Proper evaluation of Vanilla and Bootstrapped Deep Q-Networks

Apoorv Kulshreshtha
ak3963@columbia.edu

Dheeraj Kalmekolan
drk2143@columbia.edu

Sneha Nagaraj Bangalore
sb3889@columbia.edu

Abstract

Approximation of Q value function using neural networks led to the development of Deep Q Network algorithm [8]. Soon after, the bootstrapped version of the algorithm was shown to have improved results [9]. Both these papers used Arcade Learning Environment as an evaluation platform to gauge the effectiveness of the trained agent. In September 2017, the protocols for evaluating agents on Arcade Learning Environment (ALE) were published, to standardize comparison across future research in this field [5]. Our work aims at using these evaluation protocols and doing a uniform comparison of DQN and Bootstrapped-DQN algorithms. We showcase our evaluation results on 2 Atari games: Breakout and Pong.

1 Introduction

Evaluation of deep RL agents has been challenging for various reasons: computationally expensive, large training times, different ways of testing and summarizing agent performance across published research results and so on. Recently, protocols for evaluation of agents on ALE were published, benchmarking the methodology which should be employed for the same [5]. Another very recent paper called "Deep RL that matters" laid down the guidelines for making future results in Deep RL more reproducible [3]. Given that both these works were published after the DQN and Bootstrapped-DQN algorithms were published (two fairly popular algorithms in RL), our interest naturally inclined towards evaluating and comparing both the algorithms with the prescribed methodology. In this paper, we describe the techniques recommended for evaluation, followed by a brief description of DQN and Bootstrapped-DQN algorithms. Finally, we mention the hyperparameters we used for our simulations and the results we received on Breakout and Pong.

2 Evalutation Protocols for Arcade Learning Environment

Our evalutation on Breakout and Pong was done using OpenAI gym [1] package available in Python. It uses the Arcade Learning Environment(ALE) as the backend. Agents interact with ALE in an episodic way. An episode starts by resetting ALE to its initial configuration, and ends at the usual endpoint of the game. Most games end when the agent looses all its lives. ALE has been a popular platform for evaluating RL agents, since its inception. However, researchers have been testing on ALE by using many distinct experimental protocols. Due to this, the direct comparison of results has been difficult. Moreover, the high computational cost of evaluting deep RL algorithms has made it more difficult to re-evalutate results mentioned in such papers. Machado et. al [5] lists the following standard methodology for evaluating agents on ALE:

Episode Termination:
To ensure minimization of game-specific information and the uncertain utility of termination using the "lives" signal, it is recommended that only game-over signal be used for termination.

Setting of hyperparameters:
The benchmark recommends a train/test game split as a way to evaluate agents in problems they were not specifically tuned for. By using such a split to determine hyperparameters, the agents will learn to generally perform well. As our task was more akin to evaluating performance of 2 different algorithms on the same Atari games, we didn't utilize this technique. Also, the benchmarked hyperparameters were already available to us from the paper [5], so we didn't have to tune our model on a train/test game split of all Atari games, which would have taken a considerable amount of time.

Summarizing Learning Performance:
Previous research in this field employs various statistics for summarizing agent performance and this diversity has made it difficult to directly compare mentioned results. The protocol recommends reporting training performance at different

intervals during training. According to it, at regular intervals during training, we should report the average performance of the last k episodes. Thus, the protocol doesn't suggest using an explicit evaluation phase, requiring an agent to perform well while it is learning. Also, by reporting performance at multiple times during training, researchers can easily draw comparisons earlier in the learning process, reducing the computational burden of evalutating agents. In our simulations, we mention the performance during training phase as well, in addition to the evaluation phase performance numbers.

Injecting Stochasticity:
ALE is deterministic i.e. each game starts in the same state and outcomes are fully determined by the state and the action. Therefore, it is possible for an agent to achieve high scores by memorizing actions sequences which give good reward, rather than actually learning to play well. To overcome this issue, the protocol suggests using repeat-actions (or sticky-actions) to inject stochasticity during evaluation. Repeat action uses a probability parameter ς, which is the probability at every timestep that the environment will execute the agent's previous action again, instead of agent's new action. For our experiments, we use $\varsigma = 0.25$.

In addition to the above ALE protocols paper, "Deep RL that Matters" paper [3] provides many important considerations that one must adhere to, to properly evaluate deep RL algorithms. Since DQN and Bootstrapped-DQN fall into the category of Deep RL, we made sure our implementation for reproducing their results followed the recommendations of this paper as well.

Activation functions and network architectures:
Activation functions and network architectures must be picked after examining the performance of the algorithms like DQN or Bootstrapped-DQN across environments. For our reproduction of results, the games or environments considered were Breakout and Pong. We chose to use the [64, 64] architecture since that was the most promising architecture according to the Deep RL paper, and is also used in the original papers of DQN and Bootstrapped DQN. Similarly, we used ReLU for non linearity in the network based on recommendation [3] and original paper usage. [8] [9]

Evaluation strategy:
The paper recommends that for the purpose of evaluation, we perform 10 runs of episodes (for the same hyper parameter configuration) varying only the random seed each time. Then split the results into two sets of 5 and average these two groupings together. The results of this kind of an evaluation is shown in Table 1. Averaging multiple runs over different random seeds gives insight into the population distribution of the algorithm performance on an environment.

Environment selection:
The paper claims that it is not possible for an algorithm to perform well across all games and hence recommends that one show the results not only of the best performing game for a given algorithm but also other games too. We chose Breakout and Pong given that we had limited resources and time, but we made sure we did not pick these two games based on them being the best performance environments.

Video the game during evaluation:
The most important takeaway from this paper for us was to realize how necessary it is to demonstrate the learned policy in action. If a local optimum is reached, the learning curves can indicate successful optimization of the policy whereas in reality, the scores achieved are not qualitatively representative of learning the desired behavior. Hence, we decided to create video replays of the learned policy: Vanilla vs Bootstrap DQN (Breakout) and Vanilla vs Bootstrap DQN (Pong).

Offline v/s Online evaluation:
he paper [3] recommends either offline or online evaluation. In online manner, both exploration and exploitation continues to happen during evaluation. We have chosen to evaluate the value network in an offline manner (no exploration happens in evaluation phase).

Importance of Documentation:
Towards the end, the paper says that no matter what we do to evaluate our work, we need to make sure that we document the common evaluation method chosen for both the baseline and novel models and report all hyperparameters (Section 5).

After having gone through the process of hyper parameter tuning, the authors, as well as us, feel that, we need hyper parameter agnostic algorithms that incorporate hyper parameter adaptation as part of the design.

In the following sections, we give an overview of DQN and Bootstrapped-DQN algorithms, followed by our experimental results.

3 Q-Learning and Deep Q-Networks

Unlike policy gradient methods, which try to learn mapping from observation to actions, Q-Learning tries to learn the value of being in a given state, and taking a particular action in that state [4]. Q-Learning algorithm is based on a table where each row is a state and each column is possible actions in the environment. The value in each cell signifies how good it is to take a particular action in a particular state. Q-Learning uses Bellman equation to make updates to each cell in the table. According to Bellman equation, the expected long-term reward for a given action is equal to the immediate reward from the current action plus the expected reward from the best future action taken at the resulting state.

As Q-Learning uses tables with a cell for each configuration, the algorithm is not scalable for practical purposes. This is where Deep Learning comes into play. A neural network can replace the table and the network weights (called Θ) serve as the approximation for cells in the table. This neural network function approximator with weights Θ is referred as Q-Network [7] [8]. The Q-network is trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

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

where $y_i = \mathbb{E}_{s'\sim\epsilon}[r + \gamma max_{a'} Q(s', a'; \theta_{i-1})|s, a]$ (ϵ is the emulator environment) is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a. The parameters from the previous iteration θ_{i-1} are held fixed when optimizing the loss function $L_i(\theta_i)$. Differentiating the loss function with respect to the weights, we get the following gradient equation:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(.);s'\sim\epsilon}[(r + \gamma max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)]$$

For computational efficiency, instead of computing the full expectations in the above gradient, the loss function is optimized by stochastic gradient descent. The paper [8] also mentions using a technique called *Experience Replay* to improve the performance of the algorithm. The algorithm stores the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ into a data-set $\mathcal{D} = e_1, ..., e_N$, pooled over many episodes into a replay memory. During the algorithm, mini-batch updates are applied to samples of experience, $e \sim \mathcal{D}$, drawn at random from the pool of stored samples. After performing experience replay, the agent chooses and executes an action according to an ϵ-greedy policy.

Algorithm 1 Deep Q-Learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1,$M$ do
    Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for t = 1,$T$ do
        With probability $\epsilon$ select a random action $a_t$, otherwise select $a_t = max_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1}\text{, i.e. if episode terminates at step j+1} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to network parameters $\theta$, according to the
        gradient equation mentioned before
        After every C steps, reset $\hat{Q} = Q$
    end for
end for
```

4 Bootstrapped DQN

Efficient exploration remains a major challenge for reinforcement learning. We therefore study Bootstrapped DQN [9] which combines deep exploration with deep neural networks for exponentially faster learning.

Deep exploration (planning to learn) means the agent should reason about the informational value of possible observation sequences. An example of deep v/s shallow exploration can be seen in Figure 1. One such deep exploration strategy is to make algorithms have some notion of uncertainty through a distribution over possible Q values so that the agent can judge potential benefits of exploratory actions.

This notion of uncertainty is achieved in Bootstrapped DQN through random initialization (for generalization) and a network that consists of a shared architecture with K bootstrapped heads branching off independently. The shared network learns a joint feature representation across all the data, which provides significant computational advantages (Figure 2)

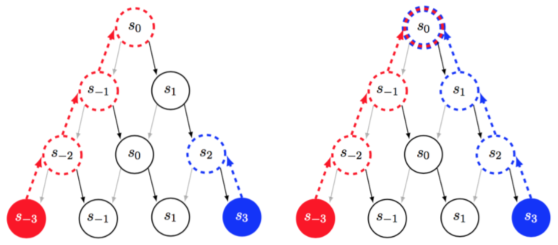


Figure 1: Shallow v/s Deep exploration
The start state is s_0, rewarding state is s_{-3}, informative state is s_3. The left figure shows an RL agent capable of deep exploitation. The right figure shows an RL agent that is capable of both deep exploration and exploitation.

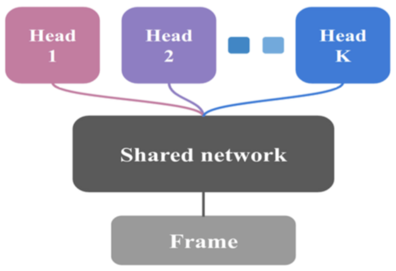


Figure 2: Bootstrapped DQN architecture

As we have seen earlier, the target value is given by

$$y_t^Q \leftarrow r_t + \gamma max_a Q(s_{t+1}, a; \theta^-)$$

A variant called double DQN modifies this target to

$$y_t^Q \leftarrow r_t + \gamma max_a Q(s_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t); \theta^-)$$

Now, Bootstrapped DQN modifies double DQN to have k value function heads $Q_k(s, a; \theta)$ each of which is trained against its own target network $Q_k(s, a; \theta^-)$.

The bootstrap principle is to approximate a population distribution by a sample distribution. The algorithm therefore approximates a bootstrap sample by selecting $k \in$ 1,..,K uniformly at random and following Q_k for the duration of that episode. In other words this is the approximate posterior sample for the Q-value.

The algorithm uses a mask m_t which decides, for each value function Q_k, whether or not it should train upon the experience generated at step t. Since the paper showed that the a variety of masks lead to the same result, we chose a simple mask m_t with all ones.

The gradients of the kth value function Q_k for the tth tuple in the replay buffer is given by

$$g_t^k = m_t^k (y_t^Q - Q_k(s_t, a_t; \theta)) \nabla_\theta Q_k(s_t, a_t; \theta)$$

The m_t^k which modulates the gradient is what produces the bootstrap behavior. The algorithm for Bootstrap DQN is shown below.

The claim the paper [9] makes is that Bootstrapped DQN substantially improves learning speed and cumulative performance across most Atari games, which is what we can see in Figures 3, 4, 5, 6

Algorithm 2 Bootstrapped DQN

```
Value function networks $Q$ with $K$ outputs $\{Q_k\}_{k=1}^{K}$. Masking distribution $M$.
Let $B$ be a replay buffer storing experience for training.
for each episode do
    Obtain initial state from environment $s_0$
    Pick a value function to act using $k \sim$ Uniform{1,...K}
    for step t = 1,...until end of episode do
        Pick an action according to $a_t \in argmax_a\ Q_k(s_t, a)$
        Receive state $s_{t+1}$ and reward $r_t$ from environment, having taking action $a_t$
        Sample bootstrap mask $m_t \sim M$
        Add $(s_t, a_t, r_{t+1}, s_{t+1}, m_t)$ to replay buffer $B$
    end for
end for
```

5 Hyperparameters

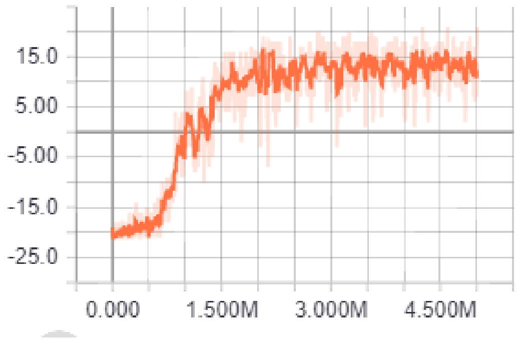
Hyperparameter	Value	Description
mini-batch size	32	Number of training cases over which Stochastic Gradient Descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	Number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The number of parameter updates before the target network is updated.
discount factor	0.99	Discount rate for future rewards.
frame skip	4	Number of frames to repeat actions for.
update frequency	4	The number of actions selected by the agent between successive SGD updates.
learning rate	0.0025	Learning rate used by Adagrad.
initial exploration	1	Initial value of ϵ in ϵ-greedy exploration.
final exploration	0.1	Final value of ϵ in ϵ-greedy exploration.
annealing exploration	1000000	The number of frames over which the initial value of ϵ is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts.
no-op max	30	Maximum number of do nothing actions at the start of an episode.
reward clipping	-1.0 to 1.0	Range around 0 to limit rewards to.
loss clipping	-10.0 to 10.0	Range around 0 to limit loss to.
gradient clipping	-10.0 to 10.0	Range around 0 to limit gradients to.
architecture	[32,64,64]	The number of filters in the conv layers.
number of heads in bootstrap	10	Number K of Bootstrapped DQN.
number of steps	5000000	Number of steps to train on.
repeat action probability	0.25	Probability of ignoring the agent action and repeating last action.
bootstrap mask probability	1.0	Probability each head has of training on each experience.

6 Simulations

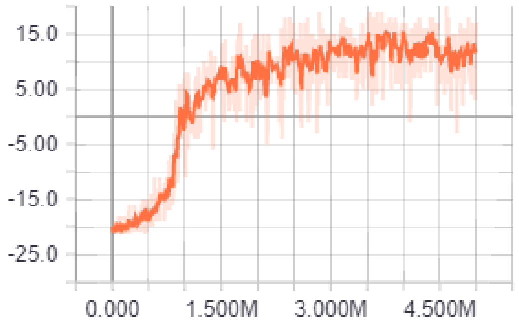
We considered 2 Atari games [2] from OpenAI [1] for simulation: Pong and Breakout [6]. We compare the performance of several metrics we obtained during the training process using tensorboard, and the evaluation of game after the learning is done. Below is a table of average scores of games after training. Like the Deep RL paper mentions [3], we took an average of two sets each of 10 runs.

Experiment	Pong	Pong + bootstrap	Breakout + bootstrap	Breakout
Trial 1 average	17.8	18	60.6	36.8
Trial 2 average	17.8	19	48	41.2

We observed that in Pong, the training is very similar for Bootstrapped and Vanilla DQN, but for Breakout, there is a significant gain of score in Bootstrapped DQN over original vanilla DQN.

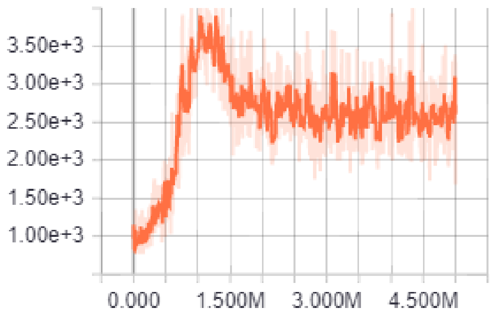


(a) Bootstrap

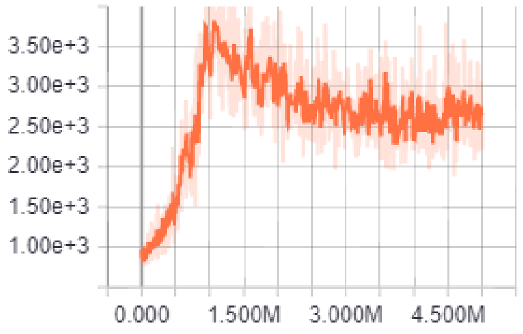


(b) No bootstrap

Figure 3: Scores vs steps taken for Pong game



(a) Bootstrap



(b) No bootstrap

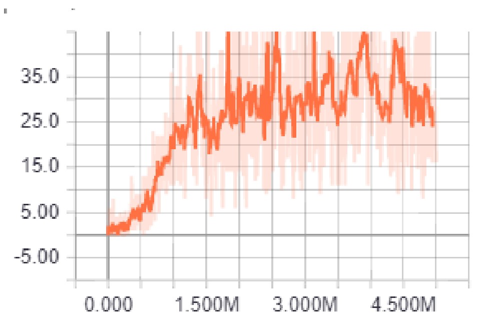
Figure 4: Alive time vs steps taken for Pong game

Percentile	Pong	Pong + bootstrap	Breakout	Breakout + bootstrap
10	-21	-20	0	2
20	-17	-16	3	4
30	4	-4	5	9
40	7	6	12	22
50	13	8	18	21
60	10	7	20	41
70	14	7	19	29
80	8	15	56	59
90	10	14	41	39
100	12	18	48	63

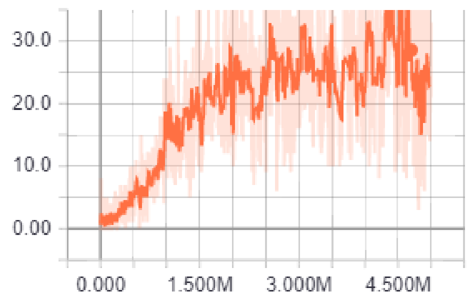
The Atari Evaluation paper [5] mentions that we need to compare different phases of training for a comparison of algorithms. We divided the total episodes into 10 buckets and took the scores from the buckets. We observed that in the case of Pong, until the later phase ,Bootstrapped DQN doesn't seem to do better than the Vanilla DQN. The effects for breakout were more easily observed.

We also looked at the tensorboard events for the training process. Specifically, we observed scores over time and the keep alive steps for each game vs the steps during the training. For Pong, from Figure 3 we can observe that the performance is almost the same at the end of training. However, we can see that the threshold score is reached earlier in Bootstrapped DQN than in Vanilla DQN. Figure 4 analyzes keep alive time for Pong. The interesting thing in this figure is that initially the agent loses to the second player in Pong and hence has a higher alive time. Gradually, the agent learns to defeat the other player and the game ends sooner thereby reducing the alive time. Again, here we can see that Bootstrapped DQN learns it quicker than the normal DQN.

For Breakout, from Figures 5, 6 it's more easier to see that bootstrap has a better score and a higher keep alive time than the normal DQN. Also the keep alive time keeps increasing in this game unlike Pong as there is only one player here.

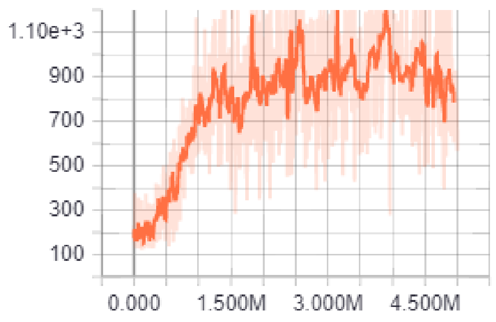


(a) Bootstrap

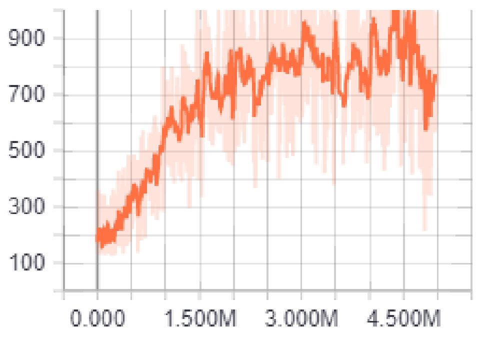


(b) No bootstrap

Figure 5: Scores vs steps taken for Breakout game



(a) Bootstrap



(b) No bootstrap

Figure 6: Alive time vs steps taken for Breakout game

7 Acknowledgements

We would like to thank Dr. Alekh Agarwal for his continuous support and guidance throughout this project. We relied on his feedback and experience to concisely scope the objective and carry out relevant experiments.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Prafulla Dhariwal, Christopher Hesse, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

[3] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*, 2017.

[4] Arthur Juliani. Q-learning with tables and neural networks, August 2016. [Online; posted 25-August-2016].

[5] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *arXiv preprint arXiv:1709.06009*, 2017.

[6] Brendan Maginnis. atari-rl. `https://github.com/brendanator/atari-rl`, 2017.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529 EP –, 02 2015.

[9] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In *Advances in Neural Information Processing Systems*, pages 4026–4034, 2016.