

Survey of Deep Q-Network variants in PyGame Learning Environment

Evalds Urtans
Riga Technical University
Riga, Latvia
+37126401317
evalds.urtans@rtu.lv

Agris Nikitenko
Riga Technical University
Riga, Latvia
+37129424825
agris.nikitenko@rtu.lv

ABSTRACT

Q-value function models based on variations of Deep Q-Network (DQN) have shown good results in many virtual environments. In this paper, we surveyed over 30 sub-algorithms that could influence the performance of DQN variants. We found important stability and repeatability aspects of state of art Deep Reinforcement Learning algorithms. We also developed Multi Deep Q-Network (MDQN) as a generalization of popular Double Deep Q-Network (DDQN) algorithm. Finally, using PyGame Learning Environment we produced visual representations of a learning process as Q-Value maps. Videos of trained models available in following link: <http://yellowrobot.xyz/mdqn>.

CCS CONCEPTS

- Theory of computation → Design and analysis of algorithms;
- Applied computing;

KEYWORDS

Deep Reinforcement Learning, Deep Learning, DQN, DDQN, MDQN

1 INTRODUCTION

This paper surveys many of the latest Deep Q-Learning algorithms in the field of Deep Reinforcement Learning. Notable examples of Deep Q-Learning (DQN) algorithms are the original DQN [12], Double Deep Q-Learning (DDQN) [19], Dueling Network [21] and asynchronous n-step DQN [11]. In addition to these Q-Value based algorithms, there are two other major branches of development in this field. One is policy gradient methods, with notable algorithms like Trust Policy Region Optimization (TRPO) [15] and Proximal Policy Optimization [17]. Another branch is combination of Q-Value and policy gradient models that is called actor-critic model with notable algorithms like Deep Deterministic Policy Gradient (DDPG) [10], Asynchronous Advantage Actor-Critic (A3C) [11], GPU A3C [2] and Actor-Critic with Experience Replay (ACER) [20]. In this paper, we focus only on Q-Value based algorithms. We studied how variations of these algorithms and hyper-parameters affect performance in PyGame Learning Environment (PLE) [18]. We also

proposed a generalization of the DDQN algorithm and extended it for use with 3 or more decoupled DQN models.

2 RELATED WORK

Recently some surveys have been conducted to assess a huge variety of Deep Reinforcement Learning algorithms [9], [3], [7]. Many Deep Reinforcement Learning algorithms suffer from large variance in results. There have been a number of papers trying to resolve this issue [1], [16]. Some research also point out problems with repeatability and identifies random seed as a significant factor that impacts results [8], [6].

3 METHODOLOGY

3.1 Deep Q-Network variants

All Q-function algorithms share underlying equations: the calculation of Cumulative Reward Equation 1 and the Bellman equation for modeling policy π through Q-function Equation 2 that is an approximation of a reward function for a given state s and action a at a time step t .

$$R = \sum_{t=0}^n \gamma^t r_t \quad (1)$$

$$Q_{\pi}(s_t, a_t) = r_t + \gamma \max_{a'} Q_{\pi}(s_{t+1}, a') \quad (2)$$

DQN algorithm relies on Equation 3 where a parametrized Q-function is based on a deep neural network. Usually, an input is a raw pixel representation trained by Convolution Neural Network (ConvNet) or a lower dimensionality representation of s . The model also usually utilize Recurrent Neural Network (RNN) like LSTM or GRU.

$$Q_{\Theta}(s_t, a_t) \leftarrow Q_{\Theta}(s_t, a_t) + \alpha (\nabla((r_t + \max_{a'} \gamma Q_{\Theta}(s_{t+1}, a') - Q_{\Theta}(s_t, a_t)))) \quad (3)$$

$$Q_{\Theta}(s_t, a_t) \leftarrow Q_{\Theta}(s_t, a_t) + \alpha (\nabla((r_t + \max_{a'} \gamma Q_{target}(s_{t+1}, a') - Q_{\Theta}(s_t, a_t)))) \quad (4)$$

DDQN algorithm is similar to DQN, but it utilizes theory from Double Q-Learning [5] by using two decoupled Q-functions like shown in Equation 4. Q_{target} function parameters are copied from Q_{Θ} with a given time step interval thereby achieving two decoupled Q-functions.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICDLT 2018, June 2018, Chongqing, China

© 2023 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

<https://doi.org/10.1145/nnnnnnnn.nnnnnnn>

3.2 Multiple Deep Q-Networks

In this research we found some differences of DDQN (Double Deep Q-Network via Target network) [19] and original DQL (Double Q-Learning) [5]. In case of DDQN Q_{target} is used as decoupled function whereas in pure DQL there should be Q_1 and Q_2 that are used intermittently. DDQN is simpler and should preserve same properties as pure DQL. We studied how this simplification impacts performance and implemented a pure version of DDQN and compared it with a standard DDQN. We also generalized DQL algorithm in order to use any number of decoupled functions in Bellman equation and call it MDQN (Multiple Deep Q-Network). MDQN with 2 decoupled functions is listed in Algorithm 1, but this could be easily expandable to more decoupled function pairs.

3.3 Other algorithmic improvements

We also made some algorithmic improvements that could be applied to other deep reinforcement learning algorithms. One of the improvements was to use the cumulative reward for training actions that were observed in an offline rollout of a episode. For example, if the offline state contains $\{s_t, a_t\}$ and calculated cumulative reward for $\{s_{t+1}, a_{t+1}\}$ it is possible to train the model using this value instead of Bellman equation. And when $\{s_t, a_t, s_{t+1}, a_{t+1}\}$ was not

observed in an episode we can use a value from Bellman equation. Idea are shown in in Algorithm 2.

In this research, we used RNN (Recurrent Neural Networks) as models of DQN variants. These models take as input observation from 5 previous frames. In order to speed up training we used RNN-ReLU instead of LSTM or GRU. We concluded that LSTM and GRU perform better than RNN-ReLU, but also take up to 7 times longer to train. We also implemented a label smoothing that prevents vanishing gradients in ReLU RNN [13].

We made all of our code used to test algorithms included in this paper open-source. Prioritized replay buffer is implemented as a separate library that could be used with a completely different set of reinforcement learning algorithms <https://github.com/evaldsurtans/dqn-prioritized-experience-replay>. It includes both types of prioritized replay buffer algorithms: proportional and ranked [14]. The main part of source code that contains variants of algorithms we tested is also available as an open-source project <https://bitbucket.org/evaldsurtans/dqn-research>. We implemented it in a way that we could utilize High-Performance Cluster (HPC) architecture where every sample of random seed is executed as a separate task on a node in a larger network. Each sample of random seed was a complete training of 10^7 frames with specified hyper-parameters.

Algorithm 1: MDQN (2 decoupled functions)

```

1: procedure TRAIN
2:   while Training = True do
3:     if random(0, 0, 1.0) < 0.5 then
4:       if  $s_t \neq$  terminal state then
5:          $Q_1(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_2(a, s_{t+1})$ 
6:       else
7:          $Q_1(a_t, s_t) \leftarrow R_t$ 
8:     else
9:       if  $s_t \neq$  terminal state then
10:         $Q_2(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_1(a, s_{t+1})$ 
11:      else
12:         $Q_2(a_t, s_t) \leftarrow R_t$ 
13:       $a_t \leftarrow \max_a \text{average}(\{Q_1(a, s_t), Q_2(a, s_t)\})$ 
14:      ...

```

Algorithm 2: Offline MDQN with a cumulative reward boost

```

1: procedure TRAIN
2:   while Training = True do
3:     for do  $\{a_t, s_t, s_{t+1}\}$  sample from ReplayBuffer
4:       if  $\{a_t, s_t, s_{t+1}\}$  in ReplayBuffer then
5:         if  $\text{random}(0, 0, 1.0) < 0.5$  then
6:            $Q_1(a_t, s_t) \leftarrow \sum_{t=0}^{t+1} \gamma^t R_t$ 
7:         else
8:            $Q_2(a_t, s_t) \leftarrow \sum_{t=0}^{t+1} \gamma^t R_t$ 
9:       else
10:        if  $\text{random}(0, 0, 1.0) < 0.5$  then
11:          if  $s_t \neq \text{terminal state}$  then
12:             $Q_1(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_2(a, s_{t+1})$ 
13:          else
14:             $Q_1(a_t, s_t) \leftarrow R_t$ 
15:        else
16:          if  $s_t \neq \text{terminal state}$  then
17:             $Q_2(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_1(a, s_{t+1})$ 
18:          else
19:             $Q_2(a_t, s_t) \leftarrow R_t$ 
20:        while  $s_t \neq \text{terminal state}$  do
21:           $a_t \leftarrow \max_a \text{average}(\{Q_1(a, s_t), Q_2(a, s_t)\})$ 
22:          ...
23:        store  $\{a_t, s_t, s_{t+1}, r_t\}$  in ReplayBuffer

```

4 EXPERIMENTS

4.1 PyGame Learning Environment

In order to evaluate results, we used the open-source game environments from the "PyGame Learning Environment" (PLE) <https://github.com/ntasfi/PyGame-Learning-Environment>. PLE contains many different games including Flappy Bird, 3D Maze, Doom, and others. For most of the game environments it is possible to get low dimensional representations of a state, which are useful for testing deep reinforcement algorithms with limited computational resources. Of course, it is also possible to train agents using high dimensional pixel representations of a state. Another very desirable feature is that game environments can be manipulated while running because full source code for each game is easily accessible. We implemented curriculum learning for the 3D raycast maze, where target moves away from starting point in later stages of training. We also implemented a way to produce Q-value map (Q-map) by manipulating a position of a game character in an environment and getting Q-value for every artificial state in a game. For example, in a game of flappy bird, we moved the bird across all pixels in a frame and calculated Q-function value that we overlaid as a heat map like in Fig. 4. This kind of representation helps to understand what DQN model has learned. In fact, we found and fixed a bug in a Flappy Bird environment by using Q-map when we noticed that DQN model learned to cross an obstacle over the top of the screen. In case of 3D raycast maze, we implemented Q-map by teleporting a player to all walkable squares and rotating incrementally player's camera around the center of each square. For every frame, it is possible to calculate average Q-Value of all actions available and then make a heat map of a maze like in Fig. 8.

4.2 Random seed and repeatability

In our research, we came across problem that all DQN, DDQN and MDQN variants are very sensitive to seed randomization. We implemented a way to restore all random seeds in order to repeat results, but this is actually not desirable because it can lead to misleading results when comparing different hyper-parameters. A better approach is to increase sample size of random seeds. This means that every training configuration should be rerun multiple times with different randomization seeds as shown in Fig. 1. We also noticed large variance between different samples of random seed. In order to make accurate comparisons, we chose a random seed size of 10, since we observed that this resulted in similar variances to sample sizes 20 and 40. Whereas using a sample size of 5 produced a much lower variance of results. We had quite limited computing resources and even random seed size of 10 took considerable time to test. It is one of the reasons why we chose experimentally initial hyper-parameter values that we changed one by one, instead of performing full grid search.

Often it is advised to reduce variance by reducing the model complexity [4]. Our results confirm this hypothesis Fig. 2, however by reducing model complexity also a maximal average score of testing set reduces as well. When constructing such models we would advise to find the compromise between model complexity, repeated random seed test set size and a variance.

Another widely used method to reduce variance is to use regularization. Again our results confirm that it reduces variance, but again it also reduces average scores as shown in Fig. 3. As for batch normalization, we also found no improvement in an average score as shown in results in an appendix.

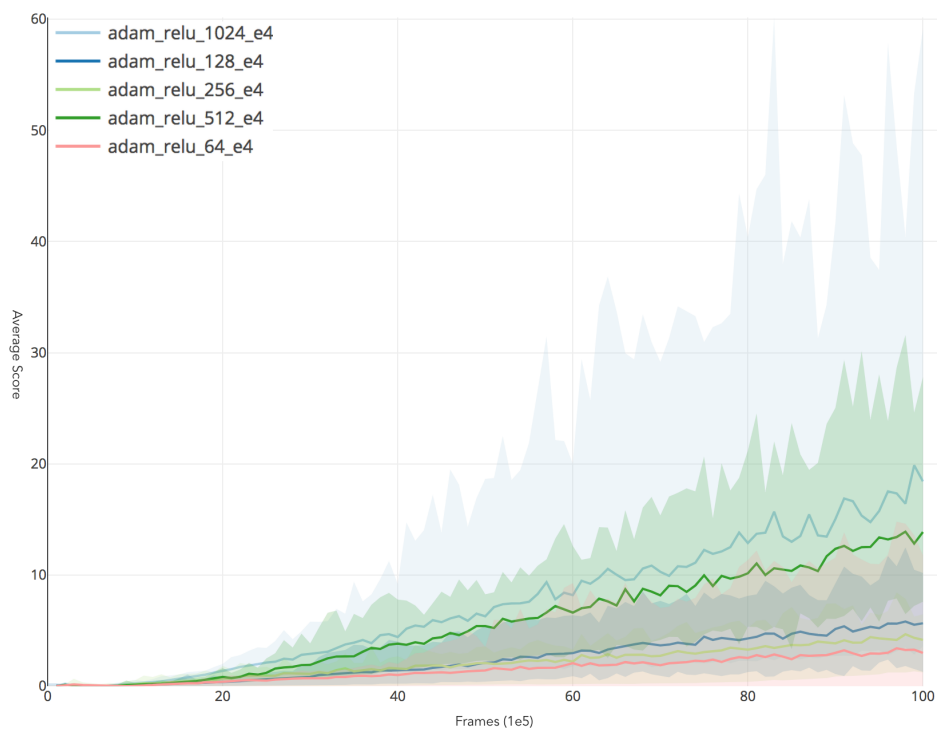


Fig. 2: Comparison of different hidden unit vector sizes. Average score for 10^7 frames.

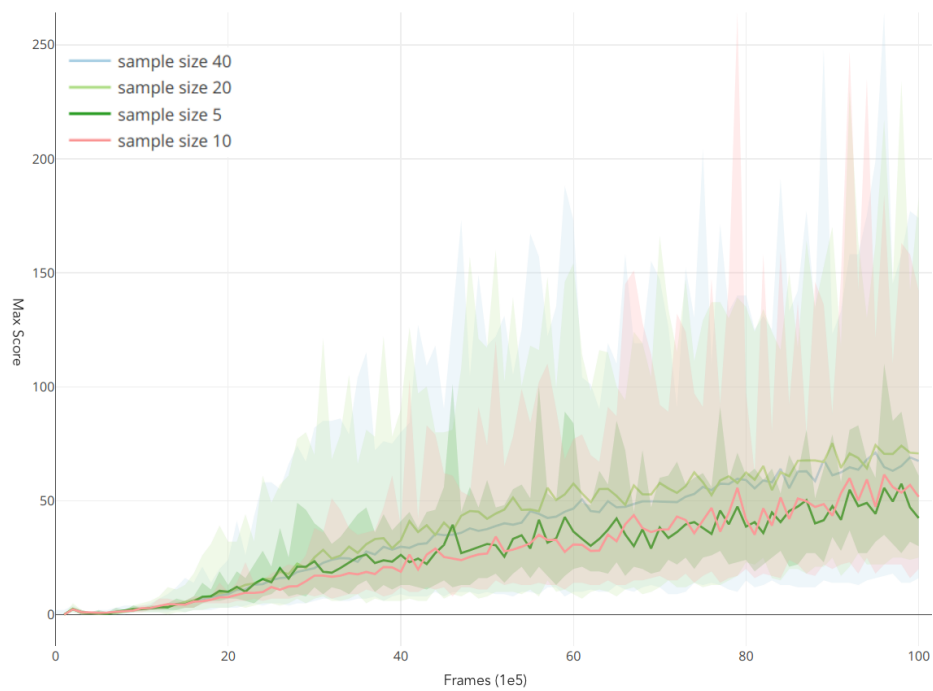


Fig. 1: Sample size of random seeds and variance of average score for 10^7 frames.

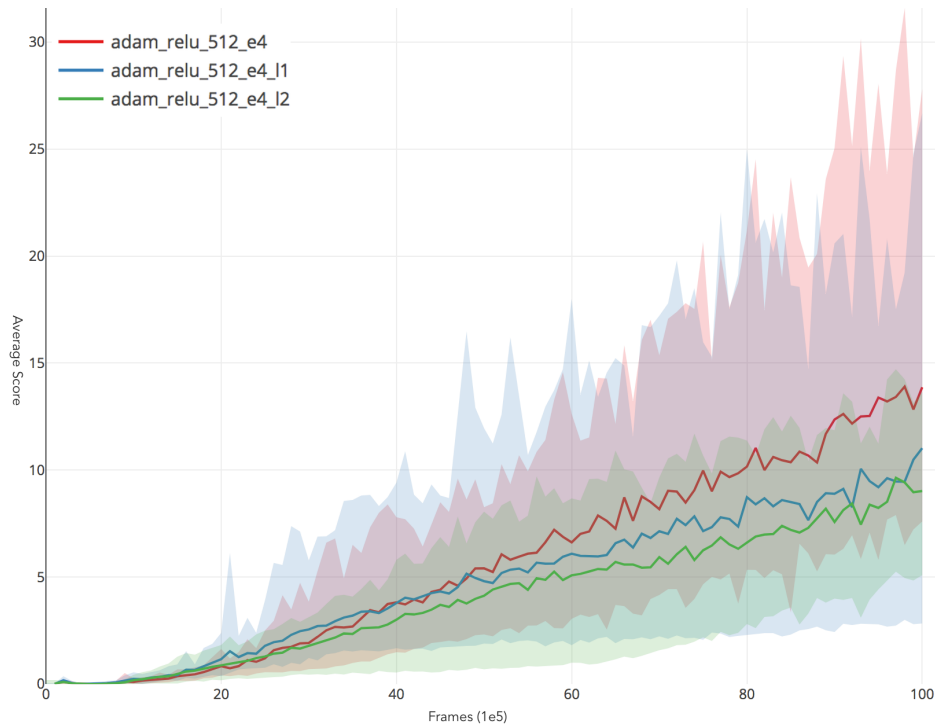


Fig. 3: Effect of L1 (Lasso) and L2 (Ridge) regularization. Average score for 10^7 frames.

Table 1: Initial hyper-parameters that other parameters were measured against in all environments

parameters	
batch norm: false	mini-batch: 32
bellman gamma: 0.99	model: 1 states to n actions
beta replay buffer: true	offline prebatch: false
cumulative reward: true	online: false
diff. states: false	optimizer: rmsprop
dropout: 0.0	pixels input: none
dueling arch.: false	priority replay buffer: ranked
epsilon greedy: true	reg.: none
epsilon start-end: $1e-3 - 1e-6$	replay buffer: $5e5$
epsilon stuck: false	rnn: relu
extra frame reward: $1e-5$	sarsa: false
frames back: 5	state prev. act. reward: false
frames before: $5e4$	target network alpha: 1.0
grad clip.: 0.0	terminal reward: $-1e3$

4.3 Flappy Bird

Initially to test more than 28 hyper-parameters of DQN variants we ran many partial grid searches on combinations of parameters. Then we did benchmarking for one step changes in each of hyper-parameters against initial parameters that are shown in Table 1. Each set of parameters were repeated for at least 10 times to ensure repeatability as described in subsection 4.2. By run, we mean full training of 10^7 frames with a defined set of hyper-parameters. We used ReLU RNN as Q-value model in order to speed up training. As seen in Fig. 5 more appropriate RNN architectures perform better, but takes much longer to train.

We compared DQN, DDQN and MDQN algorithms with full set of hyper-parameters as shown in Table 2, Table 3, Table 4 and Table 5. We conclude that classic DQN outperforms DDQN and MDQN, but our version of MDQN slightly outperform DDQN. This is nothing particularly surprising that DQN outperforms more advanced

DDQN and MDQN because in previous studies it has also been shown that different algorithms excel in different environments. In some environments, DQN is more effective, but in others DDQN. We also could not see significant improvements by applying some more interesting architectures like Dueling Network or different activation functions in RNN like Leaky ReLU, ELU and PreLU.

We also found out that none of the regularization methods such as L1, L2, Dropout or Batch Normalization didn't improve performance. This could be the case because huge data set that is gathered from training environment in itself accomplishes normalization [4].

Because of the flexibility of open-source environments in PLE we managed to produce Q-Value maps to track and compare the progress of different sets of hyper-parameters. An example of Q-Value maps is given in Fig. 4.

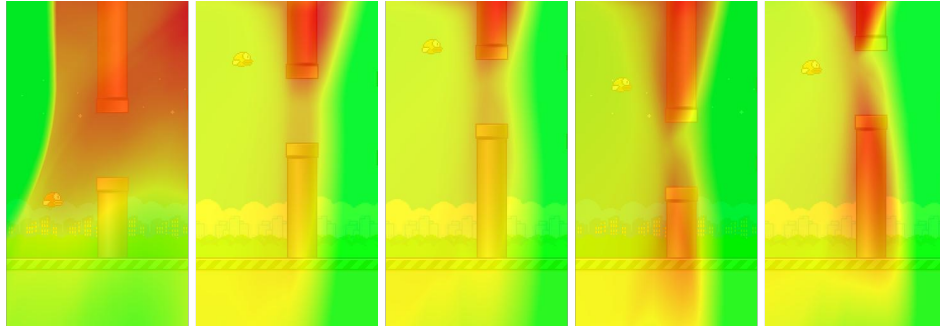


Fig. 4: Q-Maps of sequential training of Flappy Bird from beginning on left till 10^7 frame on right. Green is highest value state. Red is lowest value state.

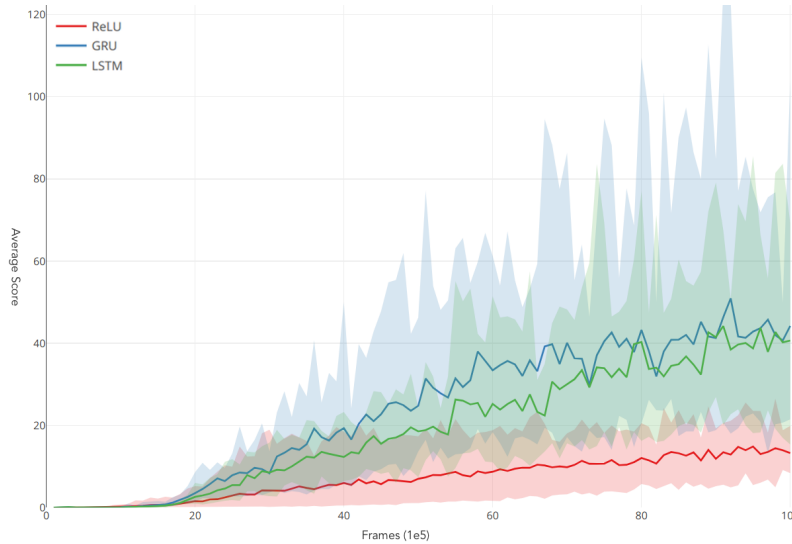


Fig. 5: Comparison of RNN types for Flappy Bird environment. Average score for 10^7 frames (sample size of random seeds: 10)

Table 2: Top 20 hyper-parameters of DQN for Flappy Bird environment

parameters	lr	avg. score	max. score	var. score	time (min.)
rnn: gru	0.0001	41.95876824	264.1	289.0515166	2829.558
rnn: lstm	0.0001	38.19286308	264.1	96.85448042	3637.49
optimizer: adam	0.001	28.78679352	264.1	303.0517411	500.211
bellman gamma: 0.90	0.0001	19.36507549	264.1	114.9246432	443.704
epsilon greedy: false	0.001	15.05309331	264.1	42.61359982	455.75
default	0.001	13.17070586	175.06628	19.87366074	592.704
bellman gamma: 0.99	0.001	13.17070586	175.06628	19.87366074	592.704
beta replay buffer: true	0.001	13.17070586	175.06628	19.87366074	592.704
cumulative reward: true	0.001	13.17070586	175.06628	19.87366074	592.704
beta replay buffer: false	0.001	12.89863154	183.06919	12.74031999	459.317
frames back: 10	0.0001	12.67872818	230.08696	9.432428545	715.232
extra frame reward: 0.0	0.0001	12.62185465	264	65.53895774	453.495
grad clip.: 1.0	0.001	12.45487645	186.07046	19.81659023	481.066
grad clip.: 10.0	0.0001	12.12482097	223.08459	37.59049115	478.056
target network alpha: 0.5	0.001	12.07657609	227.08608	30.91554801	644.119
rnn: elu	0.0001	12.02852927	215.08156	41.96880395	380.207
default	0.0001	11.91111497	264.1	31.95937102	527.8
frames before: 1e5	0.0001	11.72732532	149.0567	15.32392026	439.196
extra frame reward: 0.0	0.001	11.46642777	238	24.56193688	456.816
bellman gamma: 0.90	0.001	11.39091096	264.1	130.1593219	421.819

Table 3: Top 20 hyper-parameters of DDQN for Flappy Bird environment

parameters	lr	avg. score	max. score	var. score	time (min.)
rnn: gru	0.0001	42.97024986	264.1	5.49E+02	2603.809
rnn: lstm	1.00E-04	28.60916534	264.1	9.92E+01	3246.475
optimizer: adam	1.00E-03	16.96737049	264.1	3.69E+01	351.827
grad clip.: 1.0	0.001	12.45431387	254.096	1.90E+01	399.956
optimizer: adam	1.00E-04	10.19414131	207.07831	2.17E+01	367.396
grad clip.: 10.0	0.0001	9.765907544	184.06991	1.92E+01	361.19
grad clip.: 10.0	0.001	8.694875819	156.05916	1.28E+01	364.509
grad clip.: 1.0	0.0001	7.633139692	140.05331	1.71E+01	357.085
rnn: lstm	1.00E-03	6.614351436	182.06913	2.15E+01	3770.588
optimizer: adam	1.00E-05	1.929185929	47.0181	1.81E+00	375.648
mini-batch: 8	1.00E-05	1.922057009	40.0155	4.14E-01	505.778
bellman gamma: 0.90	1.00E-05	1.916720841	47.01799	1.19E+00	366.701
grad clip.: 10.0	1.00E-05	1.783358575	32.01237	3.22E-01	400.212
beta replay buffer: false	1.00E-05	1.647591408	60.02323	1.02E+00	423.562
default	1.00E-05	1.533690858	32.01236	3.08E-01	377.689
rnn: relu	1.00E-05	1.533690858	32.01236	0.308035851	377.689
grad clip.: 1.0	1.00E-05	1.486492849	22.00888	4.45E-01	388.789
dropout: 0.1	1.00E-05	1.466420627	36.01396	6.52E-01	394.784
epsilon greedy: false	1.00E-05	1.437370177	45.01724	0.20441139	406.223
model: n states to n act.	1.00E-05	1.412973236	34.01324	0.90539683	595.273

Table 4: Top 20 hyper-parameters of MDQN3 for Flappy Bird environment

parameters	lr	avg. score	max. score	var. score	time (min.)
rnn: gru	0.0001	24.6588361	264.1	70.62906959	2735.039
rnn: lstm	0.0001	16.17153479	224.08495	31.69684243	3041.107
optimizer: adam	0.001	12.14972485	148.05634	8.572064894	357.792
rnn: lstm	0.001	6.362237775	161.06122	10.33771355	3052.423
grad clip.: 10.0	0.001	6.148186995	130.0494	11.12354631	380.245
grad clip.: 10.0	0.0001	5.774537436	104.03959	11.92569845	361.402
optimizer: adam	0.0001	5.541812022	124.0473	18.46937444	379.416
grad clip.: 1.0	0.0001	4.706386259	119.04543	11.58234856	374.33
grad clip.: 1.0	0.001	2.676778874	65.02486	4.486235409	456.614
rnn: gru	0.001	1.117617436	64.02454	1.294722902	2537.775
target network alpha: 0.0	1.00E-05	1.047481234	19.00759	0.298771101	745.303
model: n states to n act.	1.00E-05	0.881531711	12.00483	0.101958401	600.168
epsilon stuck: true	1.00E-05	0.878962391	11.00452	0.013849234	384.174
grad clip.: 10.0	1.00E-05	0.86220963	21.00828	0.183416314	402.766
mdqn: min	1.00E-05	0.740644674	10.0041	0.05608085	376.083
mini-batch: 8	1.00E-05	0.719492781	9.0037	0.091634393	512.362
epsilon start end: 1e-3 1e-3	1.00E-05	0.704522112	10.0042	0.022748512	398.203
bellman gamma: 0.80	1.00E-05	0.667448615	35.01376	0.158929947	381.275
epsilon start end: 1e-1 1e-6	1.00E-05	0.665939117	15.00605	0.167286059	392.187
optimizer: adam	1.00E-05	0.659281593	10.00418	0.180768865	389.655

Table 5: Comparison of DQN, DDQN and MDQN models for Flappy Bird environment. Decimal number after abbreviation like mdqn3 1.0 denotes coefficient of target network. Coefficient 0.0 denotes that algorithm do not use target network.

model type	lr	avg. score	max. score	var. score	time (min.)
dqn 1.0	0.001	28.78679352	264.1	303.0517411	500.211
mdqn2 1.0	0.001	17.19567935	264.1	50.58413201	421.452
ddqn 1.0	1.00E-03	16.96737049	264.1	3.69E+01	351.827
mdqn2 0.0	0.001	14.07828206	212.08043	18.74729045	493.78
mdqn3 1.0	0.001	12.14972485	148.05634	8.572064894	357.792
mdqn3 0.0	0.001	9.328698486	179.06784	24.94178454	635.494
mdqn2 0.0	0.0001	9.311841471	202.07645	13.39635414	521.696
mdqn2 1.0	0.0001	5.351493407	127.04827	14.07459022	384.702
mdqn3 0.0	0.0001	4.406378303	102.03882	5.088549631	776.327
mdqn2 0.0	1.00E-05	1.603283236	61.02341	0.715069292	642.106
mdqn3 0.0	1.00E-05	0.872432773	12.00487	0.087176378	713.332
mdqn2 1.0	1.00E-05	0.692394861	12.00513	0.221167891	389.493

4.4 Pong

For Pong and 3D raycast maze environments, we changed in initial hyper-parameters optimizer from "rmsprop" to "adam", because it gave better results without increasing processing time. In case of Pong again DQN slightly outperformed MDQN and DDQN, but MDQN slightly outperformed DDQN as shown in Fig. 7 and Table 7. We also produced Q-Value maps by manipulating position of ball in Pong environment on frozen Q-Value model at checkpoints during training as shown in Fig. 6.

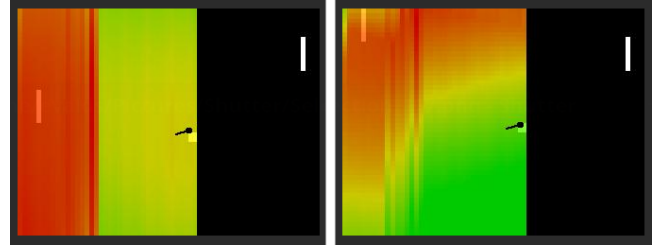


Fig. 6: Pong Q-map before and after training. After training possible to see clear path of projected ball trajectory.

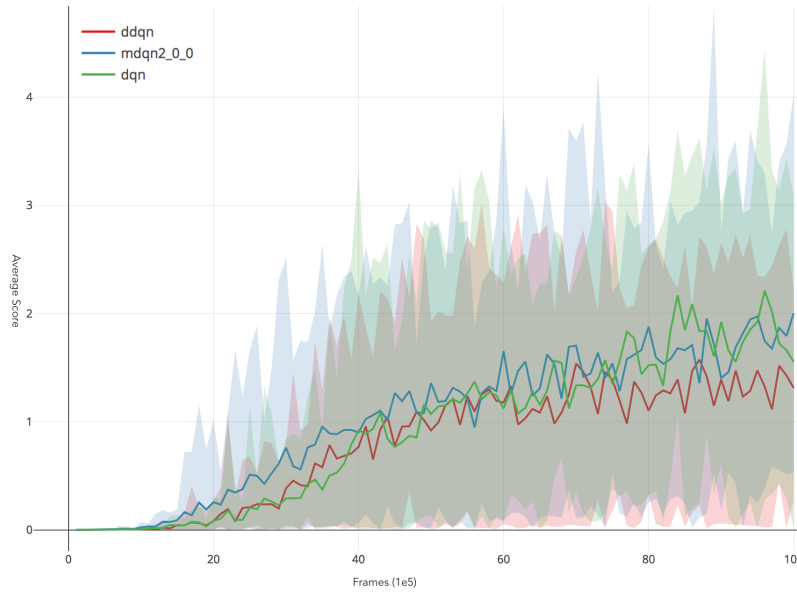


Fig. 7: Comparison between model types for Pong environment. Average score for 10^7 frames.

Table 6: Comparison of DQN, DDQN and MDQN models for Pong environment. Decimal number after abbreviation like mdqn3 1.0 denotes coefficient of target network. Coefficient 0.0 denotes that algorithm do not use target network.

model type	lr	avg. score	max. score	var. score	time (min.)
dqn 1.0	0.001	1.794835114	6.09985	0.689502582	303.509
mdqn2 0.0	0.001	1.707382894	6.09998	0.712970429	478.007
mdqn2 1.0	0.001	1.624445803	6.09998	1.162839194	252.519
mdqn3 1.0	0.001	1.398198348	6.09906	0.879391948	254.385
ddqn 1.0	0.001	1.33096317	6.09998	0.341259324	288.435
mdqn3 0.0	0.001	0.945720479	6.09998	1.587272644	572.729
dqn 1.0	0.0001	0.772168575	6.09814	0.365215527	247.754
ddqn 1.0	0.0001	0.534693025	6.09998	0.324040409	266.406
mdqn2 1.0	0.0001	0.384394197	6.09267	0.12705127	255.21
mdqn2 0.0	0.0001	0.156203775	6.09631	0.008177064	362.899
mdqn3 0.0	0.0001	0.146223293	6.07963	0.009221072	666.229
mdqn3 1.0	0.0001	0.068661735	6.0579	0.0040051	266.283
dqn 1.0	1.00E-05	0.018227918	6.02963	0.00013185	271.978
ddqn 1.0	1.00E-05	0.017447873	6.01034	0.000257357	274.989
mdqn2 0.0	1.00E-05	0.01239353	6.02133	2.66E-05	392.133
mdqn3 1.0	1.00E-05	0.0089922	6.01675	9.56E-05	279.051
mdqn2 1.0	1.00E-05	0.007004221	6.01711	1.32E-05	286.278
mdqn3 0.0	1.00E-05	0.004662119	6.01216	3.99E-06	603.306

Table 7: Comparison of DQN, DDQN and MDQN models for 3D Raycast maze environment.

model type	lr	avg. score	var. score	time (min.)
mdqn2 0.0	1.00E-05	3.904359232	0.728045918	1213.991
dqn	1.00E-05	3.88654262	2.124993494	484.859
mdqn2 1.0	1.00E-06	3.7166532	0.154117942	533.389
ddqn	1.00E-06	3.713829593	1.524318234	524.65
ddqn	1.00E-05	3.638360789	1.662039807	521.975
mdqn2 0.0	1.00E-05	3.506246831	1.746571203	809.374
mdqn2 0.0	1.00E-06	3.345749731	2.749636472	978.638
ddqn	0.0001	3.267777864	2.889255991	523.012
mdqn2 1.0	1.00E-07	3.247272282	0.576931468	500.424
mdqn2 1.0	1.00E-05	3.180342964	2.016163812	523.085
mdqn3 1.0	1.00E-05	3.056116361	2.339890159	872.317
dqn	1.00E-06	3.026868771	2.028895348	534.022
mdqn3 0.0	1.00E-05	2.807473511	2.395394139	1212.21
mdqn3 1.0	1.00E-06	2.770128326	0.714132328	864.442
mdqn3 0.0	1.00E-06	2.629530288	1.724929361	1152.146
mdqn2 0.0	0.0001	2.545370799	4.120312752	623.82
dqn	0.0001	2.24425396	2.153779645	516.078
mdqn3 1.0	0.0001	2.174641347	3.541216037	775.408
mdqn2 0.0	0.0001	2.157170755	3.235455294	899.93
mdqn2 1.0	0.0001	1.959047125	2.688871288	479.06
mdqn3 0.0	0.0001	1.678048035	3.272271359	1117.217
mdqn2 0.0	1.00E-06	1.452786487	1.887540531	809.63
mdqn2 1.0	1.00E-08	1.399096696	1.827157866	495.813
mdqn2 0.0	1.00E-07	0.178030303	0	463.67

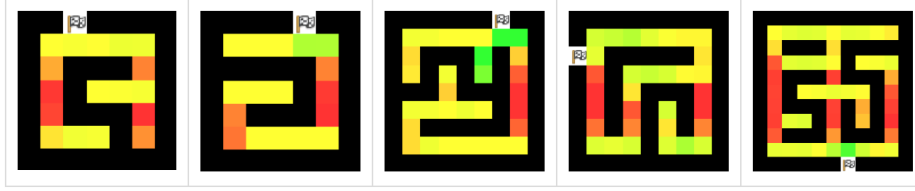


Fig. 8: 3D Raycast maze Q-map for each position in map from top down view. Each Q-map represents sequential frame checkpoint during training. On left first frame and on right 10^7 frame. Notice that map increases in size thus using curriculum learning principle.

4.5 3D Raycast maze

Finally, we benchmarked algorithms on "3D Raycast Maze" environment where instead of a low dimensional representation of a state, we used RGB 48x48 pixel input. In many environments in order to save resources pixel grayscale representation would be recommended, but to make sure that in doors have a distinguishable difference in color from walls we chose to use only two channels per pixel red and green. The model consisted of ConvNet embedding and RNN layers. All pixel inputs were normalized in a range $0.0 - 1.0$ instead of using byte value of $0 - 255$.

After the model has been trained we generated GradCAM map to visualize highest gradients in ConvNet as shown in Fig. 9. These maps are more informative than Saliency Maps used in other Deep Reinforcement Learning papers [21]. These maps help us to understand what part of input pixel array is the most important for training. In this case, it was the exit door that gives the reward when reached.

Another way to reduce dimensionality of problem was to remove some of actions available to an agent. We allowed agent only to move ahead and make turns left and right, but not to go back and wait (do nothing).

We again constructed Q-Value maps to visualize the progress of learning as shown in Fig. 8. In this case, we manipulated a position of player around the maze and recorded Q-Values by rotating player's view around this position.

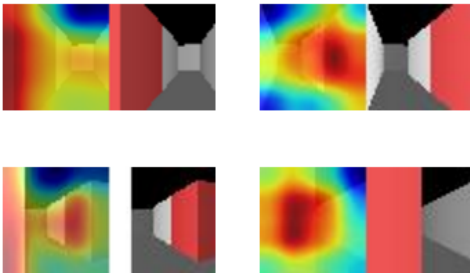


Fig. 9: GradCAM Saliency maps of trained MDQN agent in 3D Raycast maze environment. Images show attention on target in a 3D maze.

5 CONCLUSIONS

We introduced a new Deep Reinforcement Learning algorithm MDQN that slightly outperforms DDQN in some environments. Still in others DQN could work better than MDQN or DDQN. Most of DQN variants that we tested have little or no significant effect on performance. We also introduced a new way to construct Q-Value maps by manipulating training environment. Q-Value maps are useful for assessing the progress of training. We also conclude that it is essential to run a sufficient number of repeated training runs for every set of parameters, because of the impact of random seed initialization and large variance in results.

ACKNOWLEDGMENTS

Research has been completed with a support from High-Performance Computing Center of Riga Technical University.

APPENDIX

In this paper we have included heat-map of average score in a game of Flappy Bird after 10^7 frames for each hyper-parameter and studied learning rates. All hyper-parameters have been studied for DQN, DDQN and MDQN variants of algorithms. Results are available here: <http://yellowrobot.xyz/full-survey-flappybird.pdf>

REFERENCES

- [1] Oron Anschel, Nir Baram, and Nahum Shimkin. [n. d.]. Deep Reinforcement Learning with Averaged Target DQN. ([n. d.]). arXiv:1611.01929 <http://arxiv.org/abs/1611.01929>
- [2] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. [n. d.]. GA3C: GPU-Based A3C for Deep Reinforcement Learning. ([n. d.]). <http://arxiv.org/abs/1611.06256>
- [3] Yan Duan, Xi Chen, Rein Houthoofd, John Schulman, and Pieter Abbeel. [n. d.]. Benchmarking Deep Reinforcement Learning for Continuous Control. 48 ([n. d.]), 1329–1338. arXiv:1604.06778 <http://arxiv.org/abs/1604.06778>
- [4] A. Geron. [n. d.]. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media. <https://books.google.lv/books?id=bRpYDgAAQBAJ>
- [5] Hado V. Hasselt. [n. d.]. Double Q-Learning. In *Advances in Neural Information Processing Systems* 23, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta (Eds.). Curran Associates, Inc., 2613–2621. <http://papers.nips.cc/paper/3964-double-q-learning.pdf>
- [6] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. [n. d.]. Deep Reinforcement Learning That Matters. arXiv:1709.06560 <http://arxiv.org/abs/1709.06560> <https://arxiv.org/pdf/1709.06560.pdf>
- [7] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. [n. d.]. Rainbow: Combining Improvements in Deep Reinforcement Learning. abs/1710.02298 ([n. d.]). arXiv:1710.02298 <http://arxiv.org/abs/1710.02298>

- [8] Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. [n. d.]. Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. ([n. d.]). arXiv:1708.04133 <http://arxiv.org/abs/1708.04133>
- [9] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. [n. d.]. A Brief Survey of Deep Reinforcement Learning. ([n. d.]). <https://arxiv.org/abs/1708.05866>
- [10] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. [n. d.]. Continuous Control with Deep Reinforcement Learning. ([n. d.]). <http://arxiv.org/abs/1509.02971>
- [11] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. [n. d.]. Asynchronous Methods for Deep Reinforcement Learning. 48 ([n. d.]), 1928–1937. <http://arxiv.org/abs/1602.01783>
- [12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fiedland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. [n. d.]. Human-Level Control through Deep Reinforcement Learning. 518, 7540 ([n. d.]), 529–533. <http://dx.doi.org/10.1038/nature14236>
- [13] David Pfau and Oriol Vinyals. [n. d.]. Connecting Generative Adversarial Networks and Actor-Critic Methods. ([n. d.]). arXiv:1610.01945 <http://arxiv.org/abs/1610.01945>
- [14] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. [n. d.]. Prioritized Experience Replay. ([n. d.]). arXiv:1511.05952 <http://arxiv.org/abs/1511.05952>
- [15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. [n. d.]. Trust Region Policy Optimization. ([n. d.]), 1889–1897. <http://arxiv.org/abs/1502.05477>
- [16] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. [n. d.]. High-Dimensional Continuous Control Using Generalized Advantage Estimation. ([n. d.]). arXiv:1506.02438 <http://arxiv.org/abs/1506.02438>
- [17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. [n. d.]. Proximal Policy Optimization Algorithms. abs/1707.06347 ([n. d.]). arXiv:1707.06347 <http://arxiv.org/abs/1707.06347>
- [18] Norman Tasfi. [n. d.]. PyGame Learning Environment. ([n. d.]). <https://github.com/ntasfi/PyGame-Learning-Environment>
- [19] Hado van Hasselt, Arthur Guez, and David Silver. [n. d.]. Deep Reinforcement Learning with Double Q-Learning. 13 ([n. d.]), 2094–2100. <http://arxiv.org/abs/1509.06461>
- [20] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. [n. d.]. Sample Efficient Actor-Critic with Experience Replay. ([n. d.]). arXiv:1611.01224 <http://arxiv.org/abs/1611.01224>
- [21] Ziyu Wang, Nando de Freitas, and Marc Lanctot. [n. d.]. Dueling Network Architectures for Deep Reinforcement Learning. 16 ([n. d.]), 1995–2003. <http://arxiv.org/abs/1511.06581>