998.

[8] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. (Jun. 2016) Openai gym. [Online]. Available: https://arxiv.org/pdf/1606.01540

[9] T. Tieleman and G. Hinton, "Rmsprop gradient optimization," accessed Apr. 2019. [Online]. Available: http://https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[10] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://keras.io

[11] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[12] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2016.

[13] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," *arXiv preprint arXiv:1611.05397*, 2016.