Jagiellonian University
Department of Theoretical Computer Science

Kamil Kropiewnicki
kamil.kropiewnicki@gmail.com

Value-based methods of deep reinforcement learning

Bachelor's Thesis
Computer Science – IT Analyst

Supervisor:
dr Michał Wrona

August 2018

ABSTRACT

In this thesis we provide a brief introduction to reinforcement learning. After reviewing basic knowledge in the domain we take a look at solving complex problems by combining reinforcement learning value-based methods and artificial neural networks. We discuss and implement methods introduced in four fundamental papers of deep reinforcement learning: “Playing Atari with Deep Reinforcement Learning”[dqn], “Human-level control through deep reinforcement learning”[dqn-target], “Deep Reinforcement Learning with Double Q-learning”[ddqn] and “Dueling Network Architectures for Deep Reinforcement Learning”[dueling]. We apply our implementations on two Atari games, Breakout and Space Invaders, provided by OpenAI Gym[gym]. The objectives of this thesis are: presenting the foundations of reinforcement learning, delivering implementations of the mentioned methods and testing how they work with shared hyperparameters. Basic knowledge about machine learning is assumed.

Content

1. INTRODUCTION TO REINFORCEMENT LEARNING

Humans, among other species, learn by collecting experiences and drawing conclusions from them. This trial-and-error method seems to be a very natural approach for learning without strict supervision, given that a student at least vaguely knows how to improve their actions. No wonder it has been an inspiration for researchers working on artificial intelligence. These observations laid the ground for reinforcement learning.

Reinforcement learning is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning. [sutton]

Due to the unique nature of reinforcement learning, it derives from many fields such as computer science, engineering and mathematics as well as neuroscience and psychology. One may perceive it as using the former to implement ideas inspired by the latter. Since reinforcement learning systems improve their performance over time, it falls under machine learning category. However, its goal is not finding a hidden structure in data, it has no supervision either. That is why it is called the third machine learning paradigm, next to unsupervised and supervised learning.

What makes reinforcement learning unique?

- There is no labeled data, only a reward signal.
- Feedback is usually not instantaneous.
- Data is hardly ever independent and identically distributed (since it is generated as the effect of consecutively taken actions).
- Exploration, exploitation and the trade-off between them.

Examples of reinforcement learning problems include:

- Making a robot (or virtual model of robot) move.
- Playing a game (chess/go/Atari Breakout/Starcraft).
- Driving an autonomous car.
- Finding the optimal routing in a dynamic network.

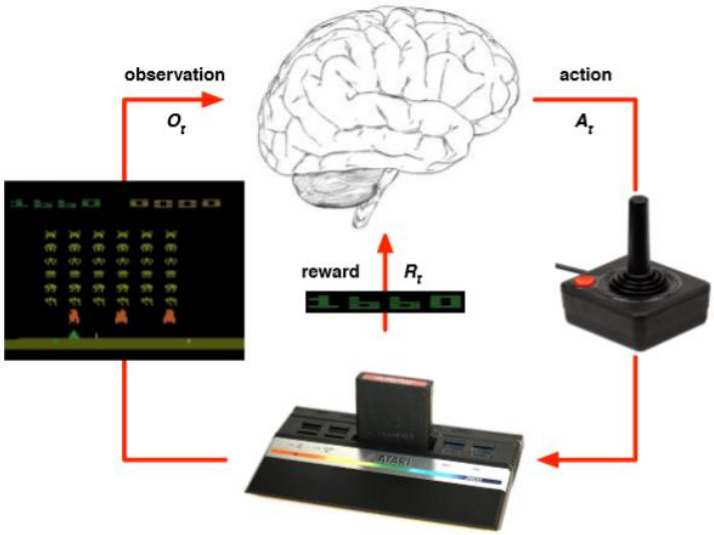

Figure 1. *The reinforcement learning loop. Source: [ucl-course]*

Reinforcement learning environments may be split into two categories:

- Fully observable, where an agent directly observes an environment state that provides complete information. An example of a fully observable environment is a chess board. In a fully observable environment every single observation is sufficient to build a state with Markov property, which means that the probability distribution of future states depends only upon the present state (and the chosen action), not on the sequence of events that preceded it. Hence it is often stated that $\text{agent state} = \text{observation} = \text{environment state}$. Note that while an agent state may be equal with the last observation, it does not have to be – we may decide to drop some information in order to obtain a more concise, memory efficient state representation. It is important not to crop too much as it may result in losing Markov property.
- Partially observable, where an agent indirectly observes an environment state. An example of a partially observable environment is a robot with a camera in a maze. An agent's observation would be a current frame from the camera, whereas an environment state would be a position of the robot in the maze. Contrary to the fully observable setting, here we may decide to enrich an agent's observation since a single one does not provide enough information. The robot may need to accumulate knowledge based on the history of traversal in order to have more information required to find a solution. In a partially observable environment a single observation is not sufficient to build an agent state with Markov property.

More formally, we may describe a fully observable environment as a Markov decision process (MDP). A period between starting a game and losing all lives is called an episode. t denotes a current time step in a given episode and T denotes a terminal time step. If specified, losing a single life may be interpreted as a termination.

Definition:
A Markov decision process is a tuple (S, A, P, R, γ), where:

- S is a set of states. $\forall s \in S: s$ holds Markov property.
- A is a set of actions.
- P is a state transition probability function. $P(s, a, s') = P(S_{t+1} = s' \mid S_t = s,\ A_t = a)$.
- R is a reward function, $\mathrm{R}(s, a) = \mathbb{E}[r \mid S_t = s,\ A_t = a]$.
- $\gamma \in [0,1]$ is a discount factor, which represents the difference in importance between future and immediate rewards. For example, if $\gamma = 0$, then only an immediate reward obtained after taking an action is taken into account. $\gamma = 1$ is allowed only if all sequences terminate – otherwise it could result in a divergent series.

R_t is a reward obtained after being in the state S_t and taking the action A_t. G_t is a complete return from time step t equal to the sum of discounted rewards received after time step t.

$$G_t = \sum_{i=0}^{T} \gamma^i R_{t+i} \ \textit{including the possibility that } T = \infty \textit{ or } \gamma = 1 \ (\textit{but not both}).$$

A goal of reinforcement learning is to find an optimal policy – a strategy that maximizes cumulative reward obtained by an agent in a given environment.

Definition:
Let $M = (S, A, P, R, \gamma)$ be an MDP. A policy π is a probability distribution over actions for each state.
$\pi(a|s) = P[A_t = a|S_t = s]$

- A policy fully defines the behavior of an agent.
- Due to the Markov property MDP policies depend only on a current state.
- A deterministic policy is a function that maps a state to an action.

There are three classes of algorithms present in reinforcement learning:

- Policy-based methods which operate directly on a policy π.
- Value-based methods which compute a value function from which a policy can be obtained. Generally, a value function takes one or two arguments:

- $V_\pi(s) = \mathbb{E}_\pi[G_t \,|S_t = s]$ is the expected return starting from the state s and then following the policy π. It describes how valuable it is to be in the state s.
- $Q_\pi(s,a) = \mathbb{E}_\pi[G_t \,|S_t = s, A_t = a]$ is the expected return starting from the state s, taking the action a and then following the policy π. It describes how valuable it is to take the action a being in the state s.

- Actor critic, which utilize the two above.

Three commonly-used policies can be identified:

- Random policy – each action is taken with the same probability.
- Greedy policy – an action with the highest expected reward is always taken.
- ε-greedy policy – for given $\varepsilon \in [0,1]$, an action is selected according to the random policy with ε probability; the greedy policy is used otherwise.

As mentioned before, one of the features separating reinforcement learning from the other machine learning paradigms is the trade-off between exploitation and exploration. On the one hand, an agent should take actions that seem to return the highest reward as they are the most promising ones. On the other hand, some state-action pairs may be underestimated or even undiscovered. Choosing only the best actions may result in reaching a poor local minimum of the true value function. This dilemma leads to the conclusion that during a training an agent should explore rarely visited state-action pairs as well as exploit its current knowledge and follow the most promising ones. The ε-greedy policy with value of ε decreased in time is a simple policy that implements this idea.

Definition:
A model of an environment is an agent's representation of an environment. A model mimics the behavior of it – predicts what the actual environment would do next. It is an approximation of the MDP that describes the actual environment.

We can distinguish two methods of reinforcement learning based on the model factor:

- Model-free, where an agent learns the about optimal policy by interacting with an actual environment. It involves a trial-and-error approach as it learns to solve the problem directly using samples from environment. Underlying idea is that usually an agent does not need to understand all the rules and mechanics of a given environment in order to behave well in it. Model-free methods are often referred to as learning methods. Examples of model-free algorithms include Q-learning and SARSA.
- Model-based, which consists of two steps. The first one is to build a model – an estimation of an environment. An agent is able to choose the optimal actions since it can predict the behavior of an environment. This approach is preferred in situations when learning directly from an environment is very expensive. For example, it is convenient to create a simulation of the real world in order to train a robot or an autonomous car. Learning from scratch in the real world would result in many damages and possibly dangerous situations. Model-based methods are often referred to as planning methods.

Bootstrapping and Bellman Equation

As mentioned before, value-based methods utilize a state-value function or an action-value function or both. For MDPs the state-value function under a given policy π can be defined as:

$$V_\pi(s) = \mathbb{E}_\pi[G_t \,|S_t = s\,] = \mathbb{E}_\pi[\sum_{i=0}^{T} \gamma^i R_{t+i} \,|S_t = s\,] = \mathbb{E}_\pi[R_t + \gamma G_{t+1} | S_t = s]$$

$$= \sum_a \pi\,(a|s) \sum_{s',r} P(s', r\,|s, a)\,[r + \gamma V_\pi(s')]$$

The last equation is the Bellman equation for V_π. A succinct way to express it with matrix notation is:

$$V_\pi = R_\pi + \gamma P_\pi V_\pi$$

Similarly, the action-value function:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{i=0}^{T} \gamma^i R_{t+i} \mid S_t = s, A_t = a] = \mathbb{E}_\pi[R_t + \gamma G_{t+1} \mid S_t = s]$$

$$= \sum_{s',r} P(s', r \mid s, a)\,[r + \gamma \sum_{a'} \pi(a' \mid s') Q_\pi(s', a')]$$

The first equations refers to Monte-Carlo approach. However, we can observe the recursive nature of both value functions, which lead to the concise formula with a look-ahead step. This idea is called bootstrapping. Thanks to it we can obtain the true value function with dynamic programming methods. One of them will be presented later on.

Let's define partial order on policies:

$\pi \geq \pi' \; if \; \forall s: V_\pi(s) \geq V_{\pi'}(s)$

The optimal value function is the best value function over all policies. Strictly speaking, it is $V_\pi(s)$ for a policy π that is the greatest element in partial order set of policies. For any finite Markov Decision Process there exists at least one optimal policy π_*[sutton]. While there may be many optimal policies, they generate the same optimal value function.

The optimal state-value function: $V_*(s) = \max_\pi V_\pi(s)$

The optimal action-value function: $Q_*(s,a) = \max_\pi Q_\pi(s,a)$

Bellman optimality equation

It is obvious that the greedy policy combined with the optimal value function should provide the highest expected reward. Having that in mind, Bellman equation and the greedy policy form Bellman optimality equation:

$$V_*(s) = \mathbb{E}_{\pi_*}[G_t \mid S_t = s] = \mathbb{E}_{\pi_*}[\sum_{i=0}^{T} \gamma^i R_{t+i} \mid S_t = s] = \mathbb{E}_{\pi_*}[R_t + \gamma G_{t+1} \mid S_t = s]$$

$$= \max_a \sum_{s',r} P(s', r \mid s, a)\,[r + \gamma V_*(s')] = \mathbb{E}_{\pi_*}[R_t + \gamma V_*(S_{t+1}) \mid S_t = s]$$

The same applies to an action-value function:

$$Q_*(s,a) = \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] = \mathbb{E}_{\pi_*}[\sum_{i=0}^{T} \gamma^i R_{t+i} \mid S_t = s, A_t = a] = \mathbb{E}_{\pi_*}[R_t + \gamma G_{t+1} \mid S_t = s]$$

$$= \sum_{s',r} P(s', r \mid s, a)\,[r + \gamma \max_{a'} Q_*(s', a')] = \mathbb{E}_{\pi_*}[R_t + \gamma Q_*(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

In some cases, using an action-value function may be more convenient. For instance, deriving the optimal step for a given state does not require doing look-ahead – Q-function provides all the necessary information.

Algorithms may be divided into two categories: on-policy and off-policy. The former attempt to evaluate or improve the policy that is currently used to generate data. The latter is the opposite – it can work on any data while improving a current policy.

The examples of on-policy algorithms include policy-iteration and SARSA, which are discussed in detail in the book[sutton].

An example of an off-policy algorithm is Q-learning. It is an off-policy method because it does not follow a current policy during an action-value function improvement. TD in the description stands for Temporal-Difference – the algorithm updates the value function using the scaled by a factor α (learning rate) difference between the current estimation and the one obtained with one-step look-ahead.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Algorithm 1. *Q-learning algorithm. Source: [sutton]*

2. DEEP REINFORECEMENT LEARNING

In theory, methods discussed in the previous chapter are applicable to all reinforcement learning problems. In practice, in many interesting problems state space or action space are too large (sometimes even continuous) to fit all the data in matrices and work on them. Due to this fact it is convenient to use artificial neural networks (abbreviated to ANN) as function approximators. They have many advantages over traditional approach: they scale well and can update values for similar states at once thanks to their ability to generalize.

Atari games provide a wide variety of problems that seem to be interesting from reinforcement learning perspective. We will use one algorithm, architecture and set of hyperparameters to solve different games. The only input to our artificial neural network will be frames from the game. It is the most natural choice since humans also play these games using solely visual input. Moreover, it is the most convenient input as it requires no prior information about a game. It is robust to selection of a game and does not require feature handcrafting at all. Thanks to these traits we will obtain “one for all” solution.

OpenAI Gym[gym] enables playing Atari games used in this thesis. Gym provides access to the same environment that was used in DeepMind’s papers[dqn][nature][ddqn][dueling]. Games are rendered as 210px x 160px RGB frames. In each step an agent chooses one of available actions and receives a tuple of an observation, a reward and a done flag. An observation is a current frame of the game, a reward is a scalar and a done flag is a Boolean value indicating whether a game is over.

Each obtained frame is preprocessed: resized (downsampled) and turned to grayscale for the sake of memory efficiency. Cropping image to 84x84 size was introduced in [dqn] because of the hardware requirements. Nowadays hardware does not have such limitations – however, reducing state space without losing information is computationally efficient. Since each of the implemented paper involve cropping an image to 84x84, we do this as well. We will represent an agent state as four last seen preprocessed frames – it enables capturing a movement. Such states have Markov property only in a subset of Atari games, including Breakout and Space Invaders. In these games all the information is always present on a screen. On the other hand, in more complex games like Montezuma Revenge a screen contains only a portion of information about a current state in a game, which results in an agent state that does not have Markov property.

We also use a simple frame-skipping technique. It means that only every 4th frame is considered, and all frames in between are discarded. It is motivated by the fact that performing steps in Atari games is computationally cheap comparing to action selection (using ANN) and a state does not change very much during this little period of time. The selected action is repeated during the skipped frames. Since making a step in the OpenAI Gym environment is almost instant it speeds up a training up to 4 times. However,

due to the fact that games have flickering screen, it is advised to take a maximum from two adjacent frames. Summing up, a state is presented as:

$S_{t/4} = [\max(p_{t-13}, p_{t-12}), \max(p_{t-9}, p_{t-8}), \max(p_{t-5}, p_{t-4}), \max(p_{t-1}, p_t)]$
where p_i denotes i^{th} preprocessed frame.

It is advised to use OpenAI gym games with "NoFrameskip-v4" suffix to replicate these conditions. However, the flickering effect was not observed in Breakout game, hence "BreakoutDeterministic-v4" version was used (it has built-in frameskipping=4). In "Deterministic-v4" environment a state is represented as: $S_{t/4} = [p_{t-12,}, p_{t-8}, p_{t-4}, p_t]$. In case of lack of previous frames, the most recent ones are repeated, for example: $S_0 = [p_0, p_0, p_0, p_0]$.

The common part of all artificial neural network architectures used in this thesis are 3 convolutional layers followed by a fully-connected layer. Using convolutional layers is an obvious choice since our neural networks are fed with visual input. All neural networks begin with:

Common part of architectures

Input = Preprocessed input(4x (210,160,3) -> (84, 84, 4))

1. Conv1(input: Input, filters_num: 32, kernel_size: 8x8, stride: 4, activation: ReLU)
2. Conv2(input: Conv1, filters_num: 64, kernel_size: 4x4, stride: 2, activation: ReLU)
3. Conv3(input: Conv2, filters_num: 64, kernel_size: 3x3, stride: 1, activation: ReLU)
4. DenseLayer1(input: Conv3, outputs: 512, activation: ReLU)

Figure 2. *Common part of every ANN used in this thesis. Based on [dqn-target]*

ReLU (ReLU(x) = max(0, x)) is applied as the activation function on the output of every layer mentioned so far. All weights of each layer discussed in this thesis are initialized with Xavier[glorot] initialization. Also, all layers include bias unit initialized in the same way.

To deal with data that is not independent and identically distributed we use biologically inspired[dqn-target] Experience Replay technique. Instead of learning from consecutive steps, each experience $e_t = (s_t, a_t, r_t, done_t, s_{t+1})$ is stored in replay memory. During the learning phase an update is applied only on mini-batches of experiences randomly sampled from replay memory. It results in breaking correlation between training data, thus improving performance. Another worth-mentioning advantage is data-efficiency: one experience is used in many updates. It is an important feature since in many environments generating data is costly. A policy during generating an experience may differ from the one during learning from it. Hence, it is mandatory to use an off-policy algorithm. Because memory of computer is limited, only up to N last seen experiences are stored in replay buffer.

To make a training more stable with the same values of hyperparameters applicable to all Atari games, it may be helpful to clip all rewards to [-1;1] – with all negative rewards casted to -1 and all positive to 1. It may also result in smaller errors, thus smaller gradients and more stable training.

3. METHODS

3.1 DEEP Q-LEARNING, DQN

Deep Q-Network (abbreviated to DQN) is the artificial neural network designed to be used with Q-learning algorithm. Its architecture consists of the layers discussed in Chapter 2 followed by another fully connected layer with $|A|$ outputs. No activation function is applied on the output nodes since their values are interpreted as $\mathrm{Q}(s, a)$ – for given input representing a state DQN estimates value of each action. Since using the greedy policy requires estimating values of all actions, using ANN estimating a state-value function would be highly inefficient – it would result in linearly increased time complexity in regard of $|A|$.

DQN architecture

Input = Preprocessed input(4x (210,160,3) -> (84, 84, 4))

1. Conv1(input: Input, filters_num: 32, kernel_size: 8x8, stride: 4, activation: ReLU)
2. Conv2(input: Conv1, filters_num: 64, kernel_size: 4x4, stride: 2, activation: ReLU)
3. Conv3(input: Conv2, filters_num: 64, kernel_size: 3x3, stride: 1, activation: ReLU)
4. DenseLayer1(input: Conv3, outputs: 512, activation: ReLU)
5. Q = DenseLayer2(input: DenseLayer1, outputs: $|A|$, activation: None)

Figure 3. *DQN architecture[dqn-target]. The same architecture is used in methods from Chapters 3.2 and 3.3.*

Unlike in traditional Q-learning algorithm with error defined as a difference, DQN may use squared difference. However, to make a training more stable, it is useful to apply gradient clipping – using squared difference in (-1; 1) interval and absolute error outside of it. It forms Huber loss which is advised to be used as the loss function. Errors for actions not present in given experience are equal to 0.

Another novelty is the definition of action-value function Q. It is parametrized by θ which stands for all the weights present in an ANN and is interpreted as a network itself. This notation underlines the fact that the function is approximated by ANN, points to the source of estimations and thus provides the ability to use more than one ANN in an algorithm.

Having all components discussed, let's present deep Q-learning algorithm:

Deep Q-Learning with Experience Replay

$Initialize\ replay\ memory\ D\ to\ capacity\ N$
$Initialize\ action-value\ function\ Q\ with\ random\ weights$
$\boldsymbol{for}\ episode\ = 0\ \boldsymbol{to}\ M\ \boldsymbol{do}$
  $o_0 = env.reset()$
  $s_0 = build\_state(o_0)$
  $\boldsymbol{for}\ t = 0\ \boldsymbol{to}\ T\ \boldsymbol{do}$
    $With$ probability $\varepsilon\ select\ a\ random\ action\ a_t, otherwise\ select\ a_t = \max_a \left(Q(s_t, a;\ \theta)\right)$
    $o_{t+1}, r_t, done_t = env.step(a_t)$
    $s_{t+1} = build\_state(o_{t+1,} s_t)$
    $store\ experience\ (s_t, a_t, r_t, done_t, s_{t+1})\ in\ D$
    $Sample\ random\ minibatch\ of\ experiences\ (s_i, a_i, r_i, done_i, s_{i+1})\ from\ D$
    $\boldsymbol{for}\ j = 1\ \boldsymbol{to}\ |minibatch|\ \boldsymbol{do}$
      $$y_j = \begin{cases} r_j\ for\ terminal\ s_{j+1} \\ r_j + \gamma \max_a Q(s_{j+1}, a;\ \theta)\ for\ non-\ terminal\ s_{j+1} \end{cases}$$
      $loss_j = Huber\_loss(y_j - Q(s_j, a;\ \theta))$
    $\boldsymbol{end\ for}$
    $loss = \sum_{j=1}^{|minibatch|} loss_j$
    $Run\ optimizer\ on\ \theta\ to\ minimize\ loss$
  $\boldsymbol{end\ for}$
$\boldsymbol{end\ for}$

Algorithm 2. *Deep Q-Learning with Experience Replay. env.reset() resets the environment and returns the first observation. env.step(a) makes a step in the environment and returns an observation, a reward and a done flag. build_state(o) stacks preprocessed observation o 4 times and returns it as a state. Build_state(o, s) builds a state by stacking the last 3 observations from state s and a preprocessed observation o. Source of the algorithm: [dqn].*

3.2 DQN WITH TARGET NETWORK

Contrary to supervised learning, there is no oracle in reinforcement learning. Using the same network for estimating predictions and targets leads to correlation affecting stability. In order to at least partially address this issue the target network[target-dqn] θ_{target} is introduced. It is a periodic copy of a DQN θ_{online} with frozen weights which is used to estimate target in a loss function. This slight modification improves stability during training process resulting in better overall performance.

$$y_j^{targetDQN} = r_j + \gamma \max_a Q(s_{j+1}, a; \theta_{target}) \; for \; non-terminal \; s_{j+1}$$

Equation 1. *Target estimation in Target DQN. It must substitute y_j for non-terminal states in Algorithm 2 in order to obtain Deep Q-Learning with Target Network and Experience Replay.*

3.3 DOUBLE DQN

So far, max operator present in estimating Q-function have used the same values to select and evaluate an action. It is potentially a source of misbehavior – in this setting an agent is more likely to choose overestimated values. It results in propagating too high values to other state-action pairs, thus the error is spread widely.

Double Q-learning[ddqn] addresses this issue by decoupling selection and evaluation of actions. More specifically, an online DQN is used to select the most promising action for a target, but a target network (as in previous section) still evaluates its value. DQN that uses Double Q-Learning is called Double DQN (abbreviated to DDQN).

$$y_j^{DDQN} = r_j + \gamma \; Q\left(s_{j+1}, \arg\max_a Q(s_{j+1}, a; \theta_{online}); \theta_{target}\right) \; for \; non-terminal \; s_{t+1}$$

Equation 2. *Target estimation in Double DQN. It must substitute y_j for non-terminal states in Algorithm 2 in order to obtain Double Deep Q-Learning with Experience Replay.*

3.4 DUELING DQN

Dueling DQN[ddqn] introduces a different architecture for Deep Q-Learning algorithm. The architecture contains two streams, one for estimating a state-value function $\mathrm{V(s)}$ and one for estimating an advantage function $\mathrm{A(s,a)}$. Then they are combined together to produce the action-value function: $\mathrm{Q(s,a) = V(s) + A(s,a)}$. An advantage function $\mathrm{A(s,a)}$ estimates how much better it is to take a given action comparing to the other ones. There are two possibilities:

$\mathrm{Q(s,a) = V(s) + A(s,a) - \max_a (A(s,a))}$ and $\mathrm{Q(s,a) = V(s) + A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a')}$.

Only the latter one will be used since it is more stable during trainings[ddqn].

Intuitively, for some states it is unnecessary to know the estimated value of an action. It may occur when the current state is good enough and taking any action will not really affect this situation. On the other hand, there are states that are heavily affected by the chosen actions. This insight is a key motivation behind using a dueling architecture.

Dueling DQN architecture

Input = Preprocessed input(4x (210,160,3) -> (84, 84, 4))

1. Conv1(input: Input, filters_num: 32, kernel_size: 8x8, stride: 4, activation: ReLU)
2. Conv2(input: Conv1, filters_num: 64, kernel_size: 4x4, stride: 2, activation: ReLU)
3. Conv3(input: Conv2, filters_num: 64, kernel_size: 3x3, stride: 1, activation: ReLU)
4. DenseLayer1(input: Conv3, outputs: 512, activation: ReLU)
5. V = DenseLayer2(input: DenseLayer1, outputs: 1, activation: None)
6. A = DenseLayer3(input: DenseLayer1, outputs: $|A|$, activation: None)
7. Q = A + (V – mean(A))

Figure 4. *Dueling DQN architecture. Inspired by [dueling], but in the paper V and A do not share input.*

4. IMPLEMENTATION

We succeeded in implementing DQN and its extensions in Python3 using TensorFlow. Each training was performed on a single GPU – GeForce GTX 980 – and lasted for 10M frames, which took between 17 and 23 hours. Obviously, the fastest training method was Simple DQN and the slowest was Dueling DDQN. The chosen game and performance during evaluation also had an impact on training time. All hyperparameters are listed on the top of dqn-agent.py[github] file. They resemble those provided in [target-dqn]. However, we used smaller replay memory size due to hardware limitations – in Space Invaders we stored last 100 000 frames, in Breakout last 300 000 frames. RMSProp[rmsprop] was used as the optimization algorithm. The optimizer's parameters differed from the ones used in DeepMind's papers. We used: $learning\_rate = 2.5e-4$, $decay = 0.99$, $momentum = 0$, $epsilon = 1e-6$. We did not terminate a game after a loss of life.

Each plot shows a score obtained during an evaluation performed during a training session. Evaluations were run every 50 episodes. Due to the fact that the ε-greedy policy with $\varepsilon = 0.05$ was used during an evaluation, the experiences from evaluations were not stored in replay memory as they might affect the exploration stage of a training. The plots were generated using Tensorboard with smoothing parameter equal to 0.9. A faded part of each plot shows true data while a strong line shows smoothed data. Each interval on X-axis corresponds to about 20 evaluations. Y-axis corresponds to the score obtained during evaluation.

For the purpose of this thesis only [target-dqn] was precisely reproduced – there are some minor differences between our implementations and those presented in the other papers. For example, in our architectures fully-connected hidden layer always had 512 nodes, whereas [dqn] proposes 256 nodes and [dueling] proposes 1024 nodes. Also, due to our hardware limitations we used smaller replay memory size and shorter training sessions. We did not use Prioritized Experience Replay. Reward clipping was applied only in Breakout. Hyperparameters were constant across all the implemented methods. Summing up, we implemented Target DQN and made only the necessary modifications to obtain other methods.

4.1 SIMPLE DQN

Surprisingly, our Simple DQN did not manage to learn any interesting policy on Breakout game. During video replay we observed that the paddle tends to get stuck at one of the walls. However, this implementation is able to play Space Invaders.

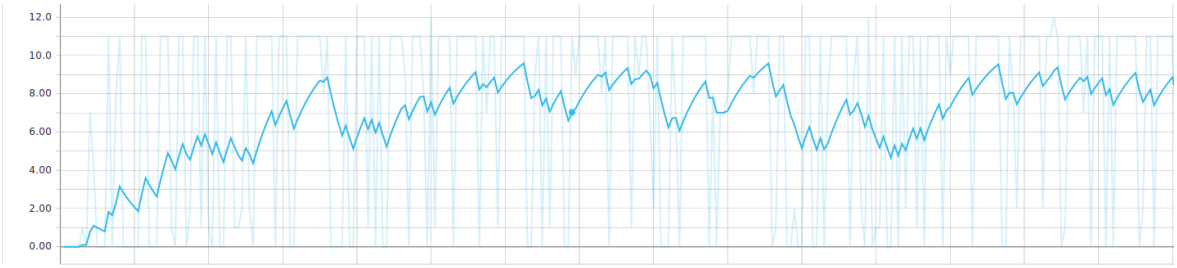

Figure 5. *Simple DQN on BreakoutDeterministic-v4.*

4.2 TARGET DQN

Precisely implemented ANN from [target-dqn]. The charts from training DQN with target network illustrate significant improvement of performance in both games.

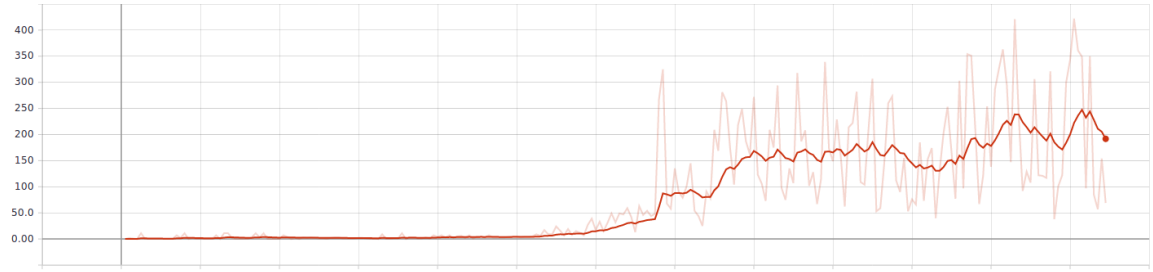

Figure 6. *Target DQN on BreakoutDeterministic-v4.*

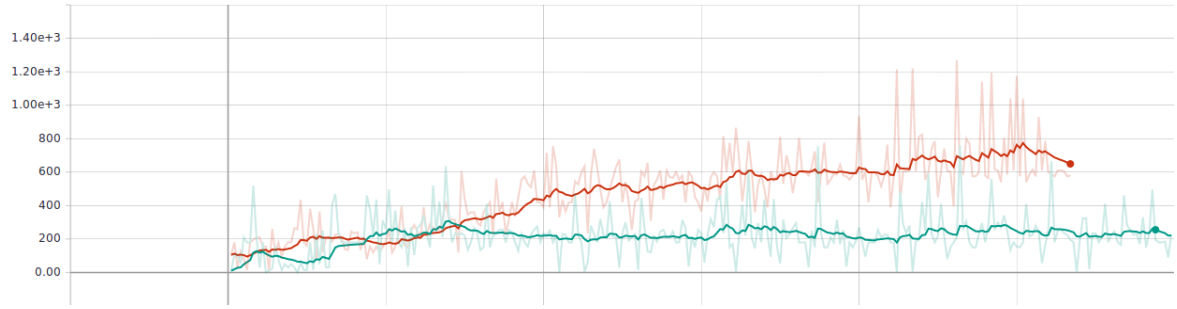

Figure 7. *Simple DQN (blue) and Target DQN (brown) on SpaceInvadersNoFrameskip-v4.*

4.3 DDQN

There is no noticeable difference in performance between this method and the previous one.

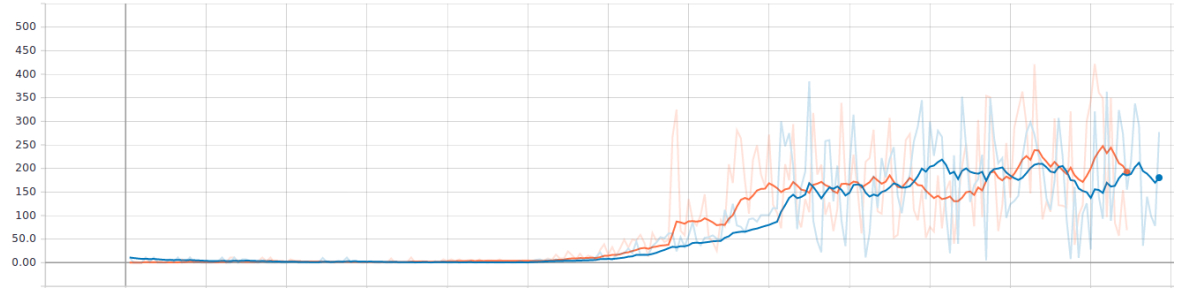

Figure 8. *Double DQN (blue) and Target DQN (brown) on BreakoutDeterministic-v4.*

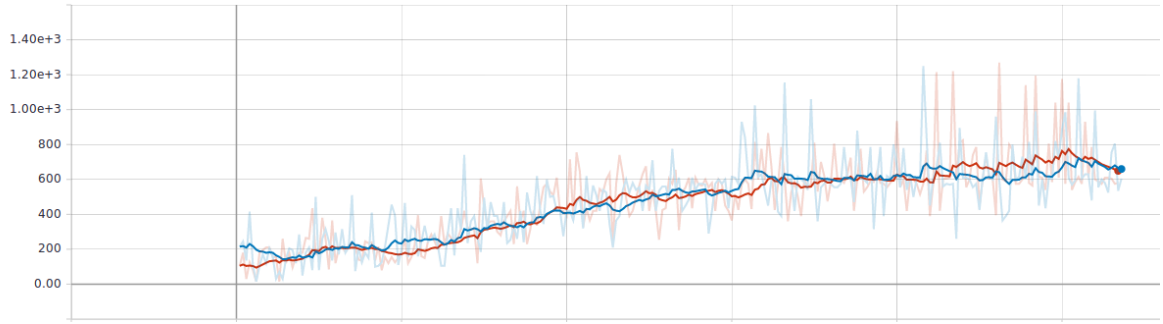

Figure 9. *Double DQN (blue) and Target DQN (brown) on SpaceInvadersNoFrameskip-v4.*

4.4 DUELING DQN

Similarly to DDQN, Dueling DQN does not provide any improvements either.

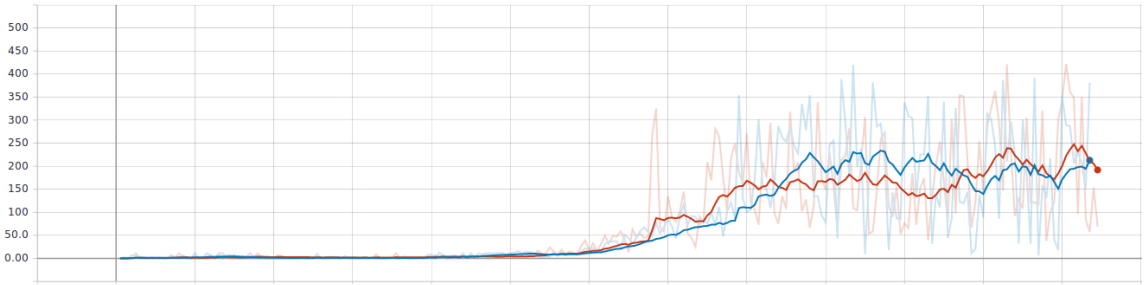

Figure 10. *Dueling DQN (blue) and Target DQN (brown) on BreakoutDeterministic-v4.*

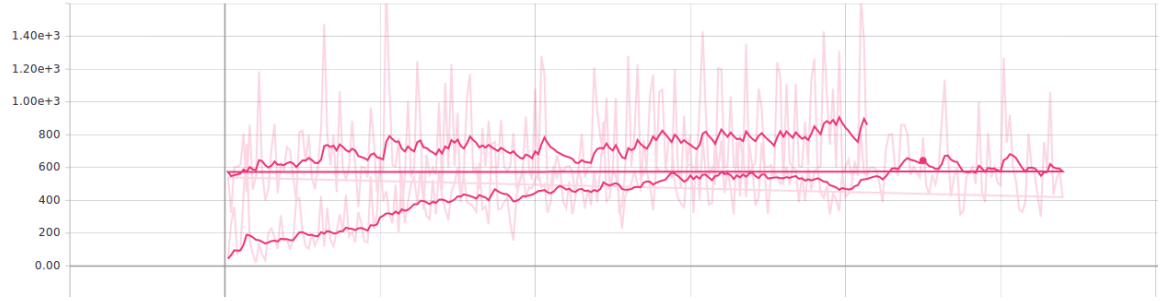

Figure 11. *Dueling DQN on SpaceInvadersNoFrameskip-v4. The horizontal line indicates the end of 10M frames training session and the beginning of the extended one.*

4.5 DUELING DDQN

Since dueling architecture and Double Deep Q-Learning are two independent of each other improvements, they can be combined together. Dueling DDQN is a method that makes use of both techniques.

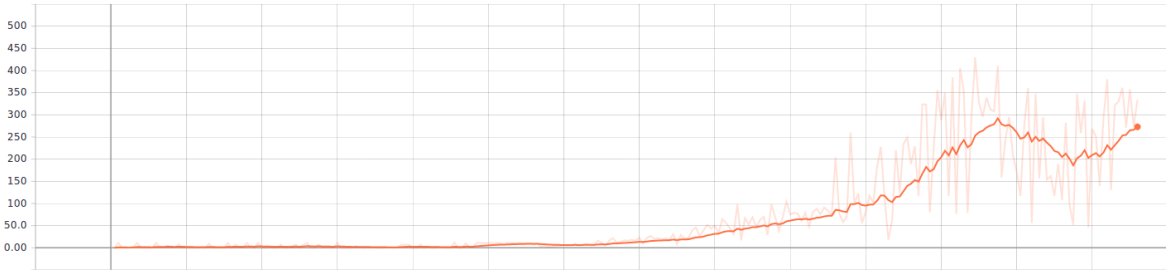

Figure 12. *Dueling DDQN on BreakoutDeterministic-v4.*

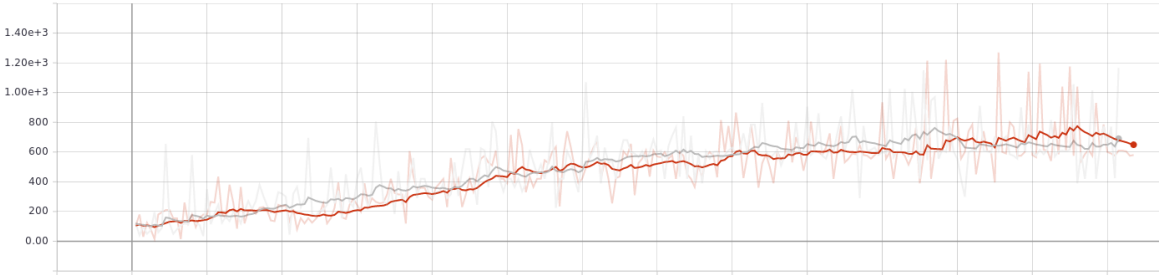

Figure 13. *Dueling DDQN (grey) and Target DQN (brown) on SpaceInvadersNoFrameskip-v4.*

4.6 BINARY FRAME

We tested the influence of additional preprocessing of images called “binary frame”. It assigns value 1 to all but black pixels (background color). The underlying idea was to make input to an ANN more transparent. We tested that idea on Breakout with DQN Target. It did not significantly change the results – surprisingly, the obtained result was even worse. However, it is too small of a difference to draw conclusions – worse result may be obtained due to randomization. The details are presented on the chart below.

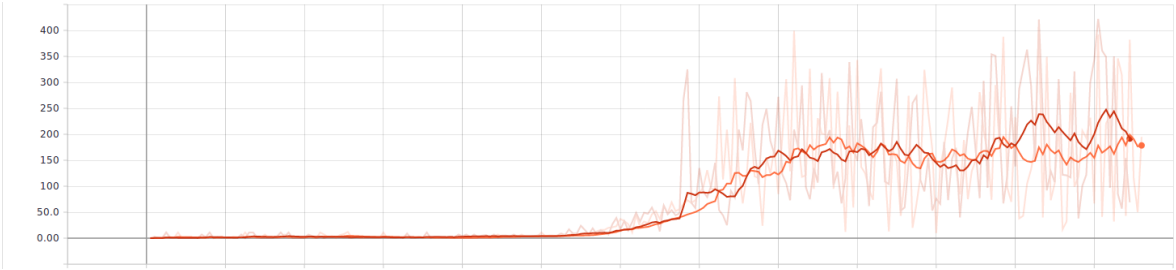

Figure 14. *Target DQN on BreakoutDeterministic-v4. The brown line is a plot from the setting with binary frame turned off, the orange line is a plot from the settings with binary frame turned on.*

4.7 REWARD CLIPPING

Another worth-mentioning difference is reward clipping. We decided to use it only on Breakout. As general as it is, it may be perceived as handcrafting features. Experiments conducted on Space Invaders showed that not only was the agent able to learn without reward clipping – it achieved better results. The example comparison of performance representing the difference between using and not using reward clipping is on the plot below.

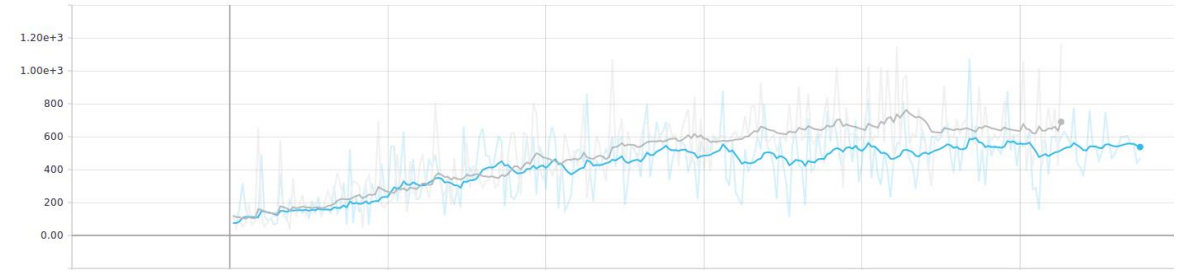

Figure 15. *Dueling DDQN on SpaceInvadersNoFrameskip-v4. The grey line is a plot from the setting with reward clipping turned off, the blue line is a plot from the settings with reward clipping applied.*

4.8 DISCUSSION

The comparison of results obtained by our implementations and those from DeepMind’s papers is present in Table 1 at the end of this thesis. Specifically, results from Target DQN indicate correctness of our implementation. Poor performance of Simple DQN in Breakout is thought-provoking. It may be caused by wrong optimizer’s parameters – the ones we used might be not the best ones. However, they worked with the other methods and tuning the optimizer is not part of this thesis. Comparison of the results obtained by DDQN and Dueling DDQN is artificial – in citied papers the training lasted 20 times longer (200M frames instead of 10M). Moreover, Prioritized Experience Replay and tuned hyperparameters were used. In light of this fact these results may be perceived as a curiosity. However, we run Double DQN on Space Invaders for the extended training session and the ongoing progress was observed. It indicates the possibility that with longer training session and tuned hyperparameters our agent would obtain similar results to those presented in papers.

Four of the introduced methods – Simple DQN, Target DQN, DDQN and Dueling DDQN – gave state-of-the-art results in the time of being published in the original papers.

5. SUMMARY

Our agents learned to successfully play two games – Breakout and Space Invaders. They learned interesting policies – in Breakout the agent is drilling a tunnel through blocks in order to let the ball bounce between blocks and the ceiling. In Space Invaders the agent hides behind the shield, dodges enemy's bullets by a margin and predicts when to fire its own in order to hit the target that is far away.

However, there is still room for improvements. The obvious ones include longer training sessions, experimenting with optimizer and tuning hyperparameters for a given architecture and algorithm. Regularization and dropout techniques may also be worth investigating. Moreover, it is advised to take a look on more recent papers such as [rainbow]. It combines the techniques mentioned in this thesis as well as the other ones into one powerful agent.

As intended, various solutions managed to learn useful policies using exactly the same hyperparameters. Nevertheless, the results leave us with the conclusion that much longer training sessions are required in order to examine the impact on performance of each modification.

Code and videos showing mentioned behavior are available on github[github].

6. REFERENCES

[dqn] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing atari with deep reinforcement learning. CoRR abs/1312.5602

[target-dqn] Mnih,V.;Kavukcuoglu,K.;Silver,D.;Rusu,A.A.;Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. Nature 518(7540):529–533.

[ddqn] van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double Q-learning. In Proc. of AAAI, 2094–2100.

[dueling] Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M.; and de Freitas, N. 2016. Dueling network architectures for deep reinforcement learning. In Proceedings of The 33rd International Conference on Machine Learning, 1995– 2003.

[gym] https://gym.openai.com/

[book] Sutton, R.; and Barto A. 2017. Reinforcement Learning: An Introduction. Second edition, in progress . Complete draft

[ucl-course] UCL Course on RL, David Silver, www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

[glorot] Glorot, X. and Bengio, Y. 2010. Understanding the difficulty of training deep feedforward neural networks. In Proc. AISTATS, volume 9, pp. 249–256

[github] https://github.com/Kropekk/value-based-methods-of-deep-reinforcement-learning

[rmsprop] www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[rainbow] Hessel, M.; Modayil, J.; van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M. and Silver, D. 2017. Rainbow: Combining improvements in deep reinforcement learning. arXiv preprint arXiv:1710.02298

	Breakout		Space Invaders	
	Our	Paper's	Our	Paper's
Simple DQN	12.0	225.0	760.0	1075.0
Target DQN	422.0	316.8	1270.0	1088.9
Double DQN	385.0	375.0	1250.0	3154.60
Dueling DQN	420.0	-	1270.0 (1720.0)	-
Dueling DDQN	430.0	345.3	1165.0	6427.3

Table 1. *Comparison of results obtained by our agent and those printed in the corresponding papers. Column "Our" shows the best reward obtained in one of the evaluations performed during training. Paper's Simple DQN scores come from "DQN Best" row in Table 1 present in [dqn]. They also present the highest obtained score. In Paper's Target DQN the result is equal to the highest average episode score (Extended Data Table 3 from [dqn-target]). The last three techniques are listed, but the comparison does not provide sufficient information to evaluate our implementation against those in the papers due to the hardware and time limitations.*