Describing the role of Artificial Neural Networks in Reinforcement Learning

Thilo Stegemann
University of Applied Sciences
Applied Computer Science
12459 Berlin, Wilhelminenhofstraße 75A
Email: t.stegemann@gmx.de

***Abstract*—In the following scientific research we will answer the question: what role do artificial neural networks have in reinforcement learning? To answer this question we will first explain what reinforcement learning is and how to learn an approximately optimal strategy (policy) in an unknown sequential stochastic environment. Key parts for learning approximately optimal strategies are reinforcement learning methods like Q-Learning and artificial neural networks for approximating strategy functions. Mnih et al. 2013 implemented a deep Q network which uses a convolutional neural network in combination with a reinforcement learning algorithm (Q-learning) to learn strategies in 7 different Atari 2600 games. Inside this paper we will explain in detail how this deep Q network is implemented, what convolutional neural networks are and which results are achieved by this and similar approaches.**

I. Introduction

In this paper we will concentrate on the use of artificial neural networks (ANN's) in particular on convolutional neural networks (CNN's) in Reinforcement Learning (RL). To understand the relationship between those two big concepts we explain key parts of both. For RL key parts are the problem definition, especially sequential stochastic decision processes (Markov Decision Process), discounted sums of delayed rewards, policy functions, and value functions and approximation approaches of those functions. Additionally it is needed to define what the RL algorithms have to achieve in form of a loss function (also called objective or cost function) and how this function can be optimized with gradient methods. This mechanism of defining a loss function and optimize it with gradient methods is also widely used in other machine learning sub domains like supervised learning. RL is one of three main machine learning sub domains: Supervised Learning (SL) where a ”teacher” defines whether something is good or bad, Reinforcement Learning (RL) where no ”teacher” is given and the algorithm only learns from trial and error experience and delayed reward signals and Unsupervised Learning (UL) where no ”teacher” or reward signal is given and the algorithm learns considering the special clustering or structure of the input data. Inside the paper we will strongly concentrate on reinforcement learning, but sometimes we refer to supervised learning for a better understanding of certain concepts.

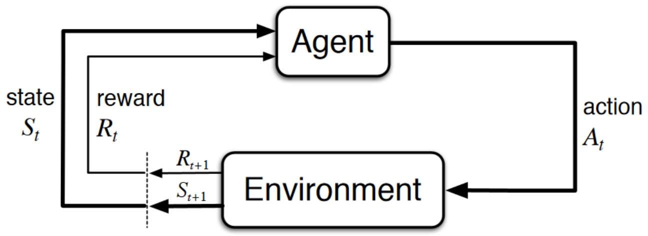

Fig. 1. Agent-environment interaction [1].

Policy and value functions represent the behaviour and learning result of an RL algorithm. One generalization approach (as we will see later) is approximating policy and value functions using ANN's. For ANN's key parts are convolutional neural networks (CNN'S), recurrent neural networks (RNN's) and backpropagation. CNN's and RNN's are special kinds of ANN's. CNN's are often used for image classification and RNN's are often used in sequential settings when we have input over time. Combination of both special types are also possible and practically used. The paper will focus only on CNN's and not on recurrent neural networks, artificial neural networks in general or combinations of CNN's with RNN's. Although those concepts can be powerful and promising as well, we will define the focus of the paper on CNN's and RL. Another approach for approximating policy and value functions is using a linear combination of features, but this will not be part of this paper as well. After we explained both concepts RL and CNN's we switch to an in depth explanation of a practical case study done by scientific researchers, which used a CNN for approximating a Q-function. We will have a closer look onto the results and how they realized their experiment. Last part of the paper is a discussion and conclusion part about how well ANN's are used in RL and other successful deep RL approaches.

II. Reinforcement Learning (RL)

Reinforcement learning (RL) problems consider an agent-environment interaction framework. Basics of reinforcement learning are mentioned in [2]–[6]. As an in depth guide for reinforcement learning see [1]. The following part is about summarizing those background RL introductions. The agent (implementation of the learning algorithm) will interact with the environment (a Markov Decision Process). The interaction is continuous in time t, so the start state at time $t = 0$ is s_0 and a trajectory (decision sequence) looks like $(s_0, r_0) \to a_0 = (s_1, r_1) \to a_1 ... a_{t-1} = (s_t, r_t) \to a_t$. The agent environment interaction is graphically displayed in Fig. 1. The agent will get a reward R_t and a state S_t from the environment and the environment will get an action A_t from the agent. This action A_t is a calculated decision based on the received reward R_t and state S_t. After the environment received action A_t it will return a state S_{t+1} and a reward signal R_{t+1}. The agent tries to learn optimal behaviour through trial and error attempts. The agent wants to know which actions in which states get the most long-term reward and fit this knowledge into a policy representation. A few main problems of this RL framework are:

- The agent only gets a numerical reward from the environment at the end of a decision-sequence.
  $\sim$ *Delayed Reward*
- How should the reward be assigned to the different steps of a decision-sequence?
  $\sim$ *Credit Assignment Problem*
- How to handle vast action- and state-spaces?
  $\sim$ *Generalization Problem*

This paper focuses heavily on the last mentioned problem. How to handle vast action- and state-spaces? In my bachelor thesis I also had the problem of high dimensional action- and state-spaces. I implemented a variant of table lookup Q-learning with a SQLite database for storing the agent experience. The agent should try to learn TicTacToe and Reversi (two board strategy games). Although the algorithm did OK for the TicTacToe (3x3 board) problem it completely failed for bigger game fields of TicTacToe or Reversi. The reason for failure was the exponential time increase proportional to the increasing dimensionality of action and state spaces. Q-functions cannot be represented in a table lookup, because the dimensions of most RL problems will lead to databases with more entries then particles in the universe (compare chess board positions and actions [7] S. 114 ff). One big and promising solution for this problem is function approximation in general and artificial neural networks in concrete.

A major goal of RL is to find a global optimal policy. A policy is a function which maps states to actions. This policy will additionally get a vector of parameters. The parameter-vector changes the policy output. This parametrisation of the policy function is called "function approximation" and Artificial Neural Networks are an approach for approximating a policy function. Combining ANN's with reinforcement learning algorithms have so far shown spectacular results e.g.: The Google DeepMind Team programmed an AI which plays Go (a very complex strategy board game) at human grandmaster level [5]. With this approximation the problem of vast action- and state-spaces can be solved. To optimise the parameter vector, methods like Policy Gradient or Temporal Difference (e.g. Q-Learning) approaches are used. Applications like TD-Gammon by Gerald Tesauro proved that learning complex strategy games with Artificial Neural Networks is possible and promising.

A. Q-learning

Q-Learning (Watkins [8], 1989) is a reinforcement learning algorithm for agents to learn how to act optimally in controlled Markov decision processes. Watkins showed in his paper that Q-Learning converges to the optimum action-value with probability 1 so long as all actions are repeatedly sampled in all states and the action-value are represented discretely. We will describe the Q-Value update in detail now, because in a later chapter Q-Learning is a fundamental part of the implementation. The update rule of one-step Q-Learning is [1]:

$$\begin{aligned} Q(S_t, A_t) \leftarrow & Q(S_t, A_t) + \alpha[R_{t+1} + \\ & \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \end{aligned} \tag{1}$$

This equation defines how to update a Q-value in one time step. Step size (or learning rate) α should be $0 \leq \alpha < 1$ but often closer to 0. This hyper parameter defines how much the agent can trust its experience. So if α is close to 0 that means the agent should update its Q-values just a little bit in the direction of the temporal difference (TD), because the complete Q-Function is initialized randomly and the agent can't trust those values completely until a certain amount of updates is done. The temporal difference is the mathematical difference between two timely successive Q-Values: $\max_a Q(S_{t+1}, a) - Q(S_t, A_t)$. TD represents the error between the two Q-Values, if TD is 0, then there is no difference between the states. Another hyper parameter is γ which is a discounter. γ defines how relevant future experience is for the agent. If γ is close to 1 or is 1, then there is no discounting and every experience is equal important. If γ is close to 0, then future experience gets more irrelevant proportional to the time distance. When γ is 0, then no future experience is considered at all. $\max_a$ is a function which should return the highest Q-value for every possible action a in state S_{t+1} (also denoted as S').

B. Loss Function

A loss function $J(f)$ (or objective function) always defines how good or bad a function f is. The result of a loss function is a scalar. Sometimes the loss function is also called a cost function, because if the resulting scalar is a high value, then its like the cost of function f is high. There are several different loss functions for different tasks. In supervised learning tasks for example linear regression a mean squared error (MSE) loss is used. Mathematically MSE loss looks like this:

$$J_D(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})^2).$$

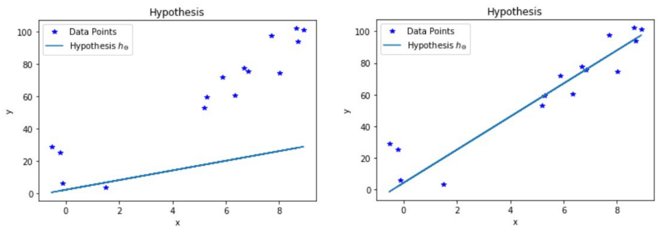

Fig. 2. Fitting a linear hypothesis with linear regression.

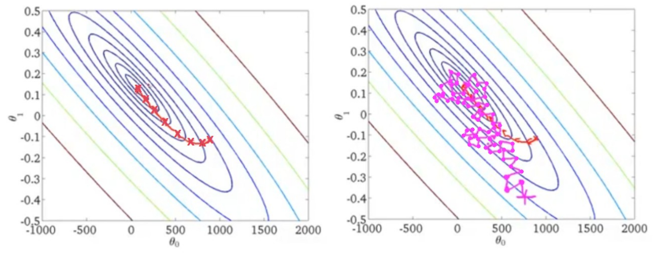

Fig. 3. Contour plots of cost function $J(\theta)$ with marked batch (left) and stochastic gradient descent (right) [11].

D is the data or the training set which includes all training values x and all target values y (labels). m is the amount of training examples inside D. Θ is a parameter vector containing several different Parameters $(\theta_0, \theta_1, ..., \theta_n)$. Parameter vector Θ affects the output of the linear hypotheses h_Θ. Using the MSE loss function $J_D(\Theta)$ and a gradient descent method it is possible to fit the linear hypothesis to the given data. Fig. 2. displays the start point and the result of a linear regression as described above. The blue data points are not measured data, these points are calculated with a Gaussian normal distribution. So to all y values a Gaussian noise is added. The left graph is the not fitted random initialized hypothesis h_Θ and the right graph is the fitted hypothesis after 400 iterations using MSE loss and gradient descent. For completeness the parameter vector Θ is optimized and so the hypothesis will fit the data, because the hypothesis h_Θ is dependent on Θ.

In reinforcement learning tasks loss functions look and behave similar as in supervised learning. Later we will examine a successful Q-Learning implementation with a neural network as function approximation using a loss function $L_i(\theta_i)$ and to understand this loss function it is helpful to understand the example loss function given above.

C. Batch vs. Stochastic Gradient Decent

In the privious chapter we already talked about gradient descent metodes, but we did not defined how gradient descent works. Now we will explain two different gradient descent methodes in detail. The following differentiation of batch and stochastic gradient descent is based on [9] and [10]: In general gradient methods are used to optimize a function respective to its partial derivatives. A parameter update with batch gradient descent will consider the whole training set for its calculation. So in large scale machine learning problems there are training sets with several millions or billions of examples. For every update step the batch gradient descent calculates the sum of the partial derivatives in respective to all examples. To find the minimum of a function multiple update steps are needed. Batch gradient descent will have an exponential computational cost for larger machine learning problems.

A Solution for this problem is the stochastic gradient descent. Instead of computing the gradient of the function exactly, each iteration (update step) estimates the gradient on the basis of a single randomly picked example. The trade-off between batch and stochastic gradient descent is that batch gradient descent achieves linear convergence, when the initial estimate is close enough to the optimum and when the gain of the discounting factor is sufficiently small. The stochastic gradient descent will not converge to a minimum like batch gradient descent does, rather the parameters which are updated will oscillate around the minimum but will may never converge to the minimum. But often the stochastic gradient descent gets the parameters close to the minimum much faster than batch gradient descent.

Algorithm 1 Stochastic gradient descent [11]

```
Randomly shuffle (reorder) training examples
repeat
    for $i := 1, ..., m$ do
        for $j := 0, ..., n$ do
            $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
        end for
    end for
until 1-10 times
```

Figure 3 illustrates how batch and stochastic gradient descent converge to a minimum. The coloured ovals define the contour of a cost function $J(\theta)$. The cost function is minimal in the middle of the smallest oval and gets higher in the outer ovals. The parameters θ_0 and θ_1 are two parameters of a parameter vector θ. Those parameter values will change the cost of function $J(\theta)$. In the left part of figure 3 parameters θ updated by batch gradient descent converges "relatively straight" to a minimum after several iterations. Whereas in the right part of figure 3 parameters θ updated by stochastic gradient descent are in general moved in the direction of the minimum but not always and in the end the parameters are wondering around close to the minimum.

Algorithm 1 is a pseudocode example of stochastic gradient descent from Andrew Ng [11]. The term m denotes the amount of training examples, n is the amount of parameters θ, hyper parameter α is the step size or learning rate which we already mentioned in chapter Q-learning and $h_\theta(x^{(i)}) - y^{(i)}$ is a part of the error term mentioned in chapter loss function. For a concrete example with $m = 300.000.000,00$ training

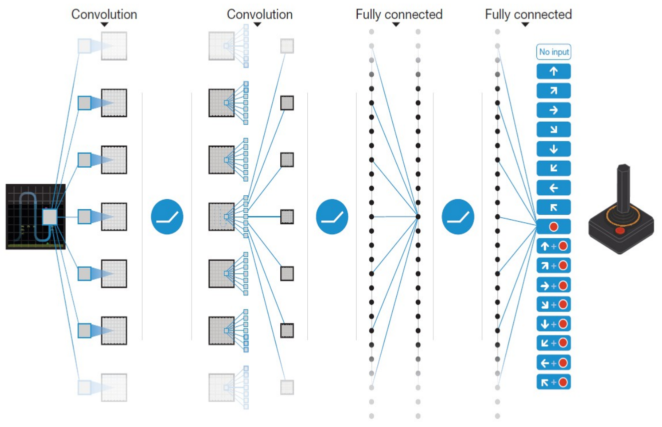

Fig. 4. Schematic illustration of the convolutional neural network [4].

examples the algorithm 1 will update all parameters θ_n for every training example. So in one single stochastic gradient descent step the parameter vector $\Theta = \theta_0, ..., \theta_n$ is updated m times. Batch gradient descent will in contrast calculate an update of all parameters θ_n considering all training examples at once and then just performing one single batch gradient descent update of the parameter vector Θ.

III. Convolutional Neural Network (CNN/ConvNet)

To better understand the later deep Q-learning algorithm (also called deep Q network (DQN)) [6] it is helpful to get into CNN's first. Knowledge base of this chapter is a Stanford class CS231n about "Convolutional Neural Networks for Visual Recognition" [12]. A convolutional neural network is a specific ANN which assumes that every input is an image. This assumption allows to reduce the amount of parameters and so improve the performance for image processing with CNN's in contrast to more general ANN's. More general because ANN's don't assume a specific input. A CNN consists of layers. If there is more then one layer between input and output layer, then the neural network is called a deep neural network. We can imagine the layers like human neurons which will get input, have output depending on the input and they are all connected to other neurons. There are different kinds of layers: input layer (INPUT), convolutional layer (CONV), rectified linear unit layer (RELU), pooling layer (POOL), fully connected layer (FC) and an output layer. The different layers mentioned before are displayed in Fig. 4. Some of these layers can appear multiple times. A simple ConvNet architecture for classification could be INPUT $\rightarrow$ CONV $\rightarrow$ RELU $\rightarrow$ POOL $\rightarrow$ FC. In the following every different layer will be explained in detail.

a) The Input layer: can be high dimensional sensory data. Considering the Atari 2600 arcade gaming environment [2]–[4] the input is a video stream of pixels at different time steps. At time step t a video signal reduces to just an image of pixels and the complete sequence of images at all time steps equals the video signal. There is still more then the raw pixels

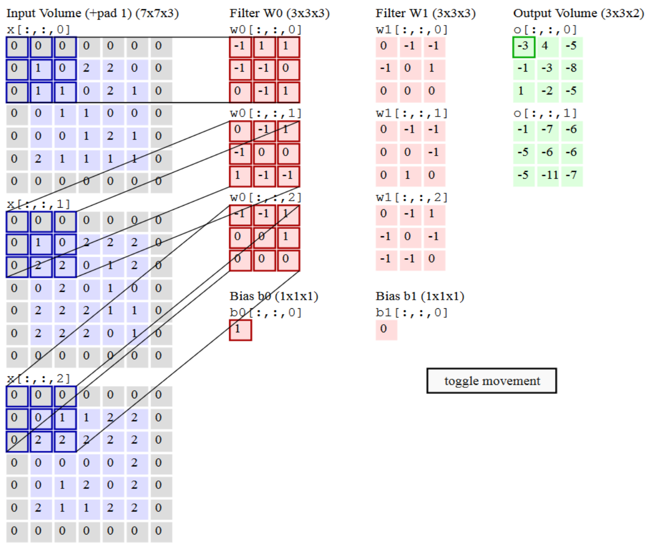

Fig. 5. Computation inside a convolution layer [12].

from the video stream like the score value at each time step [4]. In terms of reinforcement learning the score signal is a reward signal and the video stream at each time step describes the state in which the agent is in. Another example is the CIFAR-10 dataset which contains images of shape 32x32x3 (32 wide, 32 high, 3 color channels).

b) The convolution layer: is the most computationally expensive layer, because much matrix multiplications are performed on raw data (not reduced data expect preprocessing). In general this layer applies filters on input images. A filter is a window with fixed size. For the CIFAR-10 example the filters could have a size of 5x5x3. These filters are shifted around the width and height of the input images and for every position a matrix multiplication between the input image and the filter is performed. The matrix multiplication results in scalar values stored in a result matrix (feature map). Every filter produces an own feature map. Filters represent the weights of CNN's. For every filter there is an additional bias weight which need to be considered inside the computation. A filter can be a representation for edges, lines or other shapes. A combination of those low level filters results in more complex filters like an eye or an ear. Those low and high level filters are like extracted features from the input image. The output of this convolutional layer is controlled by three hyperparameters: depth, stride and zero-padding.

The computation inside a convolution layer is represented in Fig. 5. Three input matrices of shape 7x7 are independently drawn, because each matrix represents the red, green or blue color channel so the input layer has a shape of 7x7x3 (7 wide, 7 high, 3 color channels). The outer lines (matrix borders) are all zero, because the hyperparameter Zero-padding is set to 1. The hyperparameter stride is set to 2 and so the filter

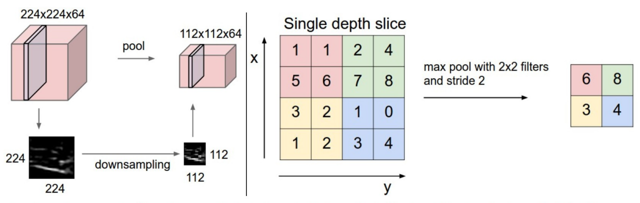

Fig. 6. The pooling layer [12].

mask will always move 2 pixels. The hyperparameter depth is set to 2 and so there is a filter $W0$ and a filter $W1$ both of shape 3x3x3 (3 pixels width, 3 pixels heigh and 3 color channels one for each input color channel matrix). The values of these three hyperparameters decide over successful or unsuccessful convolutions. Fig. 5 just shows one single step of the convolutional computation. In this single step the filter $W0$ is applied on the top left side of the input matrix. Result of this matrix multiplication is just a single value stored in an feature map (Output Volume) for filter $W0$. The next step would be move the filter mask 2 pixels left (because stride = 2) and apply filter $W0$ again resulting in a new scalar value stored inside the feature map of $W0$. If the feature map is complete, then the whole process starts with the next Filter $W1$.

c) The rectified linear unit layer: will apply an elementwise activation function, such as the $max(0, x)$ thresholding at zero. If $max(0, x)$ gets values below zero then it will return just zero and if the values are greater then zero max will return the value itself. The blue circles in Fig. 4 with a white line after CONV and FC layers represent the RELU activation function. The $max(0, x)$ rectifier is used for deep learning rather then e.g. the logistic sigmoid, because of better practical efficiency. The RELU layer is needed for an appropriate update of the neural network weights.

d) The pooling layer: often referred as max pooling, reduces the dimension of the input volume. Fig. 6 illustrates how pooling works. In simple max pooling behaves like the convolutional layer, but the filter will just return the highest (maximal) value at a position. In the example a 2x2 filter with a stride of 2 will have 4 different positions at the single input 4x4 slice. For every position it returns the maximum value so max pooling reduces a 4x4 input matrix to a 2x2 output matrix.

e) The output layer: is a one dimensional vector of probabilities. Every probability refers to a class like a car or a cat. So when we input an image of a dog inside a properly trained CNN then the highest value inside the output vector should be referencing the class dog, otherwise the CNN did not classify the image correctly (several reasons for wrong classifications). To produce the output probability vector a function called Softmax is used. Softmax normalizes a K-dimensional vector z of arbitrary real values to a K-dimensional vector $\sigma(z)$ of real values in the range $[0, 1]$ that add up to 1.

IV. Playing Atari with Deep Reinforcement Learning

The paper from DeepMind Technologies is about a deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning [6]. They defined the model as a convolutional neural network (CNN). This CNN is trained with a variant of Q-learning. Input of the CNN is row pixels and output is a value function estimating future rewards. They apply this deep learning model to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. Result of this experiment is that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

A. Deep Reinforcement Learning

The following subsection is an in depth explanation of the experiment done in paper ”Playing Atari with Deep Reinforcement Learning” [6]. Inside the deep Q-learning algorithm (see Algorithm 2) every concept explained in previous chapters are involved. This algorithm describes a deep Q network (DQN). A deep Q network is a combination of a convolutional neural networks and the reinforcement learning algorithm Q-learning. The problem setting is a Markov Decision Process defined by the Atari 2600 games emulator. The algorithm aims to learn an optimal policy like reinforcement learning agents always try to in an MDP environment.

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s, a; Q_i))^2] \tag{2}$$

Furthermore a sequence of loss functions displayed in equation 2 is used by Mnih et al. 2013 [6]. The core of the loss function equation 2 is pretty similar to the Q-function of Sutton and Barto [1] equation 1 or the original from Watkins [8]. The similarity appears because of the objective definition in Reinforcement Learning problems. RL objective is to maximize the accumulated rewards over time. This RL objective is mathematically formalized either in Sutton and Bartos one step Q-update rule (equation 1) and in Mnih et al. equation of loss functions (equation 2).

$$\begin{aligned}\nabla_{\theta_i} L_i(\theta_i) =& \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}[(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) \\ &- Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)]\end{aligned} \tag{3}$$

These loss functions are optimized by a stochastic gradient descent update referenced in equation 3.

$$Q^*(s, a) = \mathbb{E}_{s'\sim\mathcal{E}}[r + \gamma \max_{a'} Q^*(s', a')|s, a] \tag{4}$$

Mnih et al. used a convolutional neural network as function approximation to estimate the true Q-value-function $Q^*(s, a)$ defined in equation 4. So the approximation is a Q-function $Q(s, a; \theta)$ with an additional parameter vector of weights θ. The deep Q-learning algorithm optimizes the weights of the convolutional neural network function approximation for an nearly optimal approximation of $Q(s, a; \theta) \approx Q^*(s, a)$.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	-20.4	157	110	179
Sarsa [3]	996	5.2	129	-19	614	665	271
Contingency [4]	1743	6	159	-17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	-3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	-16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Fig. 7. DQN results in Atari 2600 emulator [6].

Initially the weights of the CNN are randomly initialized, so the entries (neurons) will not behave exactly homogeneous.

Algorithm 2 Deep Q-learning with Experience Replay [6]

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed
        sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$
            and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions
            $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \\ & \text{for non-terminal} \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_i, a_j; \theta))^2$
            according to equation 3
    end for
end for
```

B. Experiment results

The deep Q-learning algorithm from the previous subsection achieved better performance than an expert human player on the Atari games Breakout, Enduro and Pong and achieves close to human performance on Beam Rider. The human expert outperformed the DQN in games like Q*bert, Seaquest and Space Invaders, because the games are more challenging in terms of the DQN needs to find a strategy that extents over long time scales. Nevertheless the DQN outperforms other RL algorithms like Sarsa and Contingency. These results are summarized in table 7. The first row of the table is a random agent which acts randomly without any RL algorithm. The next 4 rows are already described previously. The last 3 rows compare the best DQN results with a evolutionary policy search approach.

V. Discussion & Conclusion

Even if the DQN from Mnih et al. 2013 outperforms a human in several Atari 2600 games, there are still games where the human is unbeatable. The paper "Deep Reinforcement Learning with Double Q-Learning" aims to determine if the recent DQN (Deep Q Network) algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain [3]. Furthermore the Google DeepMind contributors point out how the Double Q-learning algorithm can be generalized to work with large-scale function approximation to successfully reduce the DQN overoptimism, resulting in more stable and reliable learning. Finally they propose a specific adaptation to the DQN algorithm and show that the resulting algorithm (Double DQN) not only reduces the observed overestimation, as they hypothesized, but that this also leads to much better performance on several Atari 2600 games.

The main paper of Mnih et al. 2013 [6] is the foundation for another very similar paper "Human-level controll through deep reinforcement learning" [4]. The paper is about how to reach human-level control through a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. The deep Q-network agent is tested on the challenging classic Atari 2600 game environment. The result of this test demonstrated that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters.

Another promising approach using artificial neural networks as function approximation is created by scientists from Google DeepMind and Montreal Institute for Learning Algorithms, they introduced asynchronous deep learning algorithms [2]. These asynchronous algorithms are based on four standard reinforcement learning algorithms: One-step Q-learning, one-step Sarsa, n-step Q-learning and advantage actor-critic. We already mentioned one-step Q-learning in a previous section. The paper explains the background of reinforcement learning and how the asynchronous reinforcement learning methods works. The study was approved by an experiment in an Atari 2600 evaluation environment. All four asynchronous algorithms where tested within the test environment. The Atari 2600 environment tests where used to compare the performance of the four algorithms. The main finding of this study is that all four asynchronous deep reinforcement learning algorithms are able to train neural network controllers on a variety of domains in a stable manner. In addition their results show that stable training of neural networks through reinforcement learning is possible with both value-based and policy-based methods, off-policy as well as on-policy methods, and in discrete as well as continuous domains.

So far we only mentioned RL environments given by the Atari 2600 emulator, but what if we would define an extremely high dimensional board game like GO as an RL environment? The paper from D. Silver et al. 2016 concerns two different algorithm approaches for deep reinforcement learning [5] of the board game GO. The first approach uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games and reinforcement learning from games of self-play. They proof through experiments that this deep RL algorithm approach is capable of playing Go at the level of state-of-the-art Monte Carlo tree search. The second deep RL approach is a new seach algorithm that combines Monte Carlo simulation with value and policy networks. They used this search algorithm inside the application AlphaGo and the application achieved a 99.8% winning rate against other Go programs and it defeated the human European Go champion by 5 games to 0.

To close out the paper and give a final statement we will summarize this paper in short. We explained what reinforcement learning in general is, what the problems of reinforcement learning are, how to solve the problem of high dimensionality with function approximation using convolutional neural networks as function approximation, what convolutional neural networks are, how loss functions look and how they define the agent objective, what the difference between the optimization methods batch gradient decent and stochastic gradient descent are and we explained how all these concepts are used by Mnih et al. 2013 [6] to create a deep RL algorithm which outperforms human level gameplay in several Atari 2600 games. Convolutional neural networks which are more specialized artificial neural networks combined with RL methods produce powerful approaches to learn human level or above human level strategies in most of Atari 2600 like games or GO without changing hyperparameters or network architecture mostly. In several years the deep RL algorithms may beat humans in every Atari 2600 game.

References

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

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[3] 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

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, and D. H. S Legg, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015. [Online]. Available: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[7] W. Ertel, *Grundkurs Künstliche Intelligenz: eine praxisorientierte Einführung*, 2nd ed. Wiesbaden: Vieweg + Teubner, 2009.

[8] C. J. C. H. Watkins and P. Dayan, "Technical note: q-learning," *Mach. Learn.*, vol. 8, no. 3-4, pp. 279–292, May 1992.

[9] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," in *Advances in Neural Information Processing Systems*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds. NIPS Foundation (http://books.nips.cc), 2008, vol. 20, pp. 161–168. [Online]. Available: http://leon.bottou.org/papers/bottou-bousquet-2008

[10] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, Y. Lechevallier and G. Saporta, Eds. Paris, France: Springer, August 2010, pp. 177–187. [Online]. Available: http://leon.bottou.org/papers/bottou-2010

[11] A. Ng. (2017) Lecture 17.2 - large scale machine learning — stochastic gradient descent - [andrew ng]. [Online]. Available: https://www.youtube.com/watch?v=W9iWNJNFzQI

[12] A. Karpathy. (2017) Cs231n convolutional neural networks for visual recognition. [Online]. Available: http://cs231n.github.io/convolutional-networks/