Aastav Sasha Sen

"DQN and DDQN on Stable Learning in the Atari Breakout Domain"

… Empirical Evaluation

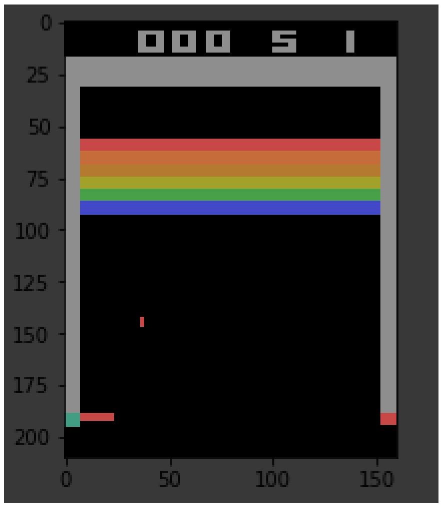
Reinforcement Learning (RL) studies how an agent can achieve goals in a complex and uncertain environment through exploring it in an unsupervised manner. This is exciting because RL is very general and encompasses all problems that involve making a sequence of decisions. Popular test beds for RL research are games. These are rich and easily available simulated environments that often have infinite possibilities. A modern standard environment for RL research and testing/evaluating RL algorithms are Atari environments [1].

Reinforcement learning presents several challenges from a deep learning perspective. RL algorithms must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed in a regime of highly correlated data sequences. Another problem faced by RL algorithms is generalization. Network architectures and parameters are usually tailor made to suit the problem, however, what we desire is a single general network that can be applied to a variety of problems. Thus, we will explore algorithms through implementation that were designed to be applied to a variety of Atari games (however we will only be training/testing on Breakout).

The goal of this project is to compare approaches to RL game solving and investigate efficient algorithms to solve the Atari 2600 game "Breakout". The papers of focus are [1, 2, 4]. The approach will be as follows (details follow):

- ***ALGORITHM 1:*** Deep Q-Learning + Experience Replay.
- ***ALGORITHM 2:*** Double Deep Q-Learning with Fixed Q-Targets + Experience Replay.

The Atari 2600 "Breakout" Environment



The environment of choice (as previously mentioned) will be the Atari game known as Breakout. In this project we will only investigate the game in the discrete rather than continuous action space.

For both algorithms, the environment used is "BreakoutDeterministic-v4" [5]. This provides an input of unprocessed pixels such that we can implement convolutional networks for feature extraction. The 'Deterministic' part ensures a fixed frame skip of 4 frames where the agents last action is repeated on skipped frames, as done in [1, 2]. This condenses our input data and is chosen instead of 'NoFrameskip' in order to reduce training time. From [2] "the agent sees and selects actions on every kth frame instead of every frame, and its last action is repeated on skipped frames" - which is directly implemented in the environment.

Background/ Motivation

The main sources of relevant work are as follows: [1] Presented Deep Q-Learning (DQN), a variant of online Q-learning that combines stochastic offline mini batch updates with experience replay memory to ease the training of deep networks for RL. This resulted in state-of-the-art scores in six of the seven games it was tested on, with no adjustment of the architecture or hyper parameters, suggesting a general solution. Our first implementation will follow this approach. [2] Also introduces a general algorithm across many Atari games that employs experience replay and "an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target". The suggested method outperformed the best existing reinforcement learning methods on 43/49 of the games without incorporating any additional prior knowledge" [2]. Finally, [4] suggests reducing overestimations (common and severe in some Atari games such as Road Runner) by decomposing the max operation in the target into action selection and action evaluation, thus introducing the concept of Double Q-Learning. At publication, it was shown that "Double DQN finds better policies, obtaining new state-of-the-art results on the Atari 2600 domain" [4].

"Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function" [2]. Convergence of the Q-function is not guaranteed with the DQN [1] algorithm. The Q-function approximator learns to ascribe unrealistically high values to state-action pairs, destroying the quality of the greedy control policy derived from Q. This along with large oscillation in training due to moving Q-targets, increases the possibility of divergence. Greater stability in the DQN algorithm is desirable. Thus, our goal is to investigate the improvements of the fixed target DDQN (improvement by [2, 4]) over the conventional DQN (suggested by [1]) and mainly evaluate the stability of learning (by analyzing the behavior of loss and accumulated episodic rewards). We will proceed by providing an overview of the implemented concepts for the algorithms before evaluating the empirical results of our 2 implementations.

Algorithm 1: Deep Q-Learning + Experience Replay [1]

Time Series Representation in Input

This was suggested and implemented in the original DQN paper [1], as well as [2, 4]. This will allow the algorithm to understand the direction of travel of the ball, an important 3 dimensional interpretation that is crucial in playing the game. Rather than a single frame as input, a stacked input of the 4 previous frames is used. This makes the input (4, 105, 80) (after preprocessing, see "Q-function approximator").

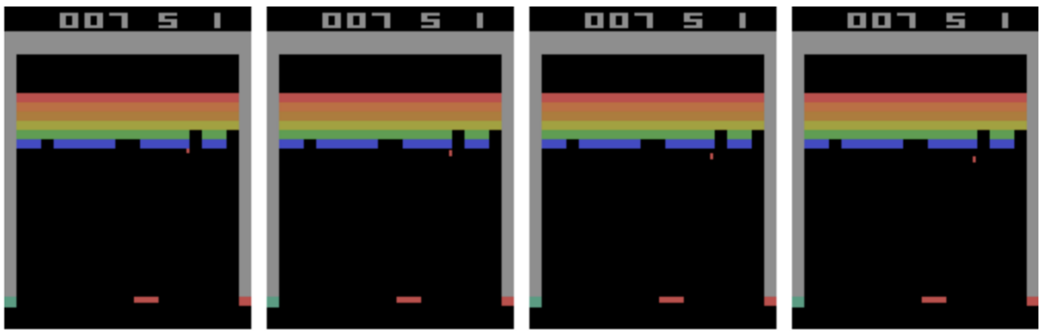


Figure 2: Stack of 4 frames (unprocessed)

Q-Function Approximator

The motivation to incorporate a convolutional neural network is to make decisions on the same visual input just as we do when we play the game, learning successful policies from video data. The input image is of shape (210, 160, 3). For faster training this image is preprocessed by downsizing (by a factor of 2) and making the image grayscale, resulting in an input shape of (105, 80) into the convolution input. Deepmind [1] does a similar image preprocess resulting in shape (84, 84). The final fully connected layer consists of a separate output node for each action. An architecture summary is given below (same as in [1]):

```
Layer (type)              Output Shape          Param #
=================================================================
lambda_2 (Lambda)          (None, 4, 105, 80)     0          => Normalize pixel inputs
_________________________________________________________________
conv2d_3 (Conv2D)          (None, 16, 25, 19)     4112
_________________________________________________________________
conv2d_4 (Conv2D)          (None, 32, 11, 8)      8224
_________________________________________________________________
flatten_2 (Flatten)        (None, 2816)           0
_________________________________________________________________
dense_3 (Dense)            (None, 256)            721152
_________________________________________________________________
dense_4 (Dense)            (None, 4)              1028
=================================================================
Trainable params: 734,516
_________________________________________________________________
```

As in [1, 2, 4], the activation functions chosen for all layers (except last = linear) is RELu. The intuition is that we are using a Deep Q network to approximate our Q-function and to avoid the vanishing gradient problem network layers we implement the RELu activation function (gradients restricted to the values of either 1=activated or 0=deactivated). For faster convergence when using RELu activations, He-et-al initialization is used [8].

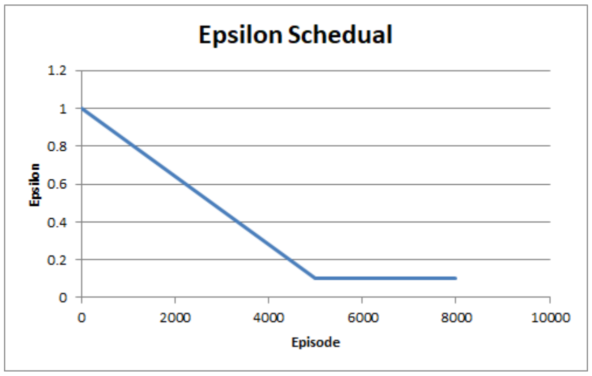
Optimizer Algorithm

As in [1, 2, 4], the RMSprop algorithm is applied. Similarly, the same optimizer parameters are used (lr=0.00025, rho=0.95, epsilon=0.01, decay=0.0) as [2]. Note also that rewards are clipped to be between [-1, 1], making the algorithm more robust in the presence of outliers.

Experience Replay

It is inefficient to learn directly from consecutive samples, as they are strongly correlated, and randomizing the samples reduces the variance of the updates. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, it is necessary to learn off-policy [2]. In [2, 4] replay memory size is specified to be 1,000,000 tuples, as we are processing less total frames our chosen replay size is 100,000 with replay being started after collecting 1,000 experience tuples. The replay gradient descent is conducted in batches of 32 and is done upon every observation/ action step [2], but, in [4] replay is done every 4 observation/ action steps. However, to decrease training time we do this every 40 observation/ action steps.

Exploration vs. Exploitation



We implement an e-greedy exploration/ exploitation (To ensure that the algorithms are comparable the same epsilon schedule is used). [1, 2, 4] use a linear decaying initial epsilon schedule that maintains its value after reaching a minimum. The reasoning behind this decreasing factor of exploration is that initially our agent knows nothing of the environment, and the most useful predictions are random actions. Note that for testing, a value of 0.05 is assigned to epsilon as in [1, 2, 4].
Such as epsilon and its schedule, to preserve comparability value between the algorithms the following parameters remain unchanged between the algorithms:

Table 1: DQN Network Hyper Parameters:	
Epsilon min. episode = 5,000 Epsilon start value= 1.0 Epsilon minimum value = 0.1 Epsilon decay = (see Epsilon Schedule) Memory max length = 10,000 Replay start size = 1,000 Batch size = 32 Replay frequency = 40 steps	Episodes = 8,000 Learning rate = 0.00025 RMSprop rho = 0.95 RMSprop epsilon (decay) = 0.01 Gamma (discount factor) = 0.99 [1,2,4] Frame stack number = 4 Frame skip = 4

Algorithm 2: Double Deep Q-Learning + Fixed Q-Targets + Experience Replay [2, 4]

Fixed Q-Targets [2]

An implementation following [1] is that of fixed Q-targets, suggested in [2]. Previously, we use the same weights/ parameters to make an estimation of the target as well as the Q-value. This means there is a big correlation between the target and the weights/ parameters we are adjusting, thus, upon updating the Q-value shifts but our target value also shifts! This leads to large oscillations in training due to oscillations in the Temporal Difference Error:

$$TDerror = R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)$$

What is suggested is to use 2 networks, a target network with frozen parameters that estimates the target Q-value (for a fixed number of steps) and an active network that is updated every step and approximates the Q-value function. As summarized in [2]: *"…we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target."* Thus this reformed TD error becomes:

$$ReformedTDerror_i = R_{t+1} + \gamma max_a Q(S_{t+1}, a; \theta_i^-) - Q(S_t, A_t; \theta_i)$$

Note that θi are the online weights. The frequency of the update of the target network weights from the main q-network is 10,000 steps [2] (as specified in their code). As we perform far less iterations we will arbitrarily choose a parameter update period of 1,000 steps. On the contrary to [1] (replay every observation/ action step), [4] suggests updating the parameters of the target network every 4 replay (of batch size 32). What we choose to do (to decrease training time) is replay every 40.

Double Q-Learning [4]

According to the optimal policy in basic Q-Learning, the agent tends to take the non-optimal action in any given state only because it has the maximum Q-value. The problem of overestimations is that the agent always choose the non-optimal action in any given state only because it has the maximum Q-value. The solution suggested by [4] is to decouple the action choice and the Q-value evaluation using 2 separate networks. Thus, our new estimation becomes:

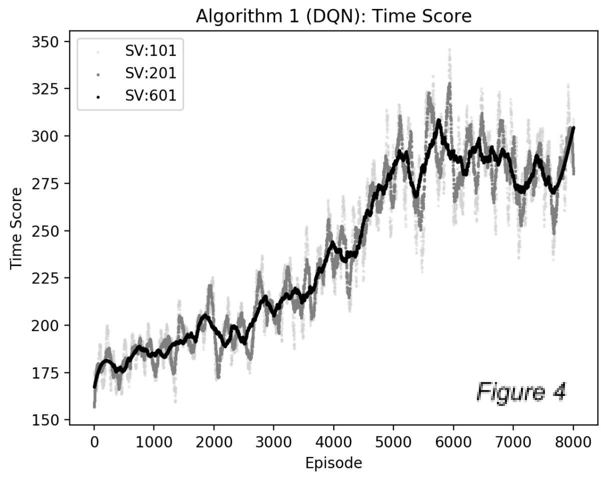
$$Q(s, a)_{Targ.} = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_i); \theta_i^-)$$

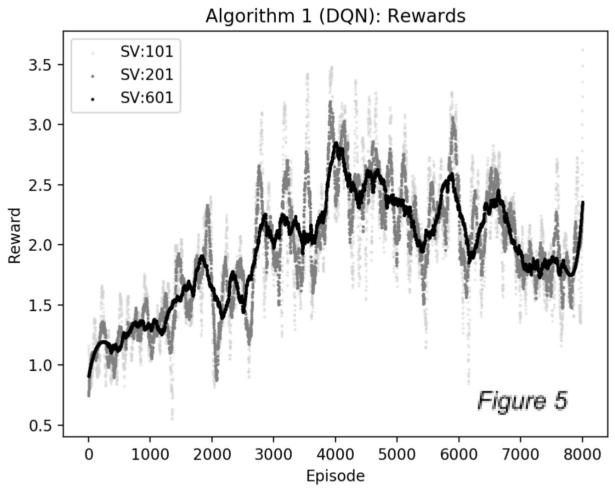
The noises from estimated Q-value will cause large positive biases in the updating procedure, uncoupling estimation into 2 different q-value estimators reduces this unwanted residual trait. Apart from the previous mentioned parameters (remain the same here), the following parameters are added to Algorithm 2:

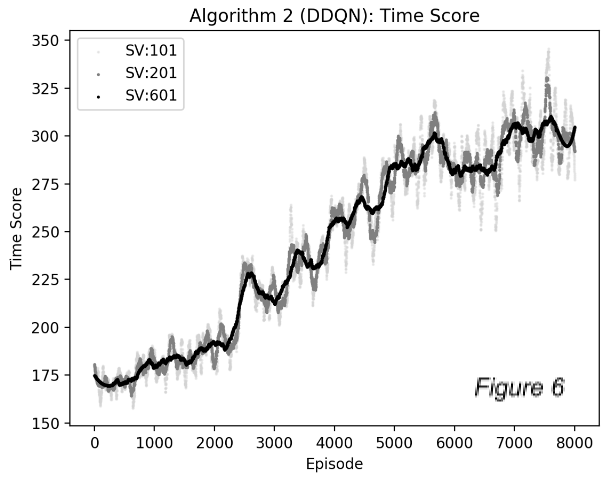
Table 2: DDQN Additional Network Hyper Parameters:	
Target network hyper parameters = Main network hyper parameters	Target network weights update frequency = 1,000 observation/ action steps

To represent the data, it is plotted with various levels of smoothing (done by the Savgol Filter of order = 3) in order to illustrate the trend and variance for the data (note that the SV windows are specified in the legend). To evaluate the agents score both the agents survival time (observation/ action steps) and the agents accumulated rewards per episode are plotted. The agent's survival time is proportional to its score and is a useful qualitative measure of score alongside the accumulated rewards per episode. The relevant plots are shown below:

Results







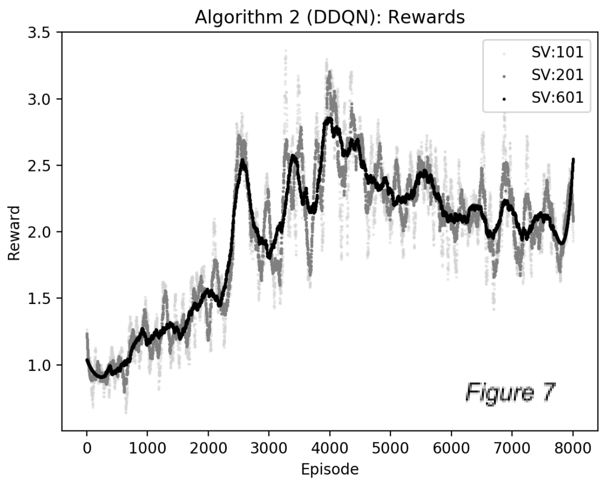
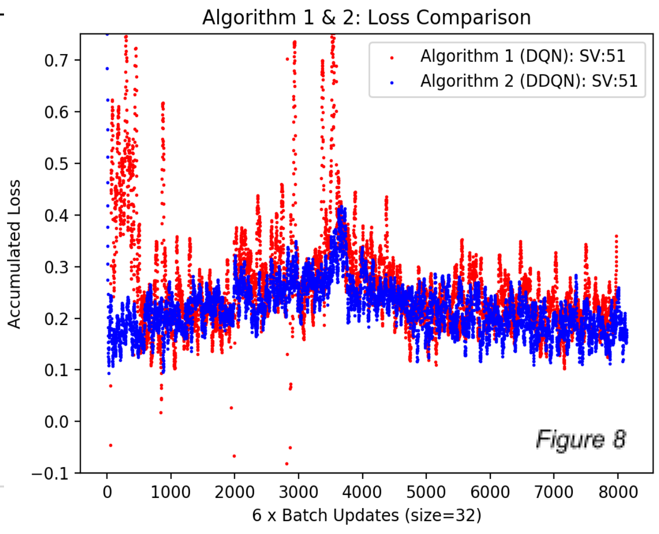


Table 3: REWARD PLOTS	**Mean (5dp)**	**Standard Deviation (5dp)**
Algorithm 1 (DQN)	1.94288	2.13802
Algorithm 2 (DDQN)	1.94913	1.57719

Table 4: LOSS PLOTS	**Mean (5dp)**	**Standard Deviation (5dp)**
Algorithm 1 (DQN)	0.00650	0.43617
Algorithm 2 (DDQN)	0.00114	0.01094



Discussion

Learning capacity/ speed in initial regime: We consider our training regime to be the 'initial' regime (prior to convergence) due to limited training iterations and updates. As can be seen from the figures above, time score and accumulated rewards per episode both improve suggesting learning for both algorithms. However, there is no indication that DDQN has a greater learning capacity in the initial regime when compared to the DQN.

Behavior of loss: The accumulated loss plot of algorithm 1 shows a greater variance and more oscillation than that of algorithm 2. Unfortunately a loss metric in RL cannot capture how good that policy is, rather a lower loss means more accurate predictions of value for the current policy. This is due to overestimations of the Q-values (large Q-values) as well as the large correlations between the target Q-value and the generated Q-value in algorithm 1 ('moving target'), resulting in large oscillations of the loss (which consists of their difference). As the loss is used for the gradient update, algorithm 2 shows greater stability in learning and a lower possibility of divergence in complex environments.

Behavior of rewards: As the agent gets better at playing the reward gets higher and the average episode length gets longer, thus the amount of variance in the reward also gets larger. As can be seen from the table and visualized on the plots above, the plot for the accumulated rewards per episode for algorithm 1 oscillate with a greater variance than for algorithm 2. As in 'behavior of loss', this is an indication that algorithm 2 shows greater stability in learning as to target Q-value estimation converges to the optimal policy through adjustment of the Q-function by introducing rewards.

Thus, algorithm 2 shows more stable convergence properties and less likelihood to diverge as shown in greater stability in the loss function and the reward plot (suggested by their lower standard deviation), thus significantly benefiting the stability of learning. This is mainly due to the implications of fixed Q-targets, and minorly due to reduced overestimation by Double Q-Learning. This is partly because Q-Learning's overestimation increases with the action space while Double Q-Learning remains unbiased. For our action space of 4 (in Breakout), overestimation is present yet minimal ([4]: Figure 1).

However, algorithm 2 has a higher time complexity, thus more training time is required, ie) a total time of **670 min** while algorithm 1 took **533 min**. This increased time complexity is due to the need to copy weights from the main Q-function network to the target Q-function network. Also, Double Q-Learning in many simpler environments can prove to be computationally wasteful. [4] "Over optimism does not always adversely affect the quality of the learned policy. For example, DQN achieves optimal behavior in Pong despite slightly overestimating the policy value". This is also the case in our experiment (concerning the Breakout environment).

Final Discussion:

Table 5: Mean Test Scores	DQN	DDQN
Reported Mean Test Score	0.522	0.686

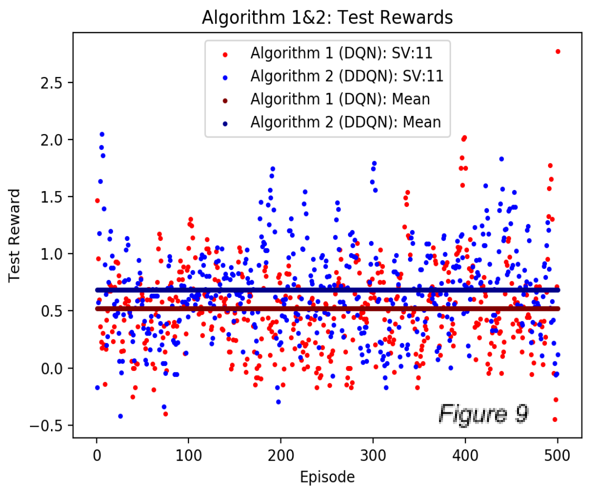


Figure 9

The best implemented solution to the Breakout Environment is the DDQN with fixed Q-targets. This is suggested by our results that suggest that the DDQN architecture results in more stable learning and a greater average test score. However, as discussed, there is a tradeoff of time complexity that may not be worth it depending on the Environment/ Problem as hand. For example, as suggested by [4]: (Figure 4) low action space games such as Breakout and Pong gain little (Breakout) or no (Pong) advantage from the DDQN architecture as opposed to using the original DQN architecture. We observe this in our own results, as can be seen from the mean accumulated testing rewards that differ only slightly, as well as the similar trend in the accumulated training rewards per episode between the two algorithms.

Improvements & Further Studies

The reward plots appear to illustrate the asymptotic convergence behavior of Q-learning, yet, we have fallen short of even the human average (~31 [1]). There are many reasons for this, however, the main reason is that we conduct a limited number of training iterations (greatly limiting the number of frames observed) and less frequent gradient descent updates in order to reduce training time. Another reason for this is that we 'limit' the effects of exploration when compared to [1, 2, 4].

Our chosen epsilon schedule greatly limits exploration. This is evident on both Figure 5 and Figure 7 as learning appears to deteriorate past 4,000 episodes in both algorithms (at this point exploration is limited; epsilon = 0.28). Although, as in [1], we linearly anneal epsilon from 1 to 0.1 over the first ~1,000,000 frames (in our case ~200 frames/ episode) then keep epsilon constant, we conduct far less gradient descent steps during this schedule as we conduct replay less frequently (every 40 steps as opposed to [1] = every step and [2] = every 4 steps). Thus, limited training on the collected distribution of experience tuples in memory results in our Q-function not considering better states and later (due to low epsilon value) acting on limited known/ seen states resulting in a non-optimal policy. Generally, increasing the exploration rate may help find the better states, at the expense of slower overall learning. Compared to sources: In [1] "One epoch corresponds to 50000 mini batch weight updates or roughly 30 minutes of training time." (for 100 epochs) This is equivalent to **160,000,000** single gradient descent steps. [2] was trained on 50,000,000 frames with replay (batch size of 32) occurring every 4 observation/ action steps. This is equivalent to **400,000,000** single gradient descent steps. In [4] a network with 1.5M parameters (more than 2x ours) is trained for **200,000,000** SGD steps. On the contrary, we do **<1,600,000** single gradient descent steps. This significantly reduces our learning capabilities (a useful measure of this could be the gradient of cumulative reward w.r.t frames observed).

For further investigation into the advantages of Double Q-learning, the change in the Q values during gameplay should be analyzed (as done in [4]) and compared to that of the conventional DQN. This will allow us to visualize how overestimations of Q-values can result in a diverging policy by showing that DDQN action value learning curves are closer to the final policy than for a DQN. This should be done on games other than breakout, particularly high action space games such as "Double Dunk". In such environments the DQN algorithm fails to even come close to human score averages due to Q-value overestimations while the DDQN algorithm exceeds this baseline (as can be seen in [4]: Figure 4).

None of the techniques tackled in this paper can be considered good enough to consider the problem solved. It is in fact an implementation of ACER - Proximal Policy Optimization that holds the current high score of Breakout according to Leaderboards (see [3]). It should also be noted that our investigated algorithms still fail to learn human level policies over a set of diverse tasks even in the regime of Atari 2600 games. Games such as Montezuma's Revenge and Private Eye require reasoning over extended time horizons, diverse reward distributions and require efficient exploration. This presents a topic of future research, as both DQN's and DDQN's are unable to learn in such environments ([4]: Figure 4). Recent variants such as [7] have been able to "solve the first level of Montezuma's Revenge" (a first in Deep RL), and the presented concepts should also be evaluated for their stability of learning and generality (applicability across many environments/ problems).

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. *Nature*, *518*(7540), p.529.

[3] Lee, S. (2019). *Atari Breakout Environment | endtoendAI*. [online] Endtoend.ai. Available at: https://www.endtoend.ai/envs/gym/atari/breakout/ [Accessed 2 Aug. 2019].

[4] Van Hasselt, H., Guez, A. and Silver, D., 2016, March. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*.

[5] Bellemare, M. (2019). *mgbellemare/Arcade-Learning-Environment*. [online] GitHub. Available at: https://github.com/mgbellemare/Arcade-Learning-Environment [Accessed 3 Aug. 2019].

[6] Simonini, T. (2018). Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed…. [Blog] *freeCodeCamp*. Available at: https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/ [Accessed 1 Aug. 2019].

[7] Pohlen, T., Piot, B., Hester, T., Azar, M.G., Horgan, D., Budden, D., Barth-Maron, G., Van Hasselt, H., Quan, J., Večerík, M. and Hessel, M., 2018. Observe and look further: Achieving consistent performance on atari. *arXiv preprint arXiv:1805.11593*.

[8] He, K., Zhang, X., Ren, S. and Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026-1034).