Explanation and comparison of deep learning models and improvement approaches used in Reinforcement Learning.

Denisz Mihajlov
HAW, Hamburg
Denisz.Mihajlov@haw-hamburg.de

March 6, 2022

Abstract

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. The purpose of reinforcement learning is for the agent to learn an optimal, or nearly-optimal, policy that maximizes the "reward function" or other user-provided reinforcement signal that accumulates from the immediate rewards. In this area are existing a lot of algorithms that can be used for different types of tasks from playing computer games till autonomous driving and robotics. This paper describes and compares some of these methods to give more understanding about them and to show if the combination of them can make learning processes more successful. All comparisons between different methods were done based on real tests that show how the agents are developing themselves during the learning process.

1 Introduction

Reinforcement Learning (RL) is the science of decision-making. It is about learning the optimal behaviour in an environment to obtain maximum reward. This optimal behaviour is learned through interactions with the environment and observations of how it responds, similar to children exploring the world around them and learning the actions that help them achieve a goal. The Reinforcement Learning problem involves an agent exploring an unknown environment to gain a maximum reward. RL is based on the hypothesis that all goals can be described by the maximization of expected cumulative reward. It exists a big amount of algorithms and network architectures that can be used in Reinforcement Learning. These approaches can be used to successfully train an agent to perform some actions. For example, they are being used to create agents that can play games automatically and reach scores bigger than a human can do. In the beginning of the process the results may not fulfil the expectations, but with the time an agent can optimise his behaviour himself, according to the rewards that he gets, and perform better till the maximum reward is reached. But all methods may deliver different results by solving different tasks with reinforcement learning. These results may reach maximal reward, or they can be insufficient, and an agent will not be able to solve the task even after learning. This paper is giving more information about existing algorithms and methods that can be used to solve tasks with Reinforcement Learning approach. The paper is giving a better understanding of presented methods and also tells about the performance of learning that can be achieved using the described methods.

This paper presents a comparison of some deep learning models with different mechanisms, that helps to improve the learning process and add to it more performance and stability. Also, it gives an overview of how these approaches are working and shows for what reasons these approaches are useful. It shows which combination of deep learning architecture and mechanism can deliver better result during the

learning process. Two different architectures: Double Deep Q-Learning and Dueling Double Deep Q-Learning were chosen to be compared and presented in this paper. Additionally, mechanisms like Experienced Replay, Prioritised Experience Replay and adding of noisy layers were used to research how they affect the performance and stability of learning. Furthermore, one more deep learning model, named Deep Q Learning, will be presented. This model is presented only to give further understanding of the other two models. Tests with this model will not be presented in the evaluation part of the paper. Two different environments were taken to perform tests and to compare results between different approaches. One of the environments is an Atari game named Pong. This environment is giving images as input to the networks, and they use information in these images to perform action that can lead to increasing of the score. The second environment is CartPole. This environment is different from the first one and gives raw data instead of images as an input to the networks, the actions will be then performed after analysing of this data.

In the next chapter, the theoretical basics of all deep models and mechanisms that were used in this paper will be presented, along with the advantages and features of these mechanisms. It serves also to give a common understanding of the methods that were presented in this paper. The chapter 3 gives more information about used environments and explains which hyperparameters were used to perform tests. It also tells for what reason these parameters are used. The chapter 4 shows the results of the performed tests and compares them between each other to find out which architecture and combination of mechanisms is the best for given environments. All results of the learning process were visualised in the graphs. The chapter 5 gives a short conclusion where all observations were put together and discussed.

2 Theoretical basics

All used algorithms and improvement features are explained in this section. It starts with a basic Q-Learning Algorithm and an explanation of the improvement technics like Experienced Replay and Prioritized Experience Replay (PER). Next parts of this section are about three different variants of Q-Learning Algorithm usage.

2.1 Q-Learning

It is considered a task in which an agent interacts with an environment ε, At each time-step the agent selects an action a_t from the set of legal game actions, $A = 1, ..., K$. The action is passed to the emulator and modifies its internal state and the score. An emulator is used for training and evaluation to simulate the environment in which the agent will be trained. The agent observes the environment after each action and may get a reward r_t depending on the state, where it will land after taking an action.

Since it is impossible to fully understand the situation every time after each step, the sequences of observations will be created $s_t = x_1, a_1; x_2, a_2; x_t, a_t$ where x_t is agent's current observation and a_t is an action that was made. Strategies to solve problems will be learned upon these sequences.

All sequences in the emulator are assumed to terminate in a finite number of time-steps. This formalism gives rise to a large but finite Markov Decision Process (MDP) in which each sequence is a distinct state. As a result, standard reinforcement learning methods for MDPs can be applied, simply by using the complete sequence s_t as the state representation at time t [1].

The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes future rewards. The future rewards are discounted by a factor of γ per time-step, and they define the

future discounted return at time t as:

$$R_t = \sum_{t'=t}^{T} = \gamma^{t'-t} * r_{t'} \tag{1}$$

where T is the time-step in which the environment reaches terminate state. The optimal action-value function can be defined $Q^*(s, a)$ as the maximum expected return achievable by following any strategy, after seeing some sequences and then taking some action a:

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

where π is a policy mapping sequences to actions. The optimal action-value function obeys an important identity known as the *Bellman equation*. This is based on the following intuition: if the optimal value $Q^*(s', a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximizing the expected value of $r + \gamma * Q^*(s', a')$

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon}[r + \gamma * \max_{a'} Q^(s', a') | s, a] \tag{3}$$

The idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update. Such value iteration algorithms converge to the optimal action-value function [2]. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator to estimate the action-value function. Mostly the linear function approximator will be used, but it is possible also to use the non-linear approximator instead. For example, a neuronal network. The network with weights Θ in case of Q-Learning is called Q-Network.

A Q-network can be trained by minimising a sequence of loss functions $L_i(\Theta_i)$ that changes at each iteration i,

$$L_i(\Theta_i) = \mathbb{E}_{s,a \sim \rho(.)}[(y_i - Q(s, a; \Theta_i))^2] \tag{4}$$

where $y_i = E_{s' \sim \epsilon}[r + \gamma * max_{a'} Q(s', a'; \Theta_{i-1}) | s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the behaviour distribution. The parameters from the previous iteration Θ_{i-1} are held fixed when optimising the loss function $Li(\Theta_i)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before the learning begins.

If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator E respectively, then we arrive at the *Q-learning algorithm* [3]. Note that this algorithm is **model-free**: it solves the reinforcement learning task directly using samples from the emulator ε, without explicitly constructing an estimate of ε. It is also off-policy: it learns about the greedy strategy $a = \max_a Q(s, a; \Theta)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with the probability 1 - ϵ and selects a random action with probability ϵ [1].

2.1.1 Experience Replay

Experienced Replay is a technic that is used to improve a stability of learning for Reinforcement Learning Algorithms. Within this technic, the learning agent remembers its experiences and repeatedly presents them to its learning algorithm as if the agent experienced again and again what it did earlier. It provides the following improvement points:

1. the process of propagation is sped up

2. an agent would get a chance to refresh what it has learned before

During the network training, if an input pattern has not been presented for quite a while, the network typically will forget what it has learned for that pattern and thus need to re-learn it when that pattern is seen again later. This problem is called the **re-learning problem**. Experienced Replay is useful to overcome exactly this situation, where the agent forgot about its previous experience [4]. To implement this kind of improvement, it is needed to store the agent's experience at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$in a dataset $\mathcal{D} = e_1, ..., e_N$, pooled over many episodes into a **replay memory**. During an inner loop of the algorithm, the Q-learning updates will be applied, drawn at random from the pool of stored samples. After the experience replay, the agent selects and executes an action according to the ϵ-greedy policy. Since the usage of histories of huge length as inputs to a neural network can be difficult, Q-function can work with fixed length representation of histories. For example, an algorithm can do the Q-learning updates with mini batches taken randomly from the whole learning history.

2.1.2 Prioritized Experience Replay

Another way to improve the learning performance of the RL Algorithm that is based on saving experience is a Prioritized Experience Replay or PER [5]. Prioritizing of experience can make the replay more efficient and effective compared to the case when all transitions are replayed uniformly. The key idea is that an RL agent can learn more effectively from some transitions than from the others. Transitions may be more or less surprising, redundant, or task-relevant. Some transitions may not be immediately useful to the agent, but might become so, when the agent competence increases [6]. Experience replay liberates online learning agents from processing transitions in the exact order they are experienced. Prioritized replay further liberates agents from considering transitions with the same frequency that they experienced. In this approach, the transitions with high expected learning progress will be replayed more frequently. It will be measured by the magnitude of temporal-difference (TD) error.

The central component of prioritized replay is the criterion by which the importance of each transition is measured. One idealized criterion would be the amount the RL agent can learn from a transition in its current state (expected learning progress). While this measure is not directly accessible, a reasonable proxy is the magnitude of a transition's TD error δ, which indicates how 'surprising' or unexpected the transition is. This is particularly suitable for incremental, online RL algorithms, such as Q-learning, that already compute the TD-error and update the parameters in proportion to δ. The value of δ will be updated for every sample from the Replay Memory that was returned from the minibatch. So it is not necessary to update δ values for all samples to achieve the successful learning. The probability of sampling transition i can be computed with this equation:

$$P(i) = \frac{p_i^\alpha}{\sum_k p} \tag{5}$$

where $p_i > 0$ is a priority of transition i. The exponent α means the importance of prioritization, if $\alpha = 0$ then prioritization is not applied and the uniform case will be used. The term p_i can be computed in two different ways. The first way is directly with $p_i = |\delta_i| + \eta$, where η is a positive constant that prevents, that a transition is not being revisited once its error is zero. The second variant of computing p_i is the rank-based prioritization, where $p_i = \frac{1}{rank(i)}$ where $rank(i)$ is the rank of transition i when the replay memory is sorted according to $|\delta_i|$.

Prioritized replay introduces bias, because it changes the distribution in an uncontrolled manner, and therefore changes the solution that the estimates will converge to (even if the policy and state

distribution are fixed). This bias can be corrected by using importance-sampling (IS) weights [5].

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^{\beta} \tag{6}$$

where N is a size of the replay buffer, and β is an exponent that corrects the bias. If $\beta = 1$ then the bias will be fully compensated for not-uniform probability $P(i)$. During the learning process, the value of β should be annealed from the initial point to the value of 1. The value of w_i will be folded into the Q-learning update by using $w_i\delta_i$ instead of δ_i

2.2 Image Preprocessing

For the experiments with Atari Game "Pong" gym environment "PongNoFrameSkip-v4" will be used. This environment has images as data that will be used for training. Before the images can be fed to the network, a small preprocessing is required.

Working directly with raw Atari frames, which are 210 × 160 pixel images with a 128 colour palette, can be computationally demanding, so we apply a basic preprocessing step aimed at reducing the input dimensionality. The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to a 110×84 image. The final input representation is obtained by cropping an 84 × 84 region of the image that roughly captures the playing area. To increase learning speed and stability, the input to the Q-function will be created with preprocessing of the last four frames of a history and stacking them together.

2.3 DQN

Now, the architecture that is used for training of an agent with Pong environment will be described. The input to the neural network consists of an 84 × 84 × 4 image produced after preprocessing. The next convolutional layer convolves 32 8 × 8 filters with stride 4 with the input image. The last convolutional layer convolves 64 4 × 4 filters with stride 1. Each of these layers is created with ReLu as activation function. The first fully-connected layer takes as input channels from the convolutional layers and returns 512 features. This layer also has an activation function ReLu. The output layer is a fully connected linear layer with a single output for each valid action.

For the training with this approach two networks with the same architecture will be created, one is a local network that will be used during training for computation and a target network. The target network will be the result network in the end of training. So it means that during the training, all weights from local network will be copied to the target network from time to time. The periodicity of this update can be set by hyperparameters. The term to update a Q-Value for a target network in the DQN approach looks as follows:

$$Q(s,a) \leftarrow Q(s,a) + lr * [r_{t+1} + \gamma \cdot \max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{7}$$

where lr is a learning rate set wit hyperparameters and t is a time frame. Here are listed the steps involved in a deep Q-network (DQN)[7]:

1. Preprocess and feed the game screen (state s) to DQN, which will return the Q-values of all possible actions in the state.
2. Select an action using the epsilon-greedy policy. With the probability epsilon, we select a random action a and with probability 1-epsilon, we select an action that has a maximum Q-value, such as a = argmax(Q(s,a,w)).

3. Perform this action in a state s and move to a new state s' to receive a reward. This state s' is the preprocessed image of the next game screen. This transition will be stored in the replay buffer.
4. Sample some random batches of transitions from the replay buffer and calculate the loss: the squared difference between target Q and local Q.
5. Perform gradient descent with respect to actual network parameters in order to minimize this loss.
6. After every C iterations, copy actual network weights to the target network weights.
7. Repeat these steps for M number of episodes.

2.4 Double DQN

The second learning approach that was used to train an agent is Double DQN [8]. This approach is very similar to the DQN approach that was described above, but there is one difference that makes this approach more stable for the learning process. The max operator, in the standard Q-Learning and DQN, uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, the selection and evaluation of the value can be decoupled. It means that the difference between DQN and DDQN is a computation of Q-Value. Since this approach also uses two networks with the same architecture, both of them will be used to update a Q-Value in DDQN. The term to update Q-Value in DDQN looks as follows:

$$Q(s,a) \leftarrow Q(s,a) + lr * [r_{t+1} + \gamma \cdot \max_a Q'(s_{t+1},a) - Q(s_t,a_t)] \tag{8}$$

where $\max_a Q'(s_{t+1},a)$ is an action with the highest Q Value computed in the local network. The process of learning with DDQN is the same as the process that was described for DQN. The Q-Value update is the only difference. Only with this small change, DDQN improves over DQN both in terms of value accuracy and in terms of policy quality. Double DQN helps us to reduce the overestimation of q values and, as a consequence, helps us to train faster and to have more stable learning. It will be shown in the graphs in the evaluation chapter. The architecture of Double Deep Q Learning model is the same as was presented in the previous chapter for DQN.

2.5 Dueling DQN

This variant of Reinforcement Learning requires changes not only in the computation of Q-Value functions, but also changes in the architecture of the neuronal network. This approach is called Dueling DQN [9]. Since Q-Values correspond to how good it is to be at that state and taking an action at that state (Q(s,a)), this term can be decomposed as the sum of:

1. V(s): the value of being at that state
2. A(s,a) the advantage of taking that action at that state (how much better is it to take this action versus all other possible actions at that state).

Dueling DQN is needed to separate the estimator of these two elements using new streams: one that estimates the state value V(s) and the other one that estimates the advantage for each action A(s,a). In the architecture, the last fully-connected layer will be divided into two streams to estimate state-value and the advantages for each action.

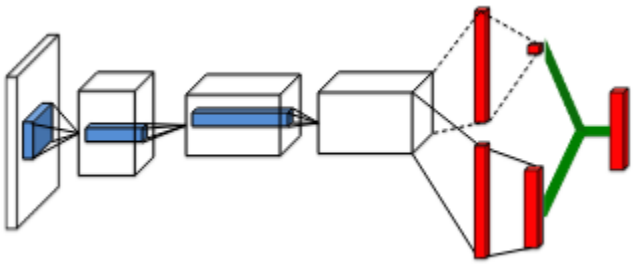

Figure 1: Dueling DQN architecture

After changes in architecture parameters of separated layers can be donated with α and β. The parameters of convolutional layers are donated as θ. With this knowledge, the Q-Value for Dueling DQN is defined as:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \tag{9}$$

This equation is unidentifiable in the sense that given Q we cannot recover V and A uniquely. To see this, add a constant to $V(s;\theta,\beta)$ and subtract the same constant from $A(s,a;\theta,\alpha)$. This constant cancels out resulting in the same Q value. This lack of identifiability is mirrored by poor practical performance when this equation is used directly. To overcome this problem, the other approach of Q-Value computation can be used. This approach computes Q-Value using the mean of all estimated values of the advantage function:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a\prime} A(s,a\prime;\theta,\alpha) \tag{10}$$

On the one hand it loses the original semantics of V and A because they are now off-target by a constant, but on the other hand it increases the stability of the optimization: with (10) the advantages only need to change as fast as the mean, instead of having to compensate any change to the optimal action's advantage. By decoupling the estimation, the network can learn which states are (or are not) valuable without having to learn the effect of each action at each state (since it's also calculating V(s)). This architecture helps to accelerate the training. It can calculate the value of a state without calculating the $Q(s,a)$ for each action at that state. And it can help to find much more reliable Q values for each action by decoupling the estimation between two streams [10].

2.6 Architecture with noisy weights

In this approach the Q-Values calculations will remain the same for each architecture as introduced in the previous sections, this approach uses Noisy neuronal network parameters to create an architecture and train an agent. The name of created architectures are NoisyNets [11]. NoisyNets are neural networks whose weights and biases are perturbed by a parametric function of the noise. These parameters are adapted with gradient descent. Noisy parameters can be defined as follows:

$$\theta \overset{\text{def}}{=\!=} \mu + \Sigma \odot \epsilon \tag{11}$$

where $\zeta \overset{\text{def}}{=\!=} (\mu, \Sigma)$ is a set of vectors of learnable parameters, ϵ is a vector of zero-mean noise and $\odot$ represents element-wise multiplication. Optimisation now occurs with respect to the set of parameters ζ.

A noisy network agent samples a new set of parameters after every step of optimisation. Between optimisation steps, the agent acts according to a fixed set of parameters (weights and biases). This

ensures that the agent always acts according to parameters that are drawn from the current noise distribution. After applying of these noisy layers to DQN or Dueling DQN ϵ-greedy exploration will not be used any more, instead the policy greedily optimises the action-value function. Secondly, the fully connected layers of the value network are parameterised as a noisy network, where the parameters are drawn from the noisy network parameter distribution after every replay step. Since DQN and Dueling take one step of optimisation for every action step, the noisy network parameters are re-sampled before every action. When replacing the linear layers by noisy layers in the network (respectively in the target network), the parameterised action-value function $Q(x, a, \epsilon; \zeta)$ (respectively $Q(x, a, \epsilon'; \zeta^)$) can be seen as a random variable and the DQN loss becomes the NoisyNet-DQN loss.

$$L(\zeta) = \mathbb{E}[\mathbb{E}_{(s,a,r,y)\sim D}[r + \gamma \max_{b \in A} Q(y, b, \epsilon'; \zeta^-) - Q(x, a, \epsilon; \zeta)]^2] \tag{12}$$

where the outer expectation is with respect to the distribution of the noise variables ϵ for the noisy value function $Q(x, a, \epsilon; \zeta)$ and the noise variable $\epsilon\prime$ for the noisy target value function $Q(y, b, \epsilon'; \zeta^-)$ [11]. The loss function for Dueling DQN will look as follows:

$$L(\zeta) = \mathbb{E}[\mathbb{E}_{(s,a,r,y)\sim D}[r + \gamma * arg \max_{b \in A} Q(y, b^*(y), \epsilon'; \zeta^-) - Q(x, a, \epsilon; \zeta)]^2] \tag{13}$$

$$b^*(y) = arg \max_{b \in A} Q(y, b(y), \epsilon''; \zeta) \tag{14}$$

At a high level, this algorithm is a randomised value function, where the functional form is a neural network. Randomised value functions provide the provably efficient means of exploration [12]. Although the improvements in performance might also come from the optimisation aspect since the cost functions are modified, the uncertainty in the parameters of the networks introduced by NoisyNet is the only exploration mechanism of the method. Having weights with greater uncertainty introduces more variability into the decisions made by the policy, which has the potential for exploratory actions.

3 Test and environment creation

In this section, all made tests and used environments will be described. The test was done for each of the described architectures with Experienced Replay. Also, the tests were done for the same architectures, but with using of PER or Noisy Layers. The implementation of the architectures and other mechanisms that were used for the tests can be found here [13]. The tests were done in the ICC Server of HAW Hamburg, the learning was done with CUDA 11.2 and GPU NVIDIA V100 16GB HBMS. The architectures were tested with two different environments. Both of them were taken from the framework **gym** [14]. Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games. Two environments were taken for tests. The first environment is using images as input data, and simulates an Atari game **Pong**. Pong is a table tennis–themed arcade sports video game, featuring simple two-dimensional graphics. The game is going until one of the participants reaches a score of 21 points, which is a goal for an agent that will be trained. Different version of this environment are existing, the one that was used for the tests is **PongNoFrameskip-v4**. The second environment simulates a CartPole [15]. A pole is attached by an unactuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. One more difference of these two environments is that Pong is a game and works based on game frames. These frames will be sent as images to the input layer of the neuronal networks. CartPole is working based on raw data, that contains useful information, that shows how the environment reacts on the done actions. In this case, raw data will be set as input of networks. The neuronal networks for Pong environment will contain convolutional layers to process given images and for CartPole these convolutional layers are not needed, so for this environment only danse layers will be used.

The test was done with the same hyperparameters for each environment. In total, there were six parameters to set. For Pong environment, the following parameters were chosen. Number of processed frames was set to 1,000,000. This is a number of frames that will be analysed during a training process. The size of replayed memory will be set to 10,000, it shows an amount of results that will be saved to replay buffer and used for Experienced Replay and PER. Learning rate will be set to 0.0001. For ϵ-greedy process will be started with 1 and annealed to minimum of 0.01. The last parameter shows for how many frames the replay buffer will be filled based on a random policy, before agent-env-interaction. That means for this amount of frames, the agent will not actually learn from the environment. For all tests with Pong environment, this value is 10,000. Multiple tests were done for both environments. The tests do not have same results, different seeds will be used. Seed is also one of the parameters that will be set for the training. Tests with different seeds will show more and better observations how the agent will learn and solve given problems. Training an agent for Pong environment takes a lot of time until 1,000,000 frames are processed. For this reason, first a model was successfully trained and tests were done with using of transfer learning approach that made tests complete 10-20 % quicker than a fully new training takes. All parameters in convolutional layers were frozen and remaining dense layers were reset. With this configuration, the model will change parameters only for dense layers and will not do calculation in terms of feature detection in convolutional layers. CartPole environment does not need images to be processed during the training. This environment does not use any convolutional layers, and here none of the parameters needs to be frozen. The parameters that were used for tests with CartPole will be presented later.

4 Evaluation

In this section, the results of the tests will be shown. The architectures will be compared in terms of reached average goals, computation time and learning effectiveness. The first six chapters present results for Pong environment, from the Chapter 4.7 tests with CartPole environment will be described.

4.1 DDQN with Pong environment

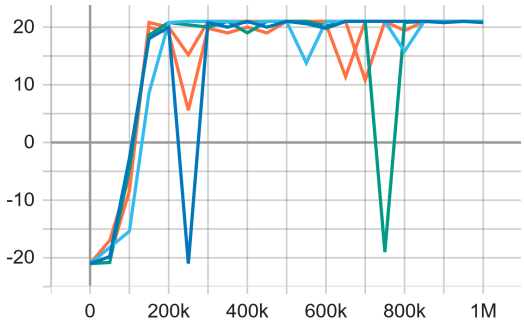

Figure 2: Rewards during learning for five tests

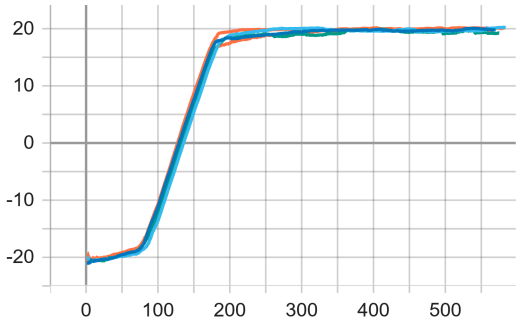

Figure 3: Average result during the learning process

Figure 4: Results for Pong with DDQN

In the graph 4 are shown the results of the learning with DDQN agent and Experienced Replay buffer. Two graphs represent different data with five tests. All tests were done with different seeds from 1 to 5. Different tests are represented with different colour in the graphs. The left graph is showing the rewards that the agent got during the training for each frame. The right graph shows

an average of these rewards for the made steps. The learning process took 251 minutes in average. The average score that the agent received during all tests is 19.50. In the early phase of learning, the scores are not improving very well. It can be caused by not enough experience that agent has in these steps of learning. After 70-80 steps, the agent gets a big improvement in terms of actual scores and average results. The improvement can be caused by experience replay mechanism that was used as an optimisation possibility. Since the agent reuses earlier experiences, it can choose better action and this improves its performance. And after almost 100 steps agent reaches the point of maximal reward of 21 and in average holds rewards between 18-21 till the end of the learning. In the graph 2 the decrease of the score for some frames in tests with seed 2 and 4 can be recognised. It can be caused by small "blackouts" that an agent can get during the training processes. Since Experienced Replay buffer was used, it happened only twice during the training, only in two tests, and it didn't have a lot of influence on the average scores and the training itself. It was the first architecture that was tested, in the next section this architecture will be compared with other architectures to find the best one among them.

4.2 Dueling DQN with Pong Environment

In this section, the results of Dueling DQN with Pong environment will be shown. Same as DDQN network, five tests with different seeds from 1 to 5 will be done.

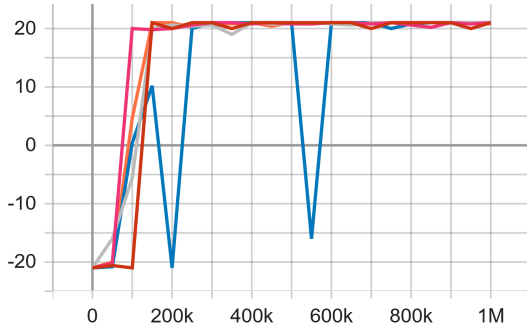

Figure 5: Rewards for Dueling DQN during learning for five tests

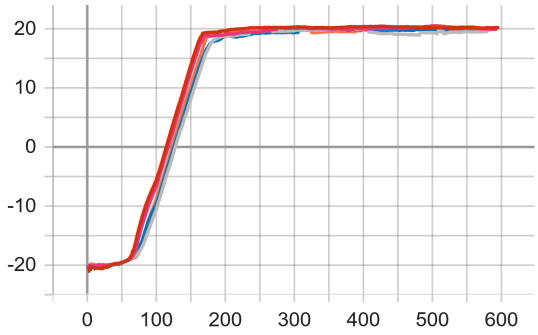

Figure 6: Average rewards for Dueling DQN result during learning process

Figure 7: Results for Pong with Dueling DQN

In the graphs 7 the average and actual reward results for the Dueling DQN architecture are shown. The results are shown after completing of five tests with different seeds from 1 to 5. Same as in the previous tests, the agent was successfully trained to play Pong Atari game. But the average score in this training got to the point of 20.2. It is 5% bigger than the agent that was trained with DDQN approach. Dueling DQN shows not massive improvement in the average score, but on the graphs it can be seen that the maximum point will be reached quicker than by DDQN. Since all tests were done with fixed hyperparameters, every time 1,000,000 frames were processed before the learning is done. Dueling DQN needed 390 minutes in average to reach this point. It is 140 minutes longer than for DDQN, but due to faster convergence of Dueling DQN Algorithm, fewer frames can be processed to reach the needed score. It can be recognised that also this architecture gets some "blackouts" during the training with experience replay buffer. According to the graph 5 it happened two times and only in one of the five tests. Even with these blackouts, Dueling DQN shows a more stable learning process compared to the DDQN approach. The graphs, where average scores of DDQN and Dueling DQN are compared, can be found in the figure 8.

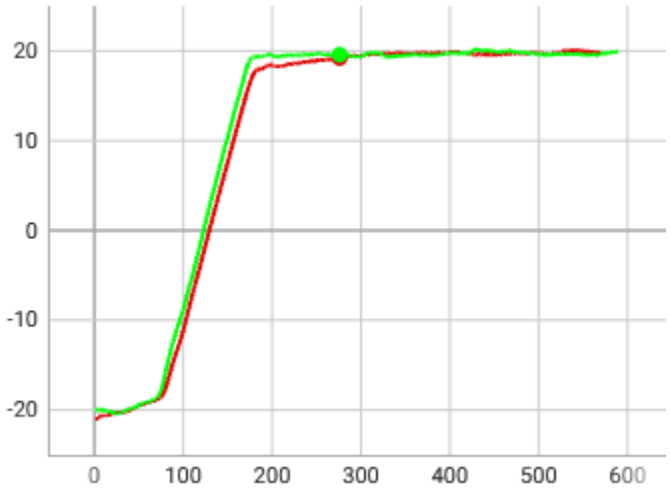

Figure 8: Comparison between DDQN and DQN. DQN red line; DDQN green line

From this graph, it can be seen that DDQN shows slightly faster convergence to the maximum point than DQN. The advantage of the Dueling architecture lies partly in its ability to learn the state-value function efficiently. With every update of the Q values in the Dueling architecture, the value stream V is updated – it contrasts with the updates in a single-stream architecture (DDQN) where only the value for one of the actions is updated, the values for all other actions remain untouched. This more frequent updating of the value stream in this approach allocates more resources to V, and thus allows for better approximation of the state values, which in turn need to be accurate for temporal-difference-based methods like Q-learning to work. Since an action space of Pong environment is small compared to some other Atari games, the improvement in these tests is only 5%. The importance of this approach is growing for environments with a complex environment and bigger action space.

4.3 DDQN with noisy weights with Pong Environment

This section presents results for agents that were learned with DDQN with noisy weights. They are presented in the graph 11:

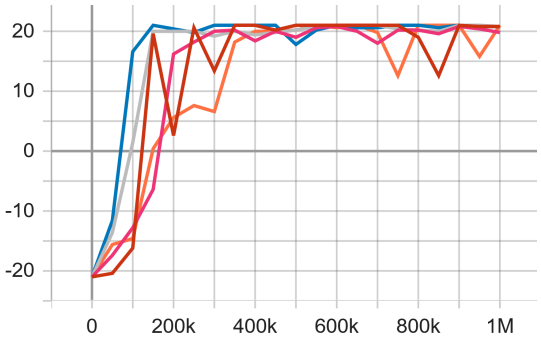

Figure 9: Rewards for noisy DDQN during learning for five tests

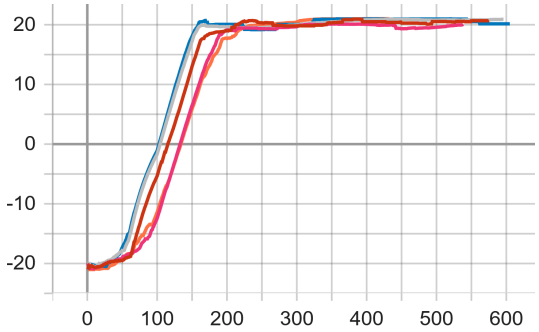

Figure 10: Average rewards for noisy DDQN result during learning process

Figure 11: Results for Pong with noisy DDQN

In average this approach is getting a reward of 20.5 that is almost same as Dueling DQN is getting and is better than normal DQN approach. Noisy layer architecture has a difference in the actual reward graph 9. It is noticeable that the agent is converting to the good results almost in the beginning of training process. Since this approach is using noisy weights that will be adjusted all the time during the training, it seems to be unstable from time to time and has more "blackouts" during the training. Even this instability is not effecting the end result in average and the agent can be also trained successfully. The improvements in performance might also come from the optimisation aspect since the cost functions are modified, the uncertainty in the parameters of the networks introduced by NoisyNet is the only exploration mechanism of the method. Having weights with greater uncertainty introduces more variability into the decisions made by the policy, which has potential for exploratory actions. In case of Pong, these improvements are not giving any big advantage compared to Dueling DQN. NoisyNet has faster processing of frames because in average this approach takes only 300 minutes to be done with training. It happens because this architecture does not split the last layer, as it is done with Dueling DQN. That is why the value function will get updated only for the action that was taken, same as in the normal DDQN approach. It requires less time of processing a single frame. Same as for Dueling DQN this approach can show more improvements with more complicated environments with bigger action spaces.

4.4 DDQN with Pong environment and PER

In this section, the agent will be trained with Double DQN architecture with Prioritized Experience Replay buffer (PER). It will be investigated if PER gives some improvements to the learning process compared to basic experience replay.

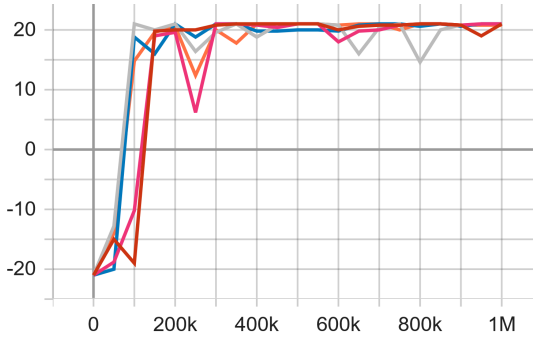

Figure 12: Rewards for DDQN with PER during learning for five tests

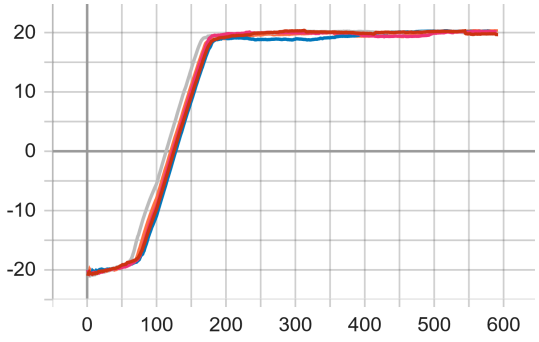

Figure 13: Average rewards for DDQN with PER result during learning process

Figure 14: Results for Pong with DDQN and PER

This approach also ended with successfully reached goal of getting 21 points. In average, an agent trained with DDQN and PER gets a score of 20.2 out of 21 (see graph 13). From the graph 12 can be recognised that some "blackouts" still happened even with using of PER mechanism, but all of them are not going to negative values of the possible score. It can be explained with a careful choice of experience that will be gathered during the training. Since these experience samples from the buffer are now prioritised, the agent takes the most valuable ones to use in the certain point of the training and minimises a possibility to gain negative rewards. Due to needed computation of priorities this approach needs in average 275 minutes to be completed for 1,000,000 frames.

The graph 17 shows a comparison between DDQN with Experienced Replay and DDQN with PER.

For reasons of clarity, the figures show only one test for each approach.

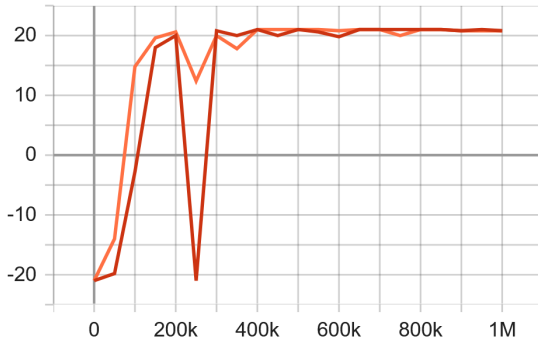

Figure 15: Rewards for DDQN with PER vs Experienced Replay during learning

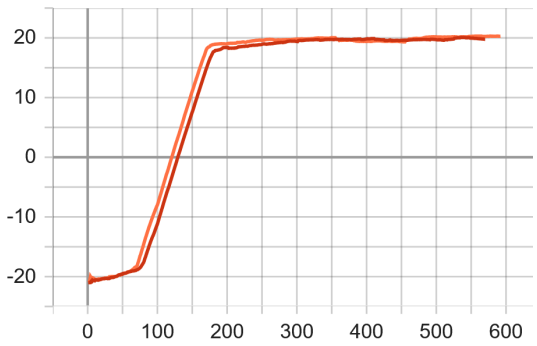

Figure 16: Average results for DDQN PER vs Experienced Replay

Figure 17: DDQN PER vs Experienced Replay for Pong environment

The training in average score with PER had slightly small improvements compared to the Experienced Replay approach. Since Pong environment has a finite score of 21, as soon as agent gains this score, the average between PER and Experienced Replay approach will be normalised. More can be seen in the graph 15 where one "blackout" happened. It shows that the score by the "blackout" with PER approach is not going to the negative area, but Experienced Replay approach got in this case a minimum score of -21. Both of these graphs show how PER improves learning compared to the Experienced Replay approach. One more thing that will be also improved by PER is a loss function, the graph 18 shows the loss during the learning process with PER and the graph 19 shows the loss with basic Experienced Replay. It shows that PER approach gives an agent more stability as well in terms of loss. Loss function values for PER approach do not have such rapid changes as they have with the normal Experienced Replay. PER chooses the most prioritised experience from the whole buffer and uses it in a particular timeframe. Since it is not random as in basic Experienced Replay approach, the loss becomes more stable and is decreasing the whole way to the end of training process. In the beginning of the training, it is already noticeable that PER has better development of loss function values than experience replay.

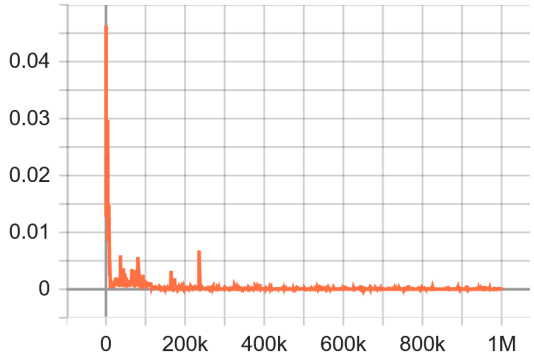

Figure 18: Loss for DDQN with PER

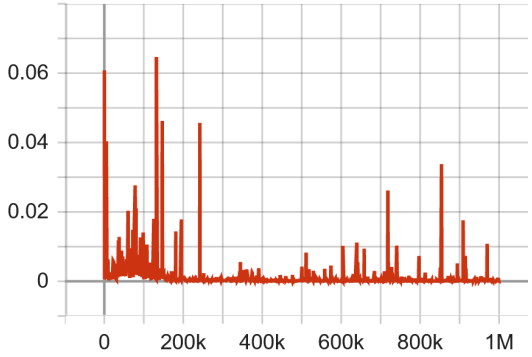

Figure 19: Loss for DDQN with Experienced Replay

Figure 20: Loss DDQN PER vs Experienced Replay for Pong environment

In the graph 21 an average result per step for all approaches with Double Deep Q-Learning is shown.

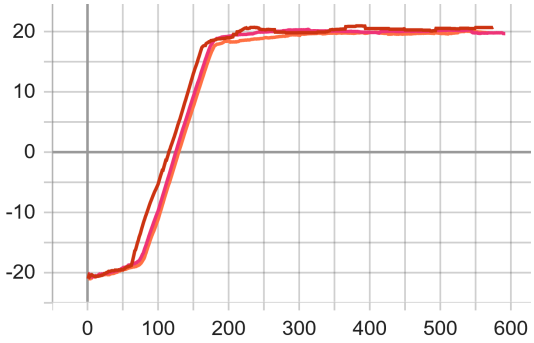

Figure 21: All DDQN approaches for Pong environment

Red curve shows performance of DDQN with noisy layers, pink curve shows DDQN with PER and orange is DDQN with normal experience replay. All tests have relatively equal behaviour, in terms of steps that were taken and the results. The difference between them is only the computation time that was needed to complete the training. The best results show DDQN with Noisy Layers, but also this approach took the most time by learning.

4.5 Dual DQN with PER for Pong environment

This section shows the test results of Dueling DQN with prioritized Experienced Replay approach. Same as for other architectures, five tests with different seeds were done to show the improvement of an agent during the learning process.

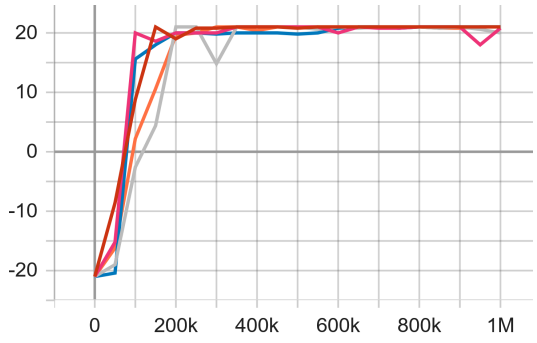

Figure 22: Dual DQN with PER rewards for five tests

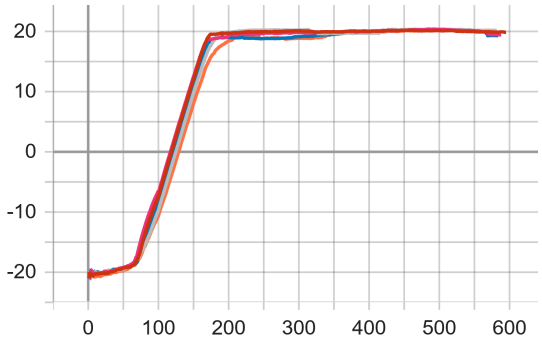

Figure 23: Dual DQN with PER Average rewards during the process

Figure 24: Dual DQN with PER for Pong environment

In the graph 23 an average reward per step that agent gets during the learning is shown. Different colours in the graph represent different seeds with which an agent will be trained. The average duration of this training is around 365 minutes. The figure 22 shows the actual reward that an agent got per frames that were processed. It shows that the training had in the beginning rather unstable behaviour, but after processing of 200,000 frames it became more stable and shown the maximal result of 21 almost all the way to the end of the training. This gives an average score of 20.4-20.6 points from 21 possible. It means that PER improves in average the training results compared to the basic experience replay approach. From the figure 22 can be seen that there were no "blackouts" during the training. This behaviour was also seen in previous tests. The figure also shows that the increase of scores is happening almost immediately after the start of learning, and as soon as 150,000 frames are processed it reaches a maximum reward point (see 23).

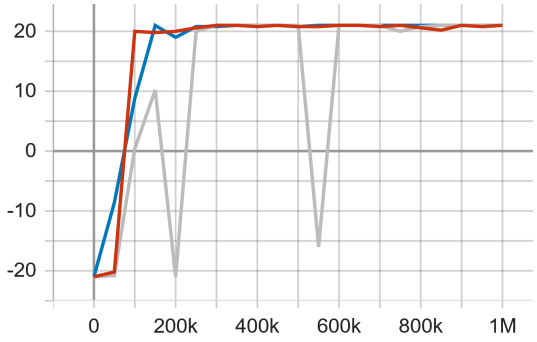

Figure 25: Dual DQN vs Dual DQN with PER rewards during the process per processed frame

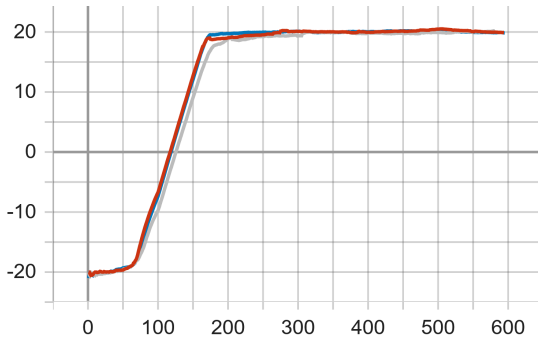

Figure 26: Dual DQN vs Dual DQN with PER average reward during the process per step

Figure 27: Dual DQN vs Dual DQN with PER for Pong environment

In the figures 26 and 25 a comparison between the Dual DQN architecture with PER and Dual DQN with Experienced Replay is shown. To make the graphs more readable, not all tests were taken for the comparison. From the graph 26, the blue curve shows the training process for an agent with PER. It can be seen that using PER gives small improvement for the learning, and makes the agent getting

to the maximal reward faster. The red curve in the graph represents a training without PER and shows almost the same result as a training with the PER approach. The gray curve represents also the training without PER. The figure 25 shows comparison between PER and non PER approach per processed frames. It can be seen that the gray curve gets some "blackouts" during the training. On the other hand, the blue curve that represents learning with PER does not have them at all. According to this observation, it can be said that PER helps an agent to avoid this specific behaviour. That guarantees a more stable learning process.

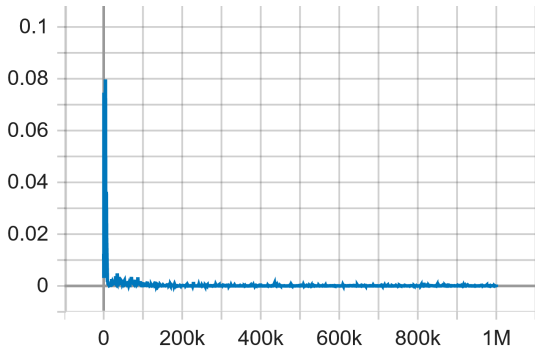

Figure 28: Loss for Dual DQN with PER

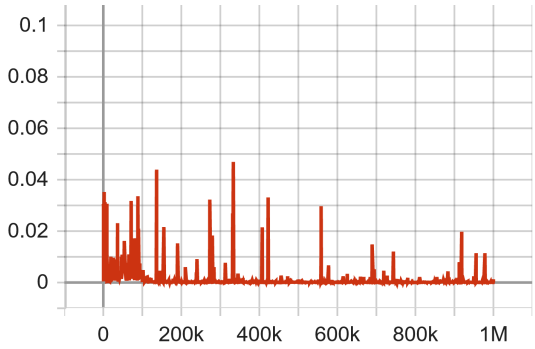

Figure 29: Loss for Dual DQN with Experienced Replay

Figure 30: Loss Dual DQN PER vs Experienced Replay for Pong environment

The figures 28 and 29 show the development of loss function during the training process with PER and with basic Experience Replay. Same as in the previous tests with DDQN, the loss development is more stable with an agent that uses PER. Already in the beginning, the loss will decrease and will remain in the interval between 0 and 0.0001. For the agent that was trained with basic experience, replay loss is jumping from 0.05 to 0.0001 during the whole training process. Since Pong environment has a fix score that can be achieved, the improvements, that PER is giving in average for Dual DQN architecture, do not speed up the learning, but they can help to stabilize the process and get rid of "blackouts".

4.6 Dual DQN with Noisy Layers for Pong environment

In this section will be shown how Noisy Layers effect the Dual DQN architecture. For this agent only tree tests were done, the reason is the long training process time. For each test, training with this agent took in average 500 minutes.

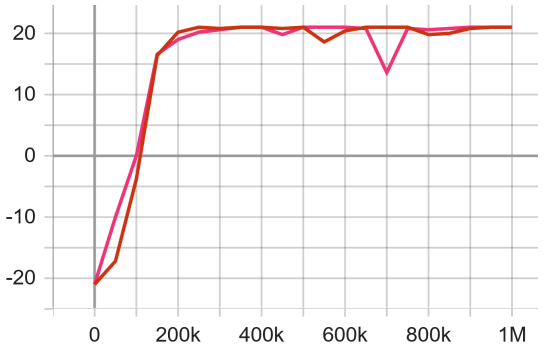

Figure 31: Rewards per processed frames with Noisy Dueling DQN for Pong environment

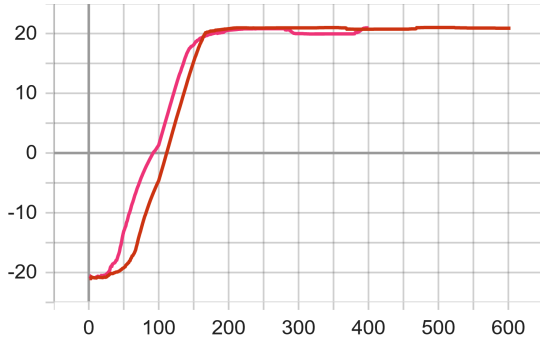

Figure 32: Average results with Noisy Dueling DQN for Pong environment

Figure 33: Noisy Dueling DQN for Pong environment

In the figure 33 the training results for an agent trained with noisy Dual DQN architecture for Pong environment are shown. This is the only environment that for one of the tests reached a maximal score of 21 points in average. In the graph 31 can be recognised that already after 100,000 processed frames an agent reaches a positive score in average. In general, the average reward that an agent got during the training among all tests is 20.9. It shows that a Dueling DQN with Noisy Layer shows the best performance in terms of gaining reward for Pong environment. Also, there are less "blackouts" that happened with agent during the training compared to normal Dueling DQN approach.

In the graph 36 a comparison of training processes for Dual DQN, Dual DQN with PER and Dual DQN with noisy layers is shown. Three tests were presented. The blue graph shows a training with Dueling DQN and Noisy layers, pink is Dueling DQN with PER and red is a basic Dueling DQN environment. Figure 35 shows that Dueling DQN with Noisy Layers is showing the best performance and reaches maximal reward faster than others. On the other hand, this approach took more time than the two others. Next observation is that Dueling DQN with Noisy layers need only 400 steps to process 1,000,000 frames compared to other approaches that took almost 600 steps to be done. That means, even with computing new noisy weights after every step, Noisy Network can process more frames per step than Dueling DQN with PER and without PER. In the figure 34 it can be seen that for Noisy and PER approaches the "blackouts" are not happening during training.

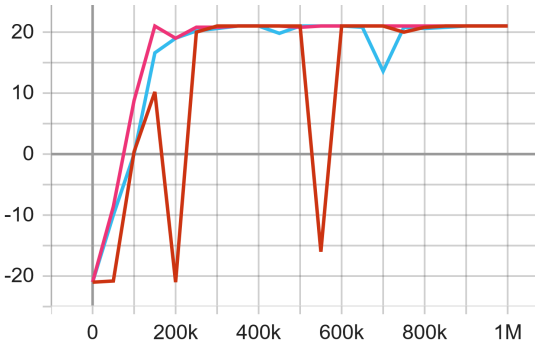

Figure 34: Rewards per processed frames with different Dueling DQN approaches for Pong environment

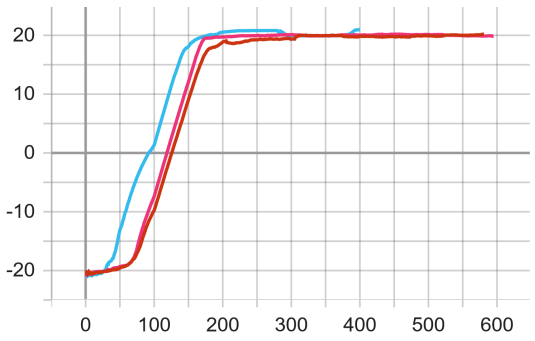

Figure 35: Average results with different Dueling DQN approaches for Pong environment

Figure 36: Different Dueling DQN approaches for Pong environment

4.7 Double Deep Q Learning Model with CartPole Environment

In this section and in the next sections, the tests that were done with presented architectures with Cartpole Environment will be described. This environment does not use any images for training, the training is done with raw data from Observation Space. All architectures do not need to have convolutional layers to be able to train agents. This section presents tests for Double Deep Q-Learning architecture. Same as for Pong environment, multiple tests with different seeds were done to get better result for analysing. The parameters for all tests are the same, only the seed will be changed from tests to test. The size of the replay memory is 35,000, same as the number of processed frames and the size of fill buffer. The learning rate is 0.0001. Minimal epsilon is 0.01 and epsilon decay process is finished in 1,000 frames. These parameters were set, because they give the best performance of learning among all other tests that were done to calculate them.

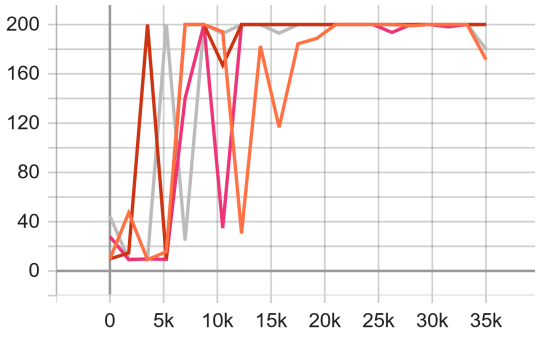

Figure 37: Rewards for DDQN with CartPole environment

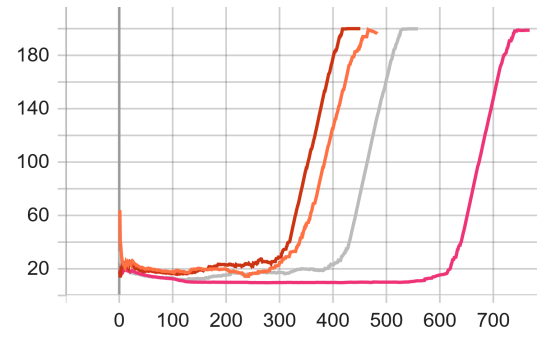

Figure 38: Average results for DDQN with CartPole environment

Figure 39: DDQN with CartPole

In the figures 37 and 37 the results of the training with Double Deep Q-Learning and environment CartPole are presented. In the figure 37 can be seen that the process has a certain instability in the beginning until about 20,000 Frames were processed. After that, most of the agents are reaching the highest point of 200. In the same figure some “blackouts” can also be recognised, it is happening

more often than for an agent trained with Pong environment. In the figure 38 the average results of training are shown. It can be seen that in the case of CartPole environment differences of seeds can be more recognisable than for Pong environment, where tests with different seeds had very similar behaviour. It can be seen that the seed represented with pink curve needed more steps to be able to reach the peak of learning. Even if the training was done for a limited amount of frames, the agent with this seed processed fewer frames per step than other agents. The average time that was needed to finish the training with CartPole environment was about 6 minutes per seed. Compared to Pong environment that with the same architecture took 350 minutes, CartPole training is faster. It is because this environment does not need convolutional layers and images for training, all processes are working based on raw data from observation space. Also, this environment is more simple than Pong, all needed information can be found without extra calculations.

4.8 Double Deep Q Learning with PER with CartPole environment

In this section, the next CartPole test will be presented. In this test, agent was trained with Double Deep Q-Learning under using of PER approach for computing experience. In the figures 41 and 40 the results of tests with different seeds are presented. As in the experiments with Pong, seeds are represented with different colours in the graphs.

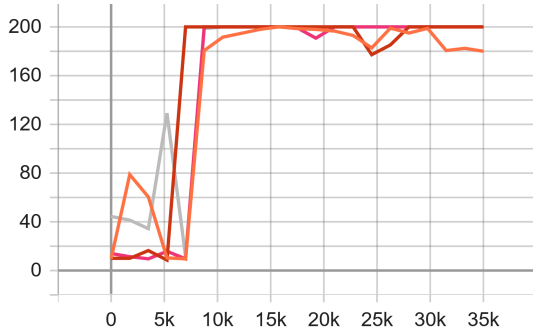

Figure 40: Rewards for DDQN PER Agent trained with CartPole environment

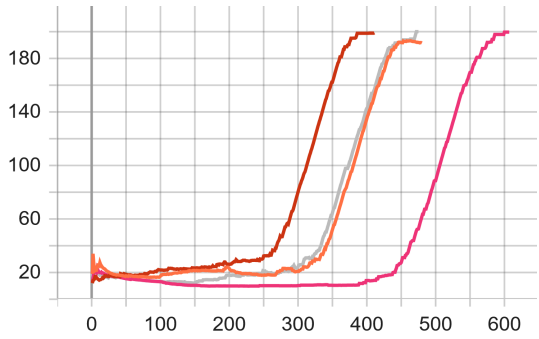

Figure 41: Average results for DDQN PER agent trained with CartPole environment

Figure 42: DDQN PER with CartPole

The graph 40 shows rewards during the learning process. It can be recognised that the instability that was in the training without PER is not happening during this training process. Same as for Pong environment, PER made training with CartPole more stable. Still, the process had some glitches till about 7,500 frames were processed. It can also be seen, that training did not suffer from "blackouts" any more, so if in Pong environment "blackouts" were only reduced, in CartPole environment they were removed with using of PER. This stability can be also seen in the graph where the loss of both training processes was compared (see graph 43). In this figure, the red curve is a PER approach and the orange curve is basic Experienced Replay. In figure 44 average results between the training processes are compared. It can be seen that PER and basic ER approach are performing in the same way. It can be said that PER is bringing stability to the learning process with this environment, but it does not give any significant improvements in terms of learning speed or performance.

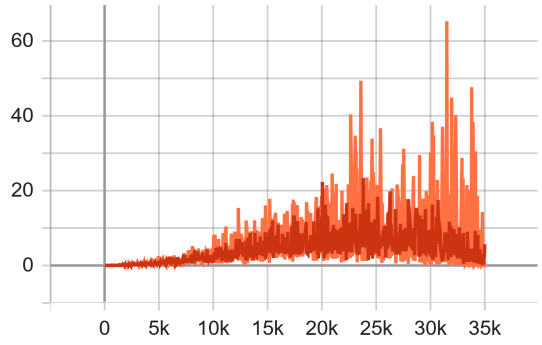

Figure 43: Comparison of loss between DDQN with PER and without PER

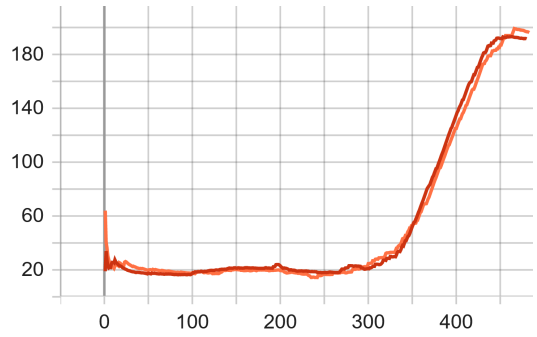

Figure 44: Comparison of average results between DDQN with PER and without PER

4.9 Double Deep Q Learning with Noisy Layers for CartPole environment

In this section, the last test for Double Deep Q Learning architecture that is using noisy layers will be presented. Since for CartPole environment, the convolutional layers are not needed, all used dense layers from the network will be converted to noisy layers for this test.

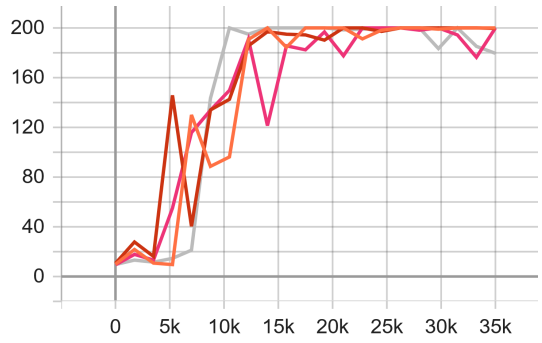

Figure 45: Double Deep Q Learning with noisy layers

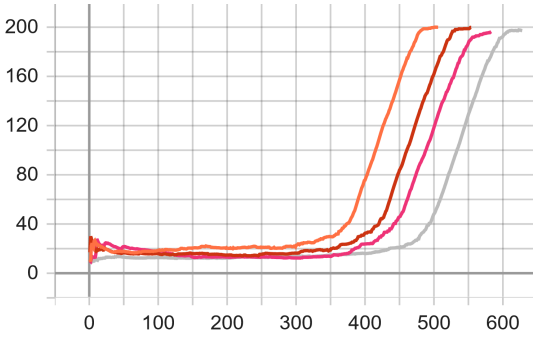

Figure 46: Average results for Double Deep Q Learning with noisy layers

The Graphs 46 and 45 are presenting the results of the learning process. Same as for all other tests, different seeds are presented with different colours. The figure 45 shows how score changed during processed frames, the learning process, that is presented there, does not provide such a steep curve as in the previous tests without noisy layers. The end result also here remains to be about 200 points. In average, agents get about 192 points after learning with this approach. The graph 46 shows an average score that the agent is achieving during training per step. It can be seen that difference in steps between different seeds is only from 25 till 50 steps. It is smaller compared to previous observation. In average, this architecture took about 550 steps and 10 minutes to process 35,000 frames. Same as for the Pong environment, this approach takes more time because of extra computing of noisy weights after each learning step. According to these observations, Double Deep Q Learning architecture with noisy layers shows the worst behaviour among all tests with different architectures for Double Deep Q Learning approach. In the figure 47 a comparison between normal DDQN (orange curve) and DDQN with noisy layer (pink curve) is shown. It can be seen that normal DDQN reaches maximal points faster than architecture with noisy layers. It takes also fewer steps to compute and process the 35,000 frames. Normal DDQN took 483 steps and noisy layers architecture needed 503 steps to finish the

training.

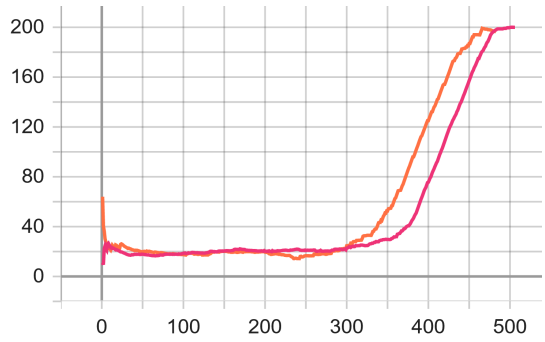

Figure 47: DDQN with noisy layers vs normal DDQN

4.10 Dual DQN for CartPole environment

In this section tests for the next type of architecture, Dual DQN with CartPole environment will be presented. Five tests with different seeds were done to inspect a learning process. The results for them are presented in the figures 49 and 48.

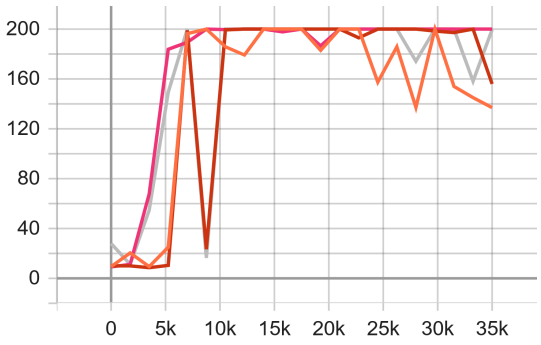

Figure 48: Dual Deep Q Learning for Pong environment

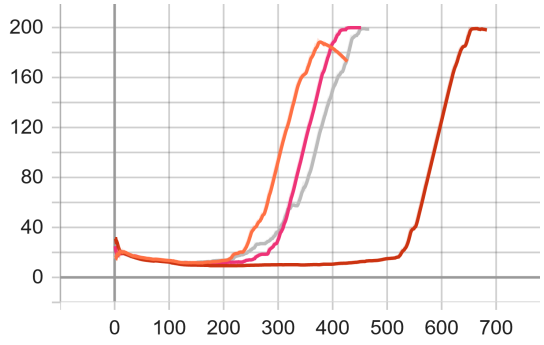

Figure 49: Average results for Dual Deep Q Learning for Pong environment

The graph 49 shows an average result of learning per step. It shows that most of the tested seeds had a similar behaviour, one of them took more steps to finish (red curve) and the other one reached only 170 points after learning (orange curve). From the other figure 48 can be seen that this architecture gives more stable process than the same tests but with normal Double Deep Q Learning approach. Also, in this architecture some "blackouts" during the training can be recognised. But compared to Double Deep Q Learning, the current approach took 4 minutes more in average to finish the learning. In the figure 51 a comparison between Double Deep Q Learning (orange curve) and Dual DQN (gray curve) approach is shown. It shows that the usage of Dueling Network Architecture improves the results of agent for CartPole environment. The improvements in stability can be also seen in the figure 50. This figure also shows that the number of "blackouts" for Dueling Architecture is smaller than for simple DDQN.

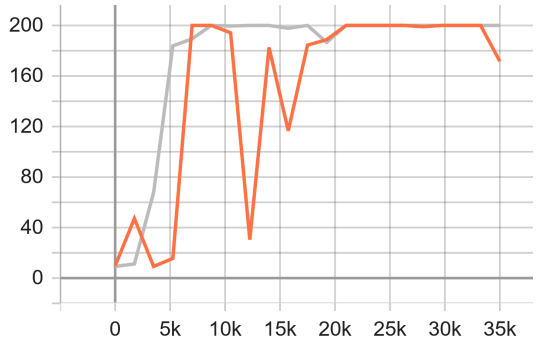

Figure 50: Dual DQN vs DDQN for Pong environment

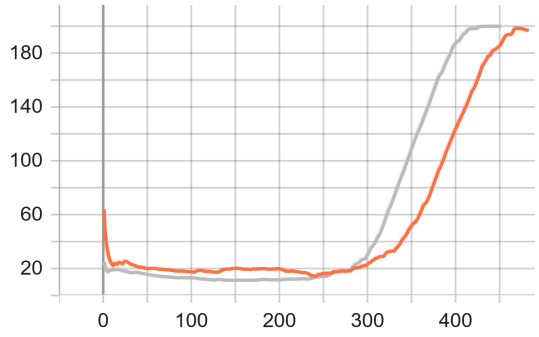

Figure 51: Average results for Dual DQN vs DDQN for Pong environment

4.11 Dual DQN with PER for CartPole environment

In this section for simple Dual DQN approach, a Prioritized Experience Replay will be added. Theoretically, it should reduce the number of "blackouts" and stabilise the learning process even more, compared to previous tests.

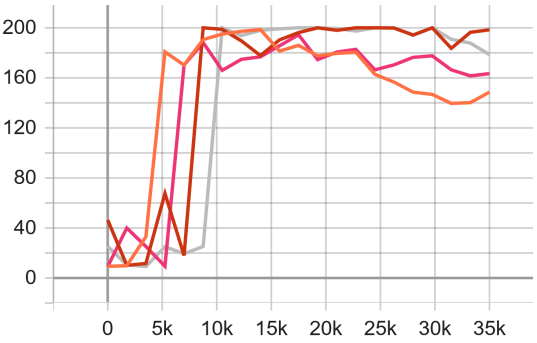

Figure 52: Dual DQN with PER for Pong environment

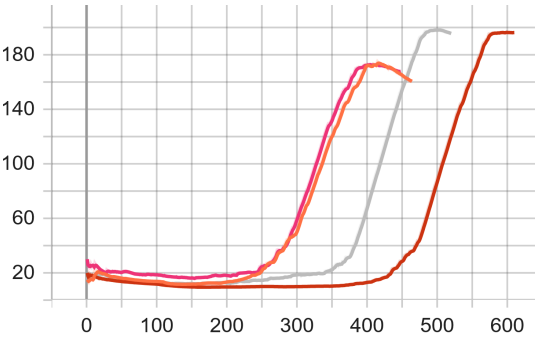

Figure 53: Average results for Dual DQN with PER for Pong environment

In the figure 53 the average results per step for all tests are shown. This graph shows that none of the tests was done successfully and reached 200 points. An average score that was achieved during these tests is 182. The figure 52 shows the results per processed frames. It can be seen that "blackouts" did not happen during the training, and agents were converting to high scores already after 5,000 processed frames. Even with this behaviour, agents do not reach 200 points during the training. Only for one seed (see red curve in figures 53 and 52) this point during training was reached. This behaviour can be caused by prioritizing mechanism of PER approach. The size of buffer is the same as the size of processed frames (35,000), and in the beginning of learning the buffer is filled with values, computed using random policy. Prioritizing algorithm can set high priority to samples that were not computed recently during the learning process, but to the samples from random policy results. This prioritization may contain not the best samples, that can be a reason why agents do not perform well during the training. A second reason that can make the process to look like this is unfavourable choice of seeds. More tests are needed to find out if there are any seeds that can be successful with this type of architecture. This kind of architecture may need more parameter tuning to create successful agents that can reach maximal score.

4.12 Dual DQN with Noisy Layer for CartPole Environment

This section presents the last tests with the last architecture that will be investigated in this paper. It is a Dual DQN architecture with Noisy Layers.

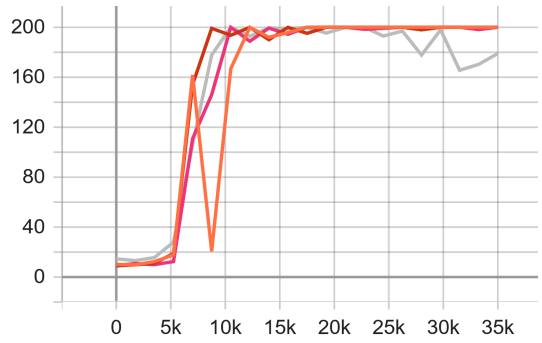

Figure 54: Dual DQN with Noisy Layers for Pong environment

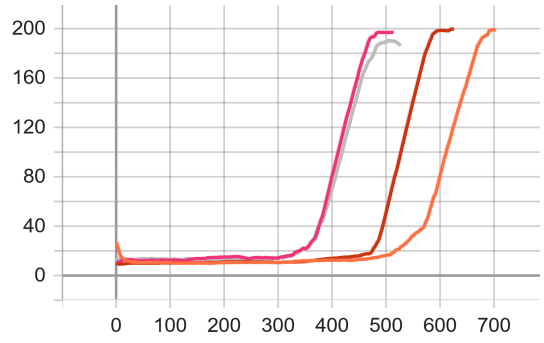

Figure 55: Average results for Dual DQN with Noisy Layers for Pong environment

The results of tests are shown in the figures 54 and 55. The figure 54 shows that the learning process with this type of architecture is stable and most of the tests do not have any "blackout" during training. Only one test (see orange curve) got during a training one "blackout" after processing about 10,000 frames. Among all tests with Dual DQN architecture, the variant with noisy layers is the most stable and does not have rapid changes in score across the whole training process. Agents during the training with this architecture managed to reach in average 195 points from 200 possible. Same as for Pong environment, this architecture shows the best result among others, but it took more time. For Pong, it took 15 minutes to finish each test. It is due to computation time that is needed to compute new noisy weights in each step that was done, also the process of updating of the whole V stream during each step is adding time for the computation. This is the reason the approach with Dual DQN and noisy layers is so time-consuming. The figure 55 shows an average result of training per step. It can be seen, that this type of architecture depends on the seed that will be used during the training. Same as for other approaches, the correct seeds with the best results can be found only after trying out the different seeds.

In the figure 56 a comparison between Dual DQN architectures with different mechanisms is shown. Simple Dual DQN is presented as gray curve, Dual DQN with PER is presented with dark blue curve and Dual DQN with noisy layer is presented with blue curve. From this figure, it can be seen that the best performance in terms of score and time was shown by a Dual DQN architecture without any additional mechanisms. This approach needed fewer steps to process 35,000 frames and reached the same result that Dual DQN with noisy layer reached in almost 700 steps. Dual DQN with PER can be considered the worst candidate for CartPole environment, because the agents that were trained with this approach could not reach the maximum score of 200 points during the training.

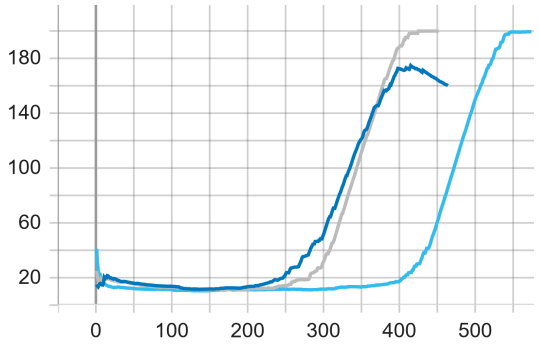

Figure 56: Comparison of all dual DQN approaches

5 Conclusion

Reinforcement learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Creating an agent that can automatically solve a task and adapt to the current environment responses is an area where Reinforcement Learning can be used. There are existing a lot of algorithms and mechanisms that can provide a successful learning for these agents. This paper was created to give a small overview of how some of these algorithms and mechanisms are working, and how the combination of these algorithms can be used to improve the learning process and make the agent perform better by solving different tasks. It shows the results of learning with these algorithms and explains the basics of how they are working from a mathematics perspective.

There were presented two neuronal models: Double Deep Q-Learning and Dueling Double Deep Q-Learning and tree types of optimisation mechanisms: experience replay, prioritized experience replay and Noisy Layers. All of them were tested within two different environments. One is a CartPole and the other one is Atari game Pong. Both of them were using different data for learning process. Pong uses images and CartPole uses simple raw data that will be given to the network as input. With this, the paper shows how presented models and mechanisms work with convolutional layers and without them. For each environment, the learning parameters were set before, seed is the only parameter that is changing from test to test. With this was presented that for Pong environment there is not a big difference of learning when seed is changing, most of the tests had similar behaviour. For CartPole seed played a huge role in the training, some tests could not reach the maximal score in average because of seeds that were chosen. The using of simple learning models with Experienced Replay mechanism shown that both environments suffer from some “blackouts” and instability that can happen during the training. Stability could be given by using other mechanisms like PER and Noisy Layers. These two mechanisms actually also helped to reduce a number of “blackout” that happened in the training process. But they also cannot be helpful all the time. The tests of Dual DDQN with PER for CartPole environment showed bad performance in terms of average reward, even if the number of “blackouts” was reduced. The agents could not reach even 180 points with this type of architecture. One more problem that was faced by the tests, is that the different mechanisms to improve stability make the learning longer, for example for Pong environment with Dual DDQN and Noisy Layers more than 500 minutes were needed to finish the training.

Also, these mechanisms affect the number of frames that would be processed in one step. Some mechanisms improve this number and make it bigger, so the training needs lees steps to convert to maximal rewards. The PER mechanism also added an improvement not in terms of rewards, it improved also a development of the loss function. All done tests for all mechanisms and neuronal

models were visualised with graphs, where all described improvements could be analysed. After the comparison of the results that were given by training with both networks and different stabilisation approaches, it can be seen that these approaches actually helped to stabilise learning with a cost of longer computation time. It shows also that not every time they are needed to achieve good results. It happened for example with CartPole environment, Dual DQN was better than the same architecture with PER or Noisy Layer approach. For Pong environment, presented approaches did not give a significant improvement by learning. First, it was because already with simple networks and Experienced Replay, maximum score of 21 could be reached. The improvements could be more visible for environments, where no score limits are existing. Still, for Pong only one architecture reached in average the maximum of 21 points and showed the best performance, it was Dueling DDQN with Noisy Layers. The disadvantage of this approach for Pong was the time that it took for training, it was more than 500 minutes per seed. To summarize, it can be said that also for this simple environment, used approaches gave more stability and performance, even if the cost was the time for computation.

References

[1] 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, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[3] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[4] L.-J. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, USA, 1992, uMI Order No. GAX93-22750.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015, cite arxiv:1511.05952Comment: Published at ICLR 2016. [Online]. Available: http://arxiv.org/abs/1511.05952

[6] J. Schmidhuber, "Curious model-building control systems," in *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, 1991, pp. 1458–1463 vol.2.

[7] "Deep q learning," https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/, last Access 12.01.2021.

[8] 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

[9] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[10] "Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed...," https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/, last Access 15.01.2021.

[11] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy networks for exploration," *CoRR*, vol. abs/1706.10295, 2017. [Online]. Available: http://arxiv.org/abs/1706.10295

[12] I. Osband, B. Van Roy, and Z. Wen, "Generalization and exploration via randomized value functions," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, p. 2377–2386.

[13] S. Dittert, "Dqn-atari-agents: Modularized pytorch implementation of several dqn agents, i.a. ddqn, dueling dqn, noisy dqn, c51, rainbow and drqn," https://github.com/BY571/DQN-Atari-Agents, 2020.

[14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[15] "Cartpole," https://gym.openai.com/envs/CartPole-v1/, last Access 17.01.2021.