Reinforcement Learning Project

Ashraf Elnaima
aelneima@aimsammi.org

Gbetondji Dovonon
gdovonon@aimsammi.org

Lynn Kelly Tchoufa
lndjate@aimsammi.org

Soona Osman
sosman@aimsammi.org

Yvan Biakeu
ybiakeu@aimsammi.org

September 2021

INTRODUCTION

Q-learning is a method used to solve reinforcement learning problems when the state space and action space are both discrete, and also when the environment is not complex. It was coupled with deep learning which led to deep Q-learning making it possible to apply a Q-learning like method to more complex state spaces. We implement deep Q-learning and test it on multiple gym environments such as `LunarLander-v2` and `LunarLanderContinuous-v2` and `Pong-ram-v0`. We compare it with the deterministic deep policy gradient for the continuous environment and study the resilience of our implementation with respect to the learning rate, epsilon decay and batch size.

1 PROBLEM

1.1 LunarLander-v2

The task in `LunarLander-v2` is to land a space-ship two flags on a landing pad without crashing. The ship has 3 engines, the main engine points downwards and controls the vertical movement. The two secondary engines are placed on the left and the right and control the horizontal movement. The state space is represented by an 8-dimensional continuous vector, but the action space is discrete. There are 4 possible actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. The state vector at any given time contains the following values: x coordinate of the ship, y coordinate of the ship, the horizontal velocity, the vertical velocity, the orientation in space, the angular velocity, whether the left leg is touching the ground (Boolean), whether the right leg is touching the ground (Boolean).
Most of the complexity in solving the `LunarLander-v2` environment comes from the delay in the reward. Most of the reward during the flying phase is between -1 and +1. The chunk of the reward is only obtained when landing or getting close to landing (a reward of +10 is obtained when the ship's leg touches the ground). This applies to both positive or negative rewards when landing/crashing.

1.2 LunarLanderContinuous-v2

`LunarLanderContinuous-v2` is a version of `LunarLander-v2` with a continuous action space. While the engines of the space ship in `LunarLander-v2` are controlled by a discrete action with 4 possible values, in this case there are two actions each in a space from -1 to +1. The first continuous action controls the main engine, turning it on if the value is positive, with the engine's power output being proportional to the value. The second continuous action either triggers the left engine (-1 to -0.5), the right engine (+0.5 to +1) or none (-0.5 to +0.5). The states and reward structure is the same as that of `LunarLander-v2`.
`LunarLanderContinuous-v2` is more complex than `LunarLander-v2` mainly because of the difference in action types. The state complexity and reward delay are the same.

1.3 Pong-ram-v0

`Pong-ram-v0` is the Atari 2600 game Pong. In this environment, the state is the RAM of the Atari machine. Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from $\{2, 3, 4\}$. Atari 2600 has only 128 bytes of RAM. On one hand, this makes our task easier, as our input is much smaller than the full screen of the console. On the other hand, the information about the game may be hard to retrieve.
We used this environment to test the limits of our implementations since its general complexity is above that of `LunarLander-v2` and texttttLunarLanderContinuous-v2.

2 ALGORITHMS

2.1 DQN

2.1.1 Description

Deep Q learning is the first deep reinforcement learning method proposed by DeepMind. It is an algorithm that combines Q learning with deep neural networks to let reinforcement Learning work for discrete, complex, high-dimensional environments, like video games, or robotics. The core difference between Deep Q learning and vanilla Q learning is how the Q value of an action-state pair is obtained. In deep Q learning, instead of using a Q table, a neural network takes in the state as input and outputs the Q values for each possible action. This serves as an approximation of the Q table.

2.1.2 Algorithm

Algorithm 1 Deep Q learning with Experience Replay: (3)

```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialise sequence $s1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    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 D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j, & \text{for } \textit{terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q^*(\phi j + 1, a'; \theta), & \text{for } \textit{non terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

2.2 DDPG

2.2.1 Description

In Deep Deterministic Policy Gradient (DDPG), we learn both a Q function that approximates the Q table and a policy. The Q function is learned using off-policy data and the Bellman equation. Similarly to Q learning and deep Q learning, DDPG relies on the fact that if we know the optimal $Q^*(s, a)$, the best action is $a^*(s)$:

$$a^*(s) = argmax_a(Q^*(s,a))$$

Solving this is simple for environments with a discrete actions spaces where we can just take the maximum value. However, on a continuous action space, we cannot exhaustively evaluate the entire space. DDPG solves that by learning a policy $\mu(s)$ and approximating $\max_a Q(s,a)$ by $Q(s,\mu(s))$

Algorithm 2 Deep Deterministic Policy Gradient: (2)

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
Receive initial observation state $s_1$
for episode = 1, M do
    Initialize a random process N for action exploration
    Initialise sequence $s1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + N_t$
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in R
        Sample a random minibatch of N transitions $(s_i, a_i, r_i, s_{i+1})$ from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i(y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
        Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end for
end for
```

3 IMPLEMENTATIONS

3.1 DQN

To implement the DQN Algorithm we wrote three classes for the network, replay buffer and agent.

- The network is made of two linear layers each followed by a ReLU function and one output layer.
- The replay buffer class is used to save the transitions during training. It has two methods: push and sample. The buffer acts as a queue where the transitions are saved in a first in - first out manner, using the push method.
- The agent class has four main methods (epsilon greedy action, episode, step, and train) and three other methods for display.
  - The epsilon_greedy_action which takes in a state and chooses an action based on the epsilon value. The higher the epsilon the most likely the agent will choose an action randomly.
  - the step method which executes one step of the DQN iteration. Here an action is sampled using the epsilon_greedy_action method. We feed the action to the environment to get a transition, push that transition to the replay buffer, sample a batch of transitions and use it to make predictions and update the Q network.
  - The episode method provides as many steps as needed until the agent is done with the environment. Here done means that the agent crashed or has landed.
  - the train method: we train the agent for a certain number of episodes. Before the first episode, since we don't have anything to sample from, we fill the replay buffer with as many random actions as the batch size.

Also note that the decay we used for the epsilon value was a linear decay. The epsilon value starts at 1 then is reduced by a decay value after every episode until it reaches a minimum value.

3.2 Modified DQN

In order to get the Deep Q learning algorithm to work with an environment with continuous actions, we modified it slightly. The main modification consists in having a Q network with two heads since there are two continuous actions. We discretize the continuous action space of values between -1 and +1 with a step size of 0.1 giving us a total of 21 possible options per action.

4 RESULTS

4.1 LunarLander-v2

The `LunarLander-v2` and `LunarLanderContinuous-v2` environments are considered solved when the cumulative reward for an episode is above 200. Our DQN implementation is able to solve the environment after a bit more than 100 episodes most of the time, however the performance is not always stable at that time. The agent is able to land consistently after 350 episodes, and after 400 episodes it lands almost all the time. The implementation seems to be quite resilient to various hyperparameter settings however the learning rate can affect the performance.

4.2 LunarLanderContinuous-v2

The modified Deep Q learning method that operates over a discretized version of the environment's action space was able to converge. The training procedure was less stable than on the `LunarLander-v2` environment. In general, with the right hyperparameters the agent is able to converge within 200 to 300 episodes and solves the environment. However even after converging it happens that a number of times it still is not able to land. This shows that the policy is not robust enough. A better training procedure including using a lower learning rate over a larger number of epochs could be helpful. We discuss the effect of various hyperparameter settings further in the algorithm resilience section.
For `DDPG` we used the implementation from the spinning up in deep RL repository from OpenAI. The results obtained by DDPG showed it was able to solve the environment within a 100 epochs. These results show that for this environment, applying the DQN on a discretized version of the action space can be a viable method to solve the environment. The results from DDPG seem more stable.

4.3 Pong-ram-v0

With the `Pong-ram-v0` environment, we do not have a threshold from which the environment can be considered solved. After training was completed, the average episodic reward over 50 episodes was -13. We tested the trained agent against the environment and it was able to win 4 out of 10 times.

5 ALGORITHMS RESILIENCE

To evaluate the resilience of our DQN implementation when it comes to various hyperparameters we run the algorithm on a grid of values for each of them. When running experiments for one of the hyperparameters, we keep the value of the other ones fixed. We evaluate the different runs using their cumulative reward and percentage of successful landing. For an episode, a landing is considered successful if the cumulative reward over that episode is not less than 200. For each of the plots in this section, we only consider the data from the 250th episode to the 500th episode. That's because the first few episodes usually have a very low reward as the agent is still learning a lot. We settled on 250 as a starting point since it will represent a point when half of the learning has been done. Here are the default values being used when the hyperparameter is not been studied:

- learning rate: **1e-3**
- batch size: **128**
- epsilon decay **1e-2**

It's also important to note that there is a relationship between the hyperparameters we study. For instance, there is a relationship between the learning rate and the batch size and they affect each other. A better way of study the effect of those hyperparameters would be to evaluate the algorithm for all the combinations of hyperparameters rather than keeping some values fixed.

5.1 LunarLander-v2

On `LunarLander-v2` the main hyperparameter affecting the performance of the agent was the learning rate. The algorithm did not seem to be heavily affected by the batch size. The epsilon decay also affected the agent, with a slower decay leading to a more stable performance but also meaning that a lot more time was spent on exploration. IN such cases, there was not always enough time to converge within the 500 episodes span we used to test all settings.

5.2 LunarLanderContinuous-v2

The `LunarLandingContinuous-v2` environment made the algorithms less resilient to the learning rate. Changes in learning rate or batch size greatly affect the ability to converge.
To conclude on the argorithm resilience, we note that the Deep Q learning algorithm had good performances on the `LunarLander-v2` environment but was sensitive to the value of the learning rate. On the `LunarLanderContinuous-v2` we notice a lot more instability which could be due to the two-headed structure of the Deep Q network.

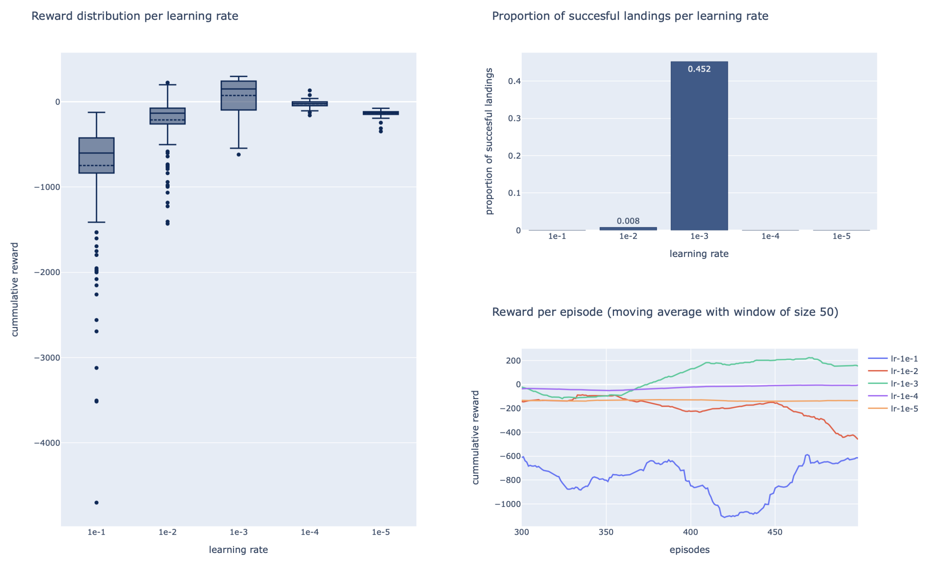

Figure 1: Effect of the learning rate on `LunarLander-v2`. The performance is much better using a learning rate of 1e-3, as shown by a landing success rate of 45.2% which is much better than all the other values. The learning rate heavily impacts the performance.

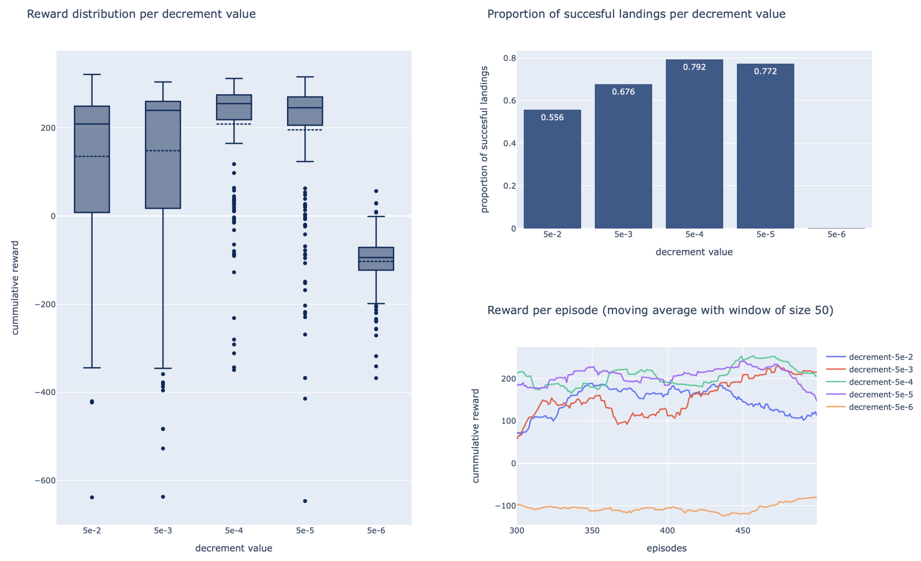

Figure 2: Effect of the epsilon decay on `LunarLander-v2`. We see an improvement in the stability of the training procedure and the landing success rate as the decay value gets smaller (more exploration). The training is more stable (smaller spread on the box plot indicating a lower variance). The exception is a decay of 5e-6 per episode. Since that value is too small that means the agent spent most of it's time epxloring using random actions.

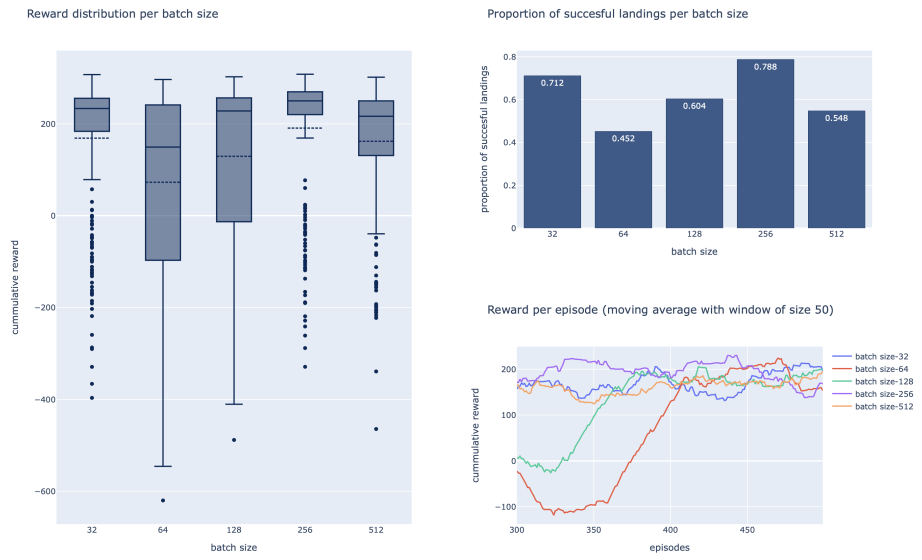

Figure 3: Effect of the batch size on `LunarLander-v2`. The performance is quite consistent accross batch size. Larger batch sizes tend to lead to more stable training.

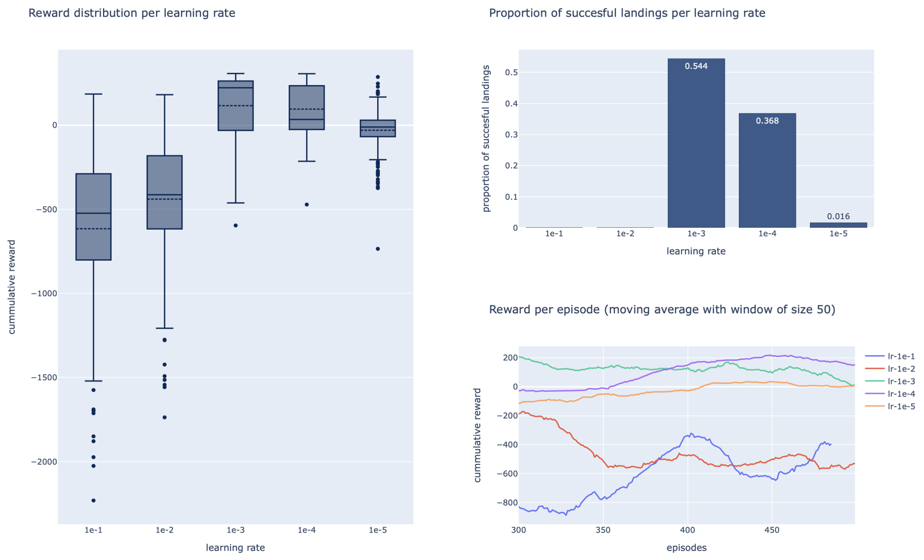

Figure 4: Effect of the learning rate on `LunarLanderContinuous-v2`. The agent overfits when the learning rate is too large and converges slowly with a small learning rate. The best value is 1e-3.

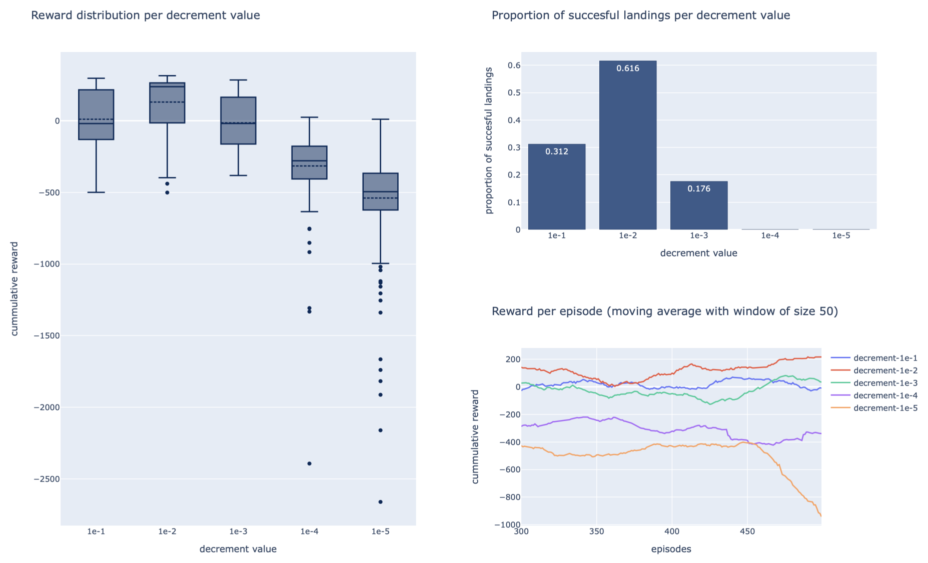

Figure 5: Effect of the epsilon decay on `LunarLanderContinuous-v2`. The best value is 1e-3. Values smaller than 1e-4 do not lead to acceptable performances.

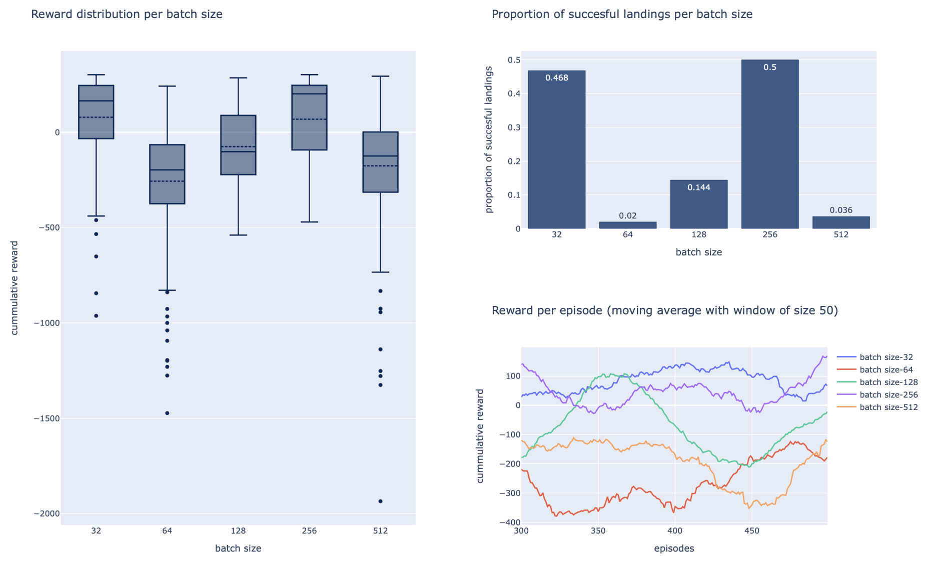

Figure 6: Effect of the batch size on `LunarLanderContinous-v2`. The performance is quite inconsistant accross batch sizes which could be due to the unstability of the training method.

CONCLUSION

The goal of this work was to implement the deep Q-learning method on the `LunarLander-v2` environment. It later extended to `LunarLanderContinuous-v2` and `Pong-ram-v0`. We also tested the resilience of our implementation with respect to the learning rate, the epsilon decay and the batch size.
Our DQN implementation as well as its modified version were able to solve `LunarLander-v2` and texttttLunarLanderContinuous-v2. On the `Pong-ram-v0`, we were able to train an agent that wins 4 out of 10 games.
Concerning the resilience of the implementation with respect to the hyperparameters, the learning rate is the one with the heaviest effect. Also applying deep Q-learning on a continuous environment by discretizing the action space leads to less stable performances.

References

[1] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym. *arXiv e-prints* (June 2016), arXiv:1606.01540.

[2] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

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