Survey of Deep Q-Network variants in PyGame Learning Environment

Evalds Urtans
Riga Technical University
Kalku iela 1
Riga, Latvia
+371 26401317
evalds.urtans@rtu.lv

Agris Nikitenko
Riga Technical University
Kalku iela 1
Riga, Latvia
+371 26401317
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, over 30 sub-algorithms were surveyed that influence the performance of DQN variants. Important stability and repeatability aspects of state of art Deep Reinforcement Learning algorithms were found. Multi Deep Q-Network (MDQN) as a generalization of popular Double Deep Q-Network (DDQN) algorithm was developed. Visual representations of a learning process as Q-Value maps were produced using PyGame Learning Environment. 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 [13], Double Deep Q-Learning (DDQN) [5], Dueling Network [23] and asynchronous n-step DQN [14]. 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) [17] and Proximal Policy Optimization [19].

Another branch is a combination of Q-Value and policy gradient models that are called actor-critic model with notable algorithms like Deep Deterministic Policy Gradient (DDPG) [12], Asynchronous Advantage Actor-Critic (A3C) [14], GPU A3C [2] and Actor-Critic with Experience Replay (ACER) [22]. This paper focuses only on Q-Value function based algorithms.

ICDLT '18, June 27–29, 2018, Chongqing, China
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6473-7/18/06…$15.00
https://doi.org/10.1145/3234804.3234816

The paper explores how variations of these algorithms and hyper-parameters affect performance in PyGame Learning Environment (PLE) [21]. In this paper generalization of the DDQN algorithm and extension is proposed. It can use 2, 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 [11], [3], [9].

Many Deep Reinforcement Learning algorithms suffer from large variance in results. There have been a number of papers trying to resolve this issue [1], [18].

Some research also points out problems with repeatability and identifies random seed as a significant factor that impacts results [10], [8].

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.

$$R = \sum_{t=0}^{n} \gamma^t r_t \tag{1}$$

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

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/CNN) or a lower dimensionality representation of s. The model also usually utilizes Recurrent Neural Network (RNN) like LSTM or GRU.

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

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

DDQN algorithm is similar to DQN, but it utilizes theory from Double Q-Learning [7] 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.

3.2 Multiple Deep Q-Networks

There are some differences of DDQN (Double Deep Q-Network via Target network) [6] and original DQL (Double Q-Learning) [7]. 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.

The paper explores this simplification impacts performance and implemented a pure version of DDQN and compared it with a standard DDQN. DQL algorithm was generalized 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 expendable to more decoupled function pairs.

Algorithm 1: MDQN (2 decoupled functions)

procedure Train
  while Training == True **do**
    if random(0.0, 1.0) < 0.5 **then**
      if $s_t \neq$ terminal state **then**
        $Q_1(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_2(a, s_{t+1})$
      else
        $Q_1(a_t, s_t) \leftarrow R_t$
    else
      if $s_t \neq$ terminal state **then**
        $Q_2(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_1(a, s_{t+1})$
      else
        $Q_2(a_t, s_t) \leftarrow R_t$
    $a_t \leftarrow max_a average(\{Q_1(a, s_t), Q_2(a, s_t)\})$
    …

3.3 Other algorithmic improvements

Some algorithmic improvements have been made that can 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 $\{s_{t+1}, a_{t+1}\}$ then it is possible to train the model using cumulative reward value instead of Bellman equation. And when $\{s_t, a_t, s_{t+1}, a_{t+1}\}$ is not observed in an episode it is possible to use a value from Bellman equation. Principle is shown in in Algorithm 2.

In this research, RNN (Recurrent Neural Networks) were used as models of DQN variants. These models take as input observation from 5 previous frames. To speed up training, RNN-ReLU were used instead of LSTM or GRU. LSTM and GRU perform better than RNN-ReLU, but also take up to 7 times longer to train. Label smoothing were implemented to prevent vanishing gradients in RNN-ReLU [15].

All source code used to test algorithms in this paper is open-source. Prioritized replay buffer is implemented as a separate library that can be used with a completely different set of reinforcement learning algorithms[1]. It includes both types of prioritized replay buffer algorithms: proportional and ranked [16].

The main part of source code that contains variants of algorithms that were tested and is also available as an open-source project[2]. Code was implemented it in a way that it could utilize High-Performance Cluster (HPC) architecture. Every training using sample of random seed were executed as a separate task on a node in a cluster. Each sample of random seed was a complete training of 10^7 frames with specified hyper-parameters.

Algorithm 2: MDQN with a cumulative reward boost

procedure Train
  while Training == True **do**
    // Offline variant of an algorithm
    while $s_t \neq$ terminal state **do**
      $a_t \leftarrow max_a average(\{Q_1(a, s_t), Q_2(a, s_t)\})$
      …
      store $\{a_t, s_t, s_{t+1}, r_t\}$ in $ReplayBuffer$

    for $\{a_t, s_t, s_{t+1}\}$ sample from $ReplayBuffer$ **do**
      $a'_t \leftarrow max_a average(\{Q_1(a, s_t), Q_2(a, s_t)\})$
      if $\{a'_t, s_t, s_{t+1}\}$ in $ReplayBuffer$ **then**
        if random(0.0, 1.0) < 0.5 **then**
          $Q_1(a_t, s_t) \leftarrow \sum_{t=0}^{t+1} \gamma^t R_t$
        else
          $Q_2(a_t, s_t) \leftarrow \sum_{t=0}^{t+1} \gamma^t R_t$
      else
        if random(0.0, 1.0) < 0.5 **then**
          if $s_t \neq$ terminal state **then**
            $Q_1(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_2(a, s_{t+1})$
          else
            $Q_1(a_t, s_t) \leftarrow R_t$
        else
          if $s_t \neq$ terminal state **then**
            $Q_2(a_t, s_t) \leftarrow R_t + \gamma \max_a Q_1(a, s_{t+1})$
          else
            $Q_2(a_t, s_t) \leftarrow R_t$

4. EXPERIMENTS

4.1 PyGame Learning Environment

In this research to evaluate results, an open-source game environments "PyGame Learning Environment" (PLE)[3] were used.

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.

Curriculum learning were implemented for the 3D raycast maze, where target moves away from starting point in later stages of training. Method to produce Q-value map (Q-map) were

[1] https://github.com/evaldsurtans/dqn-prioritized-experience-replay

[2] https://bitbucket.org/evaldsurtans/dqn-research

[3] https://github.com/ntasfi/PyGame-Learning-Environment

implemented 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, the bird character is moved across all pixels in a frame and a Q-function value is calculated that is overlaid as a heat map like in Figure 1. 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 Figure 2.

4.2 Random seed and repeatability

Our research highlights a problem that all DQN, DDQN and MDQN variants are very sensitive to seed randomization. In this research method to restore all random seeds and repeat results were implemented, but this is not desirable because it can lead to misleading results when comparing different hyper-parameters. A better approach is to increase the sample size of random seeds. This means that every training configuration should be rerun multiple times with different randomization seeds as shown in Figure 3.

Large variance between different samples of random seed were observed. To make accurate comparisons, it is necessary to choose 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.

To complete this research, 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 Figure 4, however by reducing model complexity also a maximal average score of testing set reduces as well. When constructing such models, it is necessary 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 Figure 5.

As for batch normalization, no significant improvement was found as shown in results in an appendix.

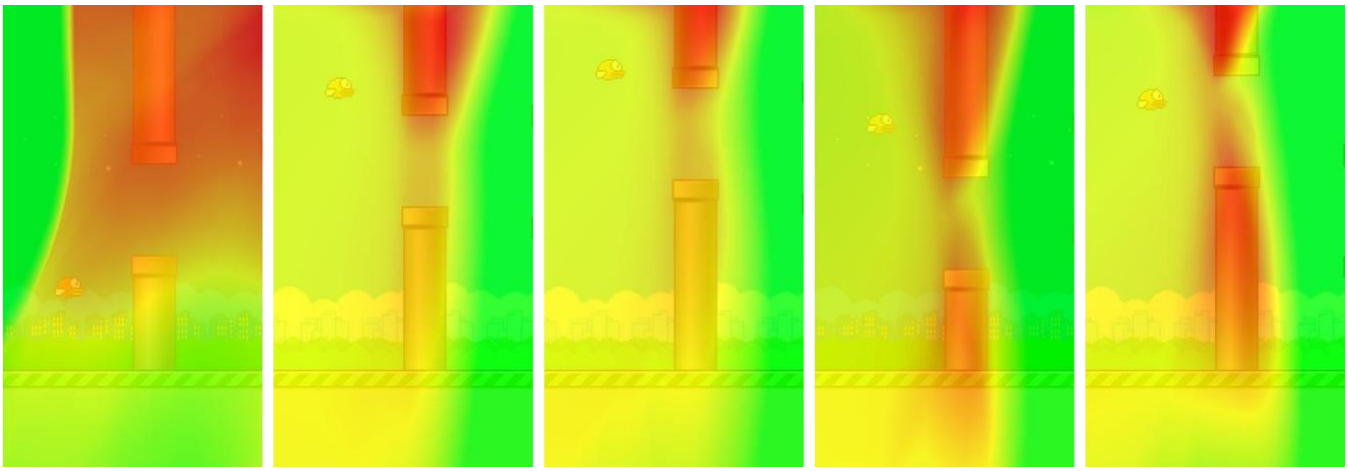

Figure 1. Q-Maps of sequential training of Flappy Bird from first frame on left till 10^7 frame on right.
Green is highest value state. Red is lowest value state.

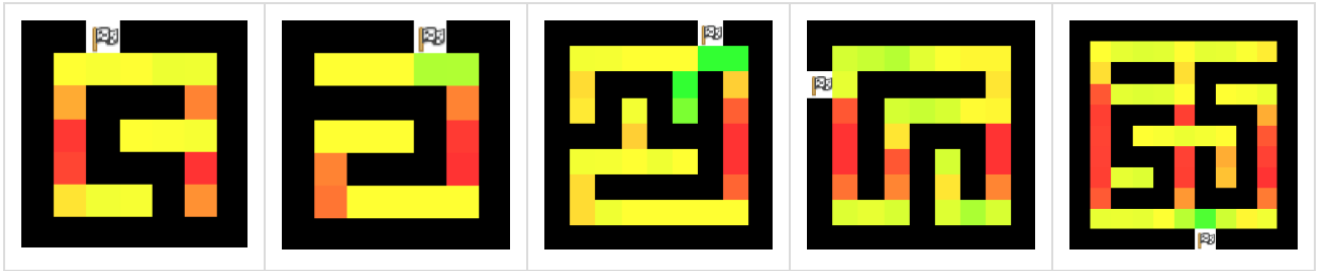

Figure 2. 3D Raycast maze Q-map for each position in map from top down view. Each Q-map represents sequential frame checkpoint during training. On left first frame and on right 10^7 frame.
Notice that map increases in size thus using curriculum learning principle.

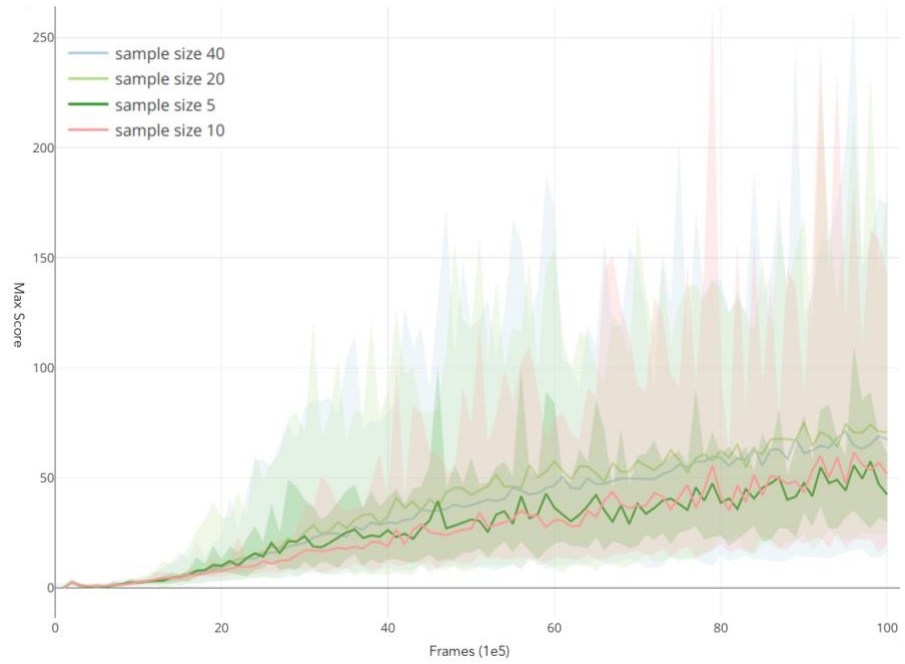

Figure 3. Sample size of random seeds and variance of average score for Flappy Bird environment.

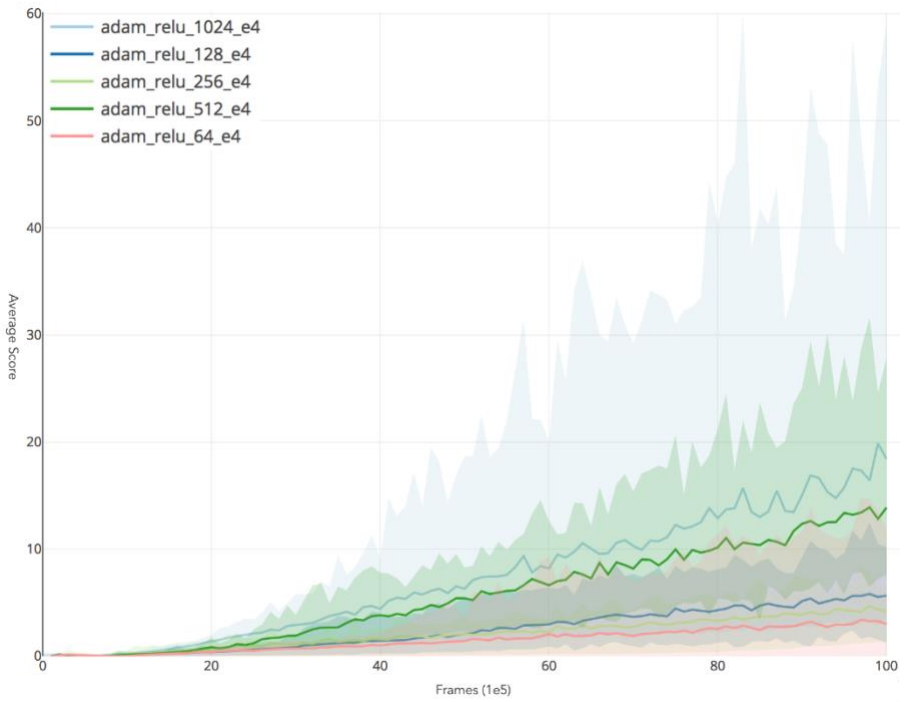

Figure 4. Comparison of different hidden unit vector sizes and variance of average score for Flappy Bird environment.

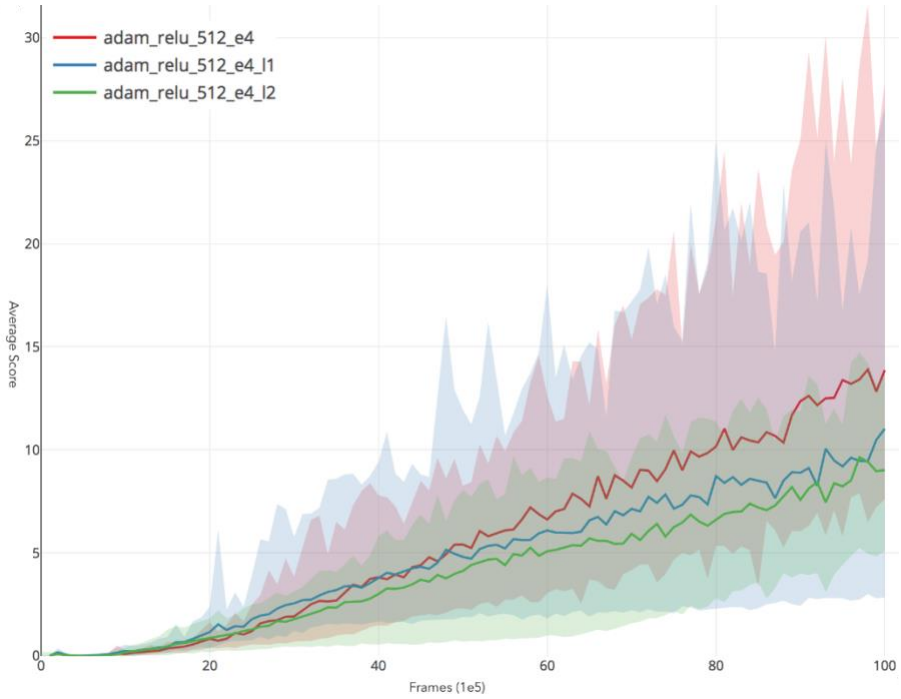

Figure 5. Effect of L1 (Lasso) and L2 (Ridge) regularization on variance of average score for Flappy Bird environment.

Table 1. Default 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

Table 2. Top 15 hyper-parameters of DDQN for Flappy Bird environment

parameter	lr	avg. score	max. score	var. score	time (min.)
rnn: gru	0.0001	42.97024986	264.1	549	2603.809
rnn: lstm	0.0001	28.60916534	264.1	99.2	3246.475
optimizer: adam	0.001	16.96737049	264.1	36.9	351.827
grad clip.: 1.0	0.001	12.45431387	254.096	19	399.956
optimizer: adam	0.0001	10.19414131	207.07831	21.7	367.396
grad clip.: 10.0	0.0001	9.765907544	184.06991	19.2	361.19
grad clip.: 10.0	0.001	8.694875819	156.05916	12.8	364.509
grad clip.: 1.0	0.0001	7.633139692	140.05331	17.1	357.085
rnn: lstm	0.001	6.614351436	182.06913	21.5	3770.588
optimizer: adam	0.00001	1.929185929	47.0181	1.81	375.648
mini-batch: 8	0.00001	1.922057009	40.0155	0.414	505.778
bellman gamma: 0.90	0.00001	1.916720841	47.01799	1.19	366.701
grad clip.: 10.0	0.00001	1.783358575	32.01237	0.322	400.212
beta replay buffer: false	0.00001	1.647591408	60.02323	1.02	423.562
default	0.00001	1.533690858	32.01236	0.308	377.689

Table 3. Top 15 hyper-parameters of MDQN3 for Flappy Bird environment

parameter	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	0.00001	1.047481234	19.00759	0.298771101	745.303
model: n states to n act.	0.00001	0.881531711	12.00483	0.101958401	600.168
epsilon stuck: true	0.00001	0.878962391	11.00452	0.013849234	384.174
grad clip.: 10.0	0.00001	0.86220963	21.00828	0.183416314	402.766
mdqn: min	0.00001	0.740644674	10.0041	0.05608085	376.083

Table 4. 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	0.001	16.96737049	264.1	36.9	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	0.00001	1.603283236	61.02341	0.715069292	642.106
mdqn3 0.0	0.00001	0.872432773	12.00487	0.087176378	713.332
mdqn2 1.0	0.00001	0.692394861	12.00513	0.221167891	389.493

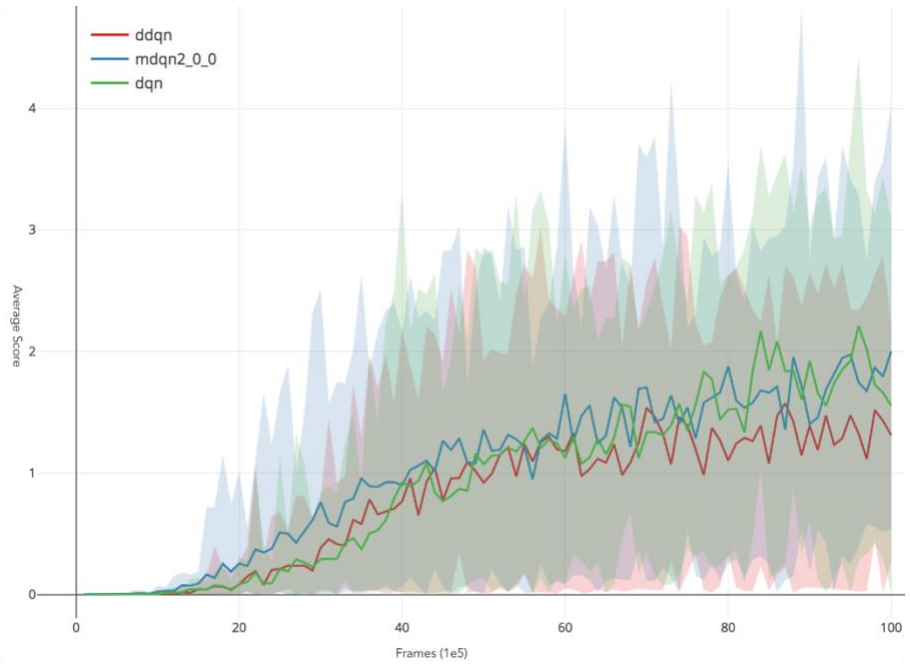

Figure 6. Comparison between DQN model types for Pong environment.

Table 5. Comparison of DQN, DDQN and MDQN models for 3D Raycast maze environment.

model type	lr	avg. score	var. score	time (min.)
mdqn2 0.0	0.00001	3.904359232	0.728045918	1213.991
dqn	0.00001	3.88654262	2.124993494	484.859
mdqn2 1.0	0.000001	3.7166532	0.154117942	533.389
ddqn	0.000001	3.713829593	1.524318234	524.65
ddqn	0.00001	3.638360789	1.662039807	521.975
mdqn2 0.0	0.00001	3.506246831	1.746571203	809.374
mdqn2 0.0	0.000001	3.345749731	2.749636472	978.638
ddqn	0.0001	3.267777864	2.889255991	523.012
mdqn2 1.0	0.0000001	3.247272282	0.576931468	500.424
mdqn2 1.0	0.00001	3.180342964	2.016163812	523.085
mdqn3 1.0	0.00001	3.056116361	2.339890159	872.317
dqn	0.000001	3.026868771	2.028895348	534.022
mdqn3 0.0	0.00001	2.807473511	2.395394139	1212.21
mdqn3 1.0	0.000001	2.770128326	0.714132328	864.442
mdqn3 0.0	0.000001	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	0.000001	1.452786487	1.887540531	809.63
mdqn2 1.0	0.00000001	1.399096696	1.827157866	495.813
mdqn2 0.0	0.0000001	0.178030303	0	463.67

4.3 Flappy Bird

Initially to test more than 28 hyper-parameters of DQN variants partial grid searches were done on combinations of parameters. Then benchmarking for one step changes were dome 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 4.2 section. By run, we mean full training of 10^7 frames with a defined set of hyper-parameters. RNN-ReLU were used as Q-value model in order to speed up training and compared DQN, DDQN and MDQN algorithms with full set of hyper-parameters as shown in Table 2, Table 3 and Table 4.

Original DQN outperformed DDQN and MDQN, but our version of MDQN slightly outperformed 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.

No significant improvements were found by applying some of more interesting architectures like Dueling Network or different activation functions in RNN like Leaky ReLU, ELU, and PreLU.

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 it was possible 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 Figure 1.

4.4 Pong

For Pong and 3D raycast maze environments, in initial hyper-parameters optimizer was changed 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 Figure 6.

Q-Value maps were generated by manipulating the position of the ball in Pong environment on the frozen Q-Value model at checkpoints during training as shown in Figure 7.

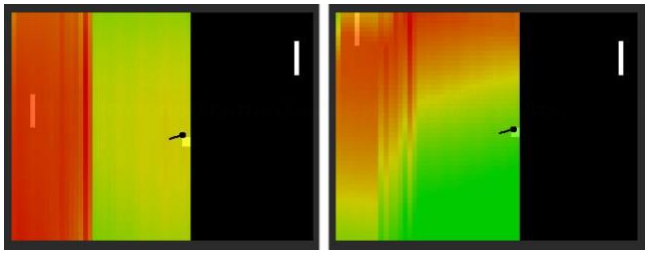

Figure 7. Pong Q-map before and after training. After training possible to see path of a ball trajectory.

4.5 3D Raycast maze

Finally, algorithms were benchmarked on "3D Raycast Maze" environment where instead of a low dimensional representation of a state, RGB 48x48 pixel input was used. In many environments, to save resources pixel grayscale representation would be recommended, but to make sure that exit door have a distinguishable difference in color form walls two channels were used 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 GradCAM maps [20] were generated to visualize highest gradients in ConvNet as shown in Figure 8. These maps are more informative than Saliency Maps used in other Deep Reinforcement Learning papers [24]. 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 the dimensionality of the problem was to remove some of the actions available to an agent. Agent was allowed only to move ahead and make turns left and right, but not to go back and wait (do nothing).

Again Q-Value maps were constructed to visualize the progress of learning as shown in Figure 2. In this case, we manipulated a position of player around the maze and recorded Q-Values by rotating player's view around this position. Visual representation is a 2D top view for 3D maze.

Again, slight performance improvement were found using MDQN in a more complex environments like 3D Raycast maze as shown in Table 5.

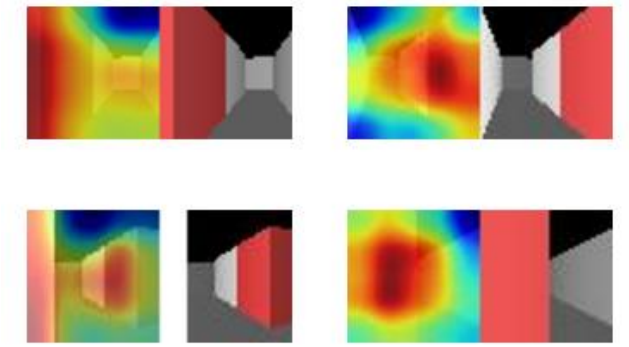

Figure 8. GradCAM maps of trained MDQN agent in 3D Raycast maze environment.
Images show attention on target in a 3D maze.

5. CONCLUSIONS

MDQN, a new Deep Reinforcement Learning algorithm was introduced that slightly outperforms DDQN in some environments. Still, in others, original DQN work better than both MDQN and DDQN.

Most of DQN variants that were tested have little or no significant effect on performance. New method to construct Q-Value maps were introduced by manipulating training environment. Q-Value maps are useful for assessing the progress of training. Results show 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.

6. APPENDIX

With this paper, spreadsheet is published of an average score in a game of Flappy Bird after 10^7 frames for each hyper-parameter with different learning rates. All hyper-parameters have been tested for DQN, DDQN and MDQN variants of algorithms. Results are available in public domain[4].

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] Anschel, O., Baram, N. and Shimkin, N. 2016. Deep Reinforcement Learning with Averaged Target DQN. *NIPS Workshop*. (2016).

[2] Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J. and Kautz, J. 2017. GA3C: GPU-based A3C for Deep Reinforcement Learning. *ICLR*. (2017).

[3] Duan, Y., Chen, X., Houthooft, R., Schulman, J. and Abbeel, P. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. *ICML*. 48, (2016), 1329–1338.

[4] Geron, A. 2017. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media.

[5] Hasselt, H. van, Guez, A. and Silver, D. 2015. Deep Reinforcement Learning with Double Q-learning. *CoRR*. abs/1509.06461, (2015).

[6] Hasselt, H. van, Guez, A. and Silver, D. 2016. Deep Reinforcement Learning with Double Q-learning. *Proceedings of AAAI*. 13, (2016), 2094–2100.

[7] Hasselt, H.V. 2010. Double Q-learning. *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.

[8] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D. and Meger, D. 2018. Deep Reinforcement Learning that Matters. (AAAI, 2018).

[9] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M.G. and Silver, D. 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning. *CoRR*. abs/1710.02298, (2017).

[10] Islam, R., Henderson, P., Gomrokchi, M. and Precup, D. 2017. Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. *ICML*. (2017).

[11] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage and Anil Anthony Bharath 2017. A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine*. (2017).

[12] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *US Patent 20170024643 A1*. (2015).

[13] Mnih, V. et al. 2015. Human-level control through deep reinforcement learning. *Nature*. 518, 7540 (2015), 529–533.

[14] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D. and Kavukcuoglu, K. 2016. Asynchronous Methods for Deep Reinforcement Learning. *ICML*. 48, (2016), 1928–1937.

[15] Pfau, D. and Vinyals, O. 2016. Connecting Generative Adversarial Networks and Actor-Critic Methods. *NIPS Workshop on Adversarial Training*. (2016).

[16] Schaul, T., Quan, J., Antonoglou, I. and Silver, D. 2016. Prioritized Experience Replay. *ICLR*. (2016).

[17] Schulman, J., Levine, S., Moritz, P., Jordan, M.I. and Abbeel, P. 2015. Trust Region Policy Optimization. *ICML*. (2015), 1889–1897.

[18] Schulman, J., Moritz, P., Levine, S., Jordan, M.I. and Abbeel, P. 2016. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *ICLR*. (2016).

[19] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *CoRR*. abs/1707.06347, (2017).

[20] Selvaraju, R.R., Das, A., Vedantam, R., Cogswell, M., Parikh, D. and Batra, D. 2017. Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization. *ICCV*. (2017).

[21] Tasfi, N. 2016. PyGame Learning Environment. *GitHub repository*. (2016).

[22] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K. and de Freitas, N. 2017. Sample Efficient Actor-Critic with Experience Replay. *ICLR*. (2017).

[23] Wang, Z., Freitas, N. de and Lanctot, M. 2015. Dueling Network Architectures for Deep Reinforcement Learning. *CoRR*. abs/1511.06581, (2015).

[24] Wang, Z., Freitas, N. de and Lanctot, M. 2016. Dueling Network Architectures for Deep Reinforcement Learning. *ICML*. 16, (2016), 1995–2003.

[4] http://yellowrobot.xyz/full-survey-flappybird.pdf