Deep Reinforcement Learning - Wet Exercise 1

Ben Kretzu - 200728947, Lior Aloni - 200616274

December 2018

1 Taxi Environment

1.1 The environment

The environment are include world which is of size 5X5, 4 stations, a passenger and a taxi. The goal is to take the passenger from origin station to destination station. In this problem we have 500 states $\mathcal{S}$ and 6 actions $\mathcal{A}$.

$$S = \{25 \text{ Taxi's location, } 5 \text{ Passenger's location, } 4 \text{ Target station}\}$$

$$A = \{\text{'south','north','east','west','pickup',drop-off'}\}$$

1.2 Fully connected DQN

1.2.1 Background

$\mathcal{Q}$-learning algorithm's goal is to learn the action-value function, $\mathcal{Q}$, of some policy π. For our problem we determinate the equation for deterministic MDP's:

$$Q^*(s_t, a_t) = r(s_t, a_t) + \gamma \max_{a' \in A} Q^*(s_{t+1}, a')$$

Where Q^* is the optimal value.

1.2.2 Architecture

Our aim is to learn $\mathcal{Q}$, for this purpose we used deep neural network with one hidden layer. The input dimension of one-hot encoding is 500 (size of $\mathcal{S}$), the output dimension is 6 (size of $\mathcal{A}$).

We estimate $\mathcal{Q}$ which depends on $\theta = \{\mathcal{W}_2, \mathcal{W}_1, b_1, b_2\}$.

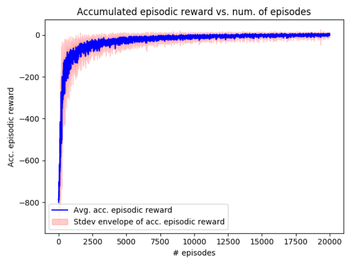
$$Q_\theta(s, :) = \sigma_2(\mathcal{W}_2 \sigma_1(\mathcal{W}_1 s + b_1) + b_2)$$

Where $\sigma_1(x) = ReLU(x)$, $\sigma_2(x) = x$. In this section we examine the learning process as function of the hidden layers size - 16, 64 and 256. Memory buffer taking place, which hold transitions from the environment. In the optimization process the network sample under uniform distribution batch of examples after every step and update the network weights with respect to the sampled batch. We decided to use the Huber-loss as objective loss function. Memory buffer updates with new transition by Last In First Out (LIFO) method. Each action step we used ϵ-greedy policy, i.e. for every time step the action is randomly w.p. ϵ and w.p. 1-ϵ the best action estimated from the policy-net was taken.

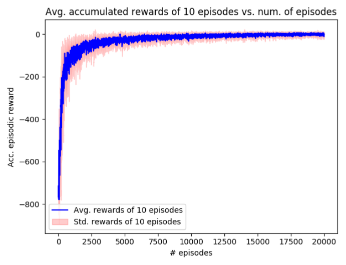
$$transition = \{\mathrm{s}_t, a_t, r_t, s_{t+1}\}$$

Each episode terminate at two conditions: first, drop-off the passenger in the desired location, second after preforming 200 steps (actions).

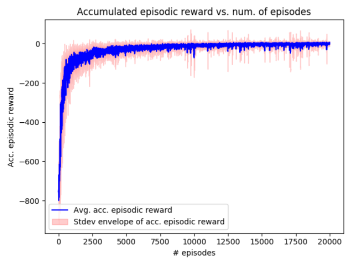
The hyper-parameters we used are: experience replay memory buffer of 20000 transitions, batch-size of 512, γ=0.999, ϵ start in 1 and exponential decay to 0.05 by $\epsilon_t = 0.05 + (0.95 - 0.05)e^{\frac{-t}{200k}}$. As DQN architecture in [1], two networks taking place, one for target and second for policy, we update the target-network weights every 500 steps.
Optimization algorithm is RMSprop with 0.01 learning rate and $\alpha = 0.99$.



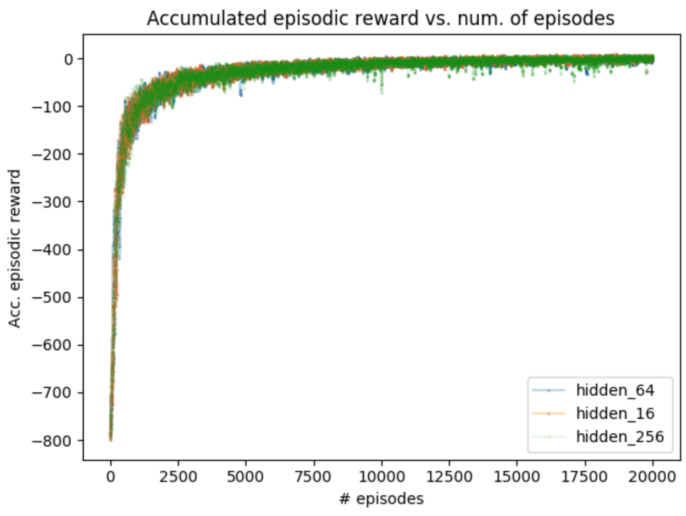
(a) Hidden layer of 16 neurons.



(b) Hidden layer of 64 neurons.



(c) Hidden layer of 256 neurons.

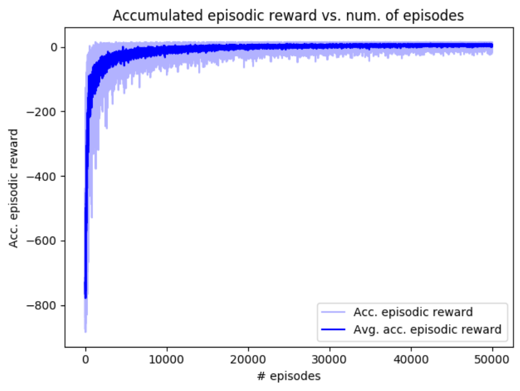


(d) Comparison between different hidden layer sizes {16, 64, 256}.

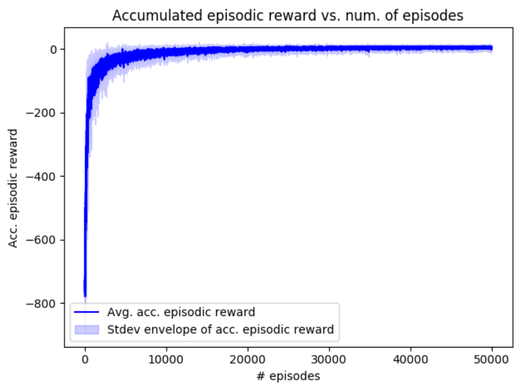
Figure 1: (a),(b) and (c) present average reward and STD of 3 different sizes of the hidden layer 16,64 and 256 respectively. Average and the standard deviation calculated over the 10 last episode's. (d) present comparison between the average reward of the different sizes.

In Fig. 1 we can see that all networks are converge to solution - average reward upon 10 episode of $+5 \pm \{3 - 8\}$, we decided to use hidden-layer with 64 neurons because we get the maximum average reward and lower standard deviation.

After we chose the size of hidden layer, we aim to achieve that the agent will be able to solve every possible state of the environment and getting the best reward, therefore we set the limit to $50k$ episodes and save the model for the best reward over all possible state. By reaching the goal that the agent will solve any state means that the agent will solve any future test. The trade-off in choosing the hidden layer size have some aspects, if the hidden layer have larger size it imply to the network more expressive abilities, however longer and complex training with more computational resources. In Fig. 2 presented the training process.



(a)



(b)

Figure 2: (a) Present the reward per episode and average reward on 10 episodes, in (b) Present the average and the standard deviation of 10 last episode's reward.

1.2.3 Dropout and L1-Regularization

In this section we tried to optimize the best architecture from the previous section with dropout on both layers and L1 regularization with $\lambda = 0.001$.

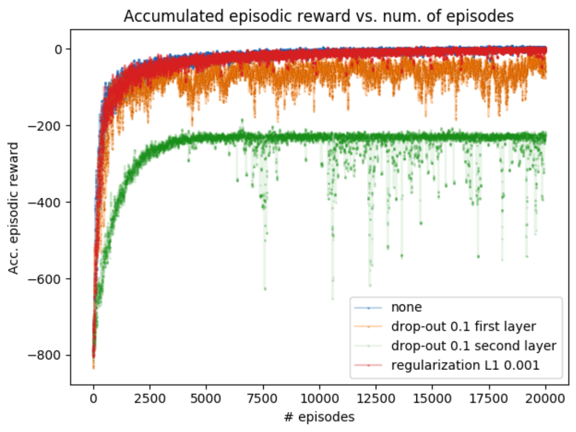


Figure 3: Comparison between episodes reward over L1-regularization dropout in first layer, dropout on the second layer and plain network.

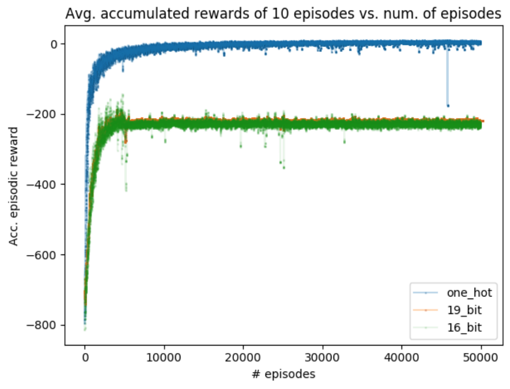
In Fig. 3 we can notice that both dropouts significantly deteriorate the accumulated reward. L_1 regularization restrict the weight updates, therefore the reward variance is a bit smaller but overall we got a little bit smaller average reward in comparison with plain network. Those results make sense in this problem, because the solution is model-based which mean that we aim to learn the whole model and every possible state, hence there is no any additional value for generalization (in counter to classification models), furthermore we have redundancy in parameters so we can achieve over fitting which is good because there is no any new information that the network didn't took in account.

1.2.4 Encoding

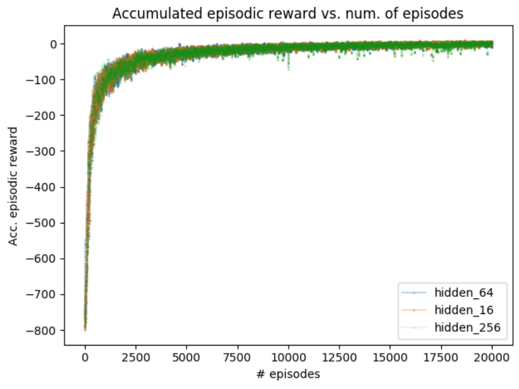
In this section we suggest alternative representation to the state. So far we used one-hot encoding technique as guided in the instruction sheet, one-hot encoding means that the vector size is the same size as all possible state, i.e. 500 as described above. All values are zero beside the current state index which equal to 1. The two alternative compact encoding that we have examine:

1. 19 vector encoding: 10 bits allocated for taxi location, 5 for current row and 5 for column with value of 1 in the correct row/column, 5 bits allocated to represent the location which the passenger can be and the last 4 allocated to represent the desired dropout stations.

2. 16 vector encoding: in more compact representation, we allocated the taxi location as the same as the 19 bit encoding, 2 bits allocated to represent if the passenger is on the taxi or waiting for pickup. The last 4 allocated to represent the desired location of the taxi to reach, meaning that the desired location can be pickup or drop-off respectively to the 2 bits status of the passenger.



(a)



(b)

Figure 4: Training process with difference encoding. Blue is one-hot encoding, red is 19 bit encoding and green is 16 bit encoding. (a) Networks with hidden layer of 64 neurons. (b) The 19 and 16 bit encoding net is of 256 neurons in the hidden layer.

In Fig. 4 we can observe in the training process comparison off all encoding methods. In (a), all network's architecture are the same, we can notice that one-hot encoding out-preformed, even more both 19 and 16 bits encoding are unable to solve the task by reaching score of -200. The reason is the number of parameters of the network, one-hot encoding have much more parameters (weights) because the input size is of size 500 (more expressive abilities to the network) and we can see that it take more episode to reach steady state in the learning process for the one-hot encoding. Therefore, we enlarge the hidden layer for the 19 and 16 bit encoding. In (b) we can see that the all methods converge to the same average reward. Additionally, one-hot encoding take much more memory because every state is presented with more bits, in the other hand, its more straight forward technique, less complexity to represent the state and better learning process (smaller varinace). Overall we got better accuracy in one-hot encoding.

1.2.5 Optimizer

In this section 3 optimizers are examine to optimize the DQN, we choose to discuss about SGD, RMSProp and Adam. In DQN optimization process, we want to minimize the objective function ($\mathcal{L}$) which in the update time-step depend only on the examples in the batch loaded from the memory buffer, i.e. the process only estimate the loss function subject to single batch, therefore the learning process tend to be extremely fluctuated.

First, lets elaborate on the selected optimizers:

- SGD - Stochastic Gradient Descent: The update rule over batch size N

$$\theta_{t+1} = \theta_t - \frac{\eta}{N} \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(\theta_t, x_i)$$

  Where η is the learning rate and x_i is sample from the batch. SGD is efficient simple approach to optimize the objective function.

- RMSProp - use adaptive learning rate individually to each parameter. The update rule is as follow:

$$g_{t+1} = \alpha g_t + (1-\alpha)(\nabla_\theta \mathcal{L}(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{g_{t+1} + \epsilon}} \nabla_\theta \mathcal{L}(\theta_t)$$

  Where η is the initial learning rate, α is smoothing parameter and ϵ for numerical stability. For clarity g_t is a vector and the product is element wise.

- ADAM - use both first and second moment of the gradient to construct adaptive learning.

$$m_t = \frac{\beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta_t)}{1-\beta_1^t}$$
$$v_t = \frac{\beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta \mathcal{L}(\theta_t))^2}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\tilde{v}_{t+1}} + \epsilon} m_t$$

  where β_1, β_2 are the average ratio in the moments, η is the initial learning rate and ϵ is for numeric stability.

Both RMSProp and ADAM based on batch computation, and for convenient reason we omit the sum notation. Additionally we add momentum to all optimizers.

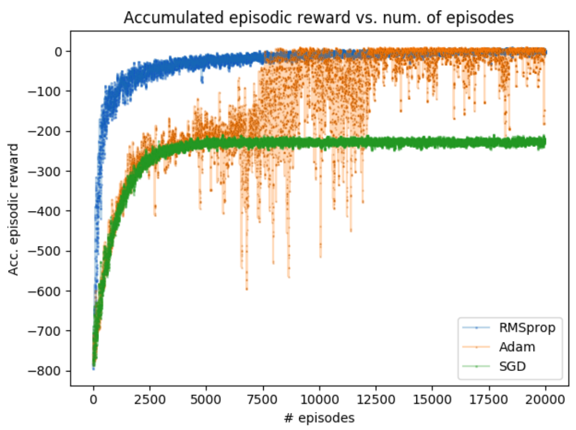


Figure 5: Training process over different optimizers - SGD, RMSProp, and ADAM.

In fig. 5 presented the accumulated reward in the learning process for the optimizers which described above, for our pick to the hyper-parameters is notable that RMSProp out preformed. SGD have fix learning rate, which give a rise to the poor converges reward but have small variance in the reward. We expected that ADAM and RMSProp will have similar results. Probably we didn't tune right ADAM optimizer and find the sweet point. We can notice that ADAM is more noisy and less stable but eventually almost converge to the same average results.

1.2.6 Policy Gradient - Actor-critic

In contrast to DQN, which the objective is to maximize the expected reward with policy π, policy gradient (PG) try to learn the best policy (π_θ) to maximize the expected reward following a learned policy. In this section We chose Actor-Critic (AC) as PG method. Here we used the advantage function and the time difference (TD) error as approximation to the advantage function:

$$A(s, a) = r(s, a) + \gamma V(s') - V(s)$$

Where $V(s)$ is the state-value function. Recall that the reward function:

$$J(t) = \mathbf{E}_\pi \Big[\sum_t^\tau \gamma^t r(s, a) \Big]$$

The network build to approximate the policy π_θ and the state-value function $V_\theta(s, t)$ and as a result approximation of the advantage function A_θ. The network objective function is as follow:

$$\mathcal{L}(\theta) = -J_\theta + Huber(A_\theta)$$

Where Huber means the huber loss function. The networks' (Fig. 6) architecture is one fully connected hidden layer with shared weights for the output layer and ReLU activation function. The output layer have two section first is 6 FC for policy and the second is a single neuron for state-value function. The input is one-hot encoding to the environment state. The optimizer is ADAM.

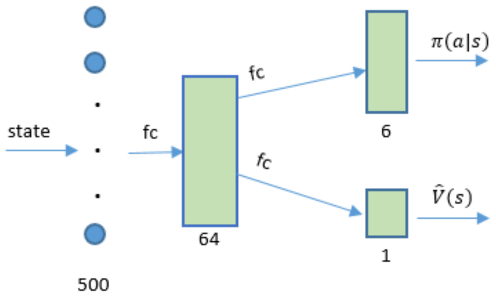
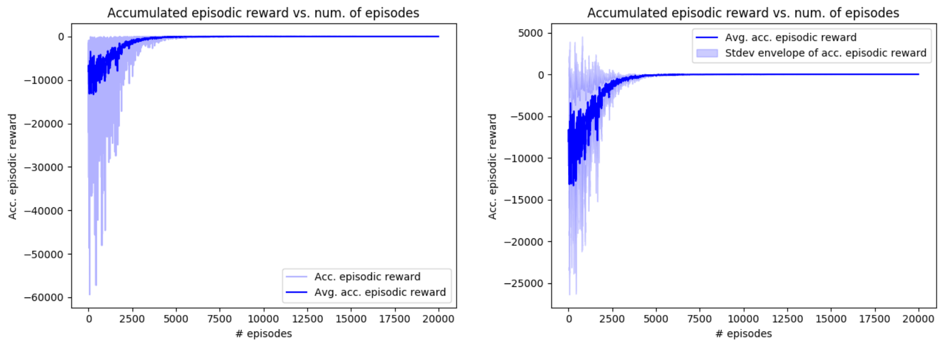


Figure 6: Architecture.



(a) (b)

Figure 7: The actor critic learning process, (a) present the reward per episode and average reward on 10 episodes, (b) present the average and the standard deviation of 10 last episode's reward.

In each time step we sample action from the policy $a_t \sim \pi_\theta(s_t)$, we store the reward and got single transition (a_t, s_t, s_{t+1}, r_t). In the end of the episode we compute the state-value function in each time step s.t. current policy and optimize the network. We use Huber-loss in a way to reduce the sensitivity of large deviations and robust the learning. We notice if the agent didn't experience drop-off of the passenger in the desired location the agent didn't achieve good performance, therefore we changed the episode termination condition to last till correct drop-off and not up to 200 steps. which turn to be a key change for the agent not to get stack. In the other hand it's took much longer time to converges because every episode can be much longer than 200 steps. Additionally, we normalized the accumulated reward for more efficient training. In Fig. 7 presented the accumulated reward in the training process. Eventually we got similar preference as DQN had in the previous section.

2 Acrobot Environment

2.1 The Environment

The acrobot system includes two joints and two links, where the joint between the two links is actuated. the task is to swing the end of the lower link up to a given height, each action imply negative reward until reaching the desired height. Three actions are exist: applying +1, 0 or -1 torque on the joint between the two links. We chose to represent the state as the pairwise difference between the current environment state (image print screen) and the next state after taking action.

$$x_i = \text{print screen of the environment at time-stamp } i$$

$$S_i = \{x_{i+1} - x_i\}$$

$$A = \{\text{Right torque, Zero torque, Left torque}\}$$

2.2 Preprocessing

The acrobot position was given by the environment by 500x500 RGB frame. The frame ain't hold a lot of information, i.e most of the pixels are white with no valuable information and the only important information is the location of the acrobot arm in the frame. Therefore we resize the frame to 40x40 RGB frame. We didn't transform the frame to gray-scale because the running time was reasonable. Another ease reason that we don't need any preprocessing is the pixels values are similar for any scenario and the varians is small, meaning all frames have the same distribution. The reason we chose to represent the state (s_i) as the pairwise difference between frame is low in computational resources in comparison to a sequence of frames and have good ability to express the arm motion.

2.3 CNN Architecture

We build a basic CNN architecture, three convolutional layers, after each layer we used max-pooling of size 2 and batch normalization. All layers convolves with 3x3 filters, first and second layer consist 32 filters and last layer consist 64 filters. On top, we concatenate fully connected layer with 576 neurons. In all convolutional layers we used ReLU as the activation function and in the final FC layer we used linear activation function. In most regions in the state there is not any valuable information as motioned above, and spatial resolution a relatively small (40x40), therefore we choose to convolve with small kernel sizes (3x3) and eliminate all none important regions with max-polling of size 2. The weight's initialization is sampled uniformly in the range of the absolute square root of the number of nets' parameters. In general we tried to build a compact net in order to reduce training time and lack of computational resources.

2.4 Robustness and Regularization

In order to reduce the influence between each hidden layer we used Batch-Normalization method [2]. When we first tried to tackle this problem we used a basic DQN [1] with simple experience replay. We failed to train the network even were we expand the memory buffer to the maximum we can and the number of episodes to reach some reasonable performance wasn't achievable. To accelerate the training process and to increase the robustness of the algorithm mainly due to scarce positive reward we used prioritized experience replay buffer [3]. While using prioritized buffer we can get a problem of large gradient therefore we reduce the learning rate.

2.5 Training

The optimizer is RMSPROP with learning rate of $\eta = 0.00025$, batch size of 64. Experience buffer of size $100k$. We used epsilon-greedy with linear decay, from 1 to 0.05 to keep relatively high exploration

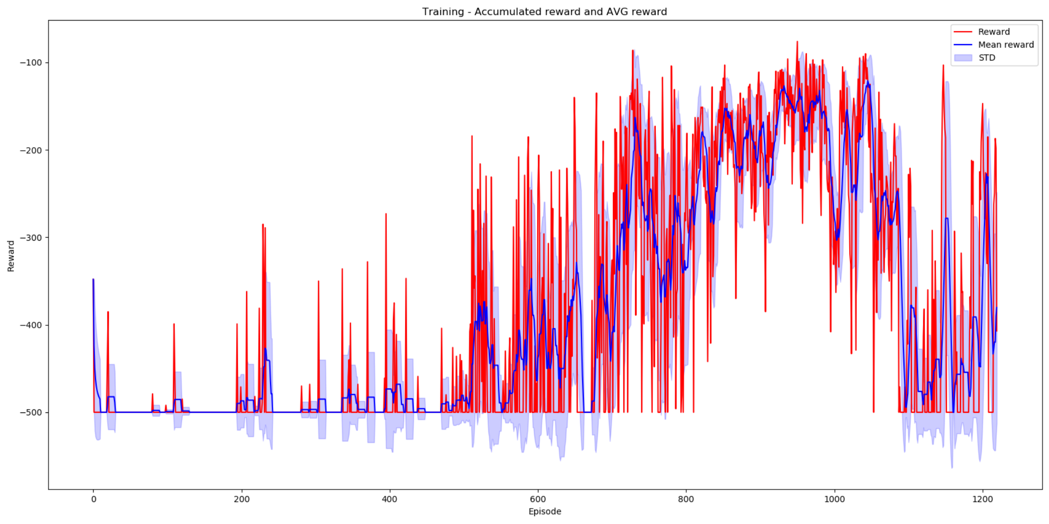


Figure 8: Acrobot training, reward (red), average reward (blue), STD of average reward. The average and STD compute on sliding window of 10 episodes.

rate during all training process. The First $50k$ action was pure exploration ($\epsilon = 1$) for initial reaction with the environment and filling the memory buffer, following linear reduction till $250k$ actions. Additionally, the learning process began only after completing the pure exploration phase.

In Fig. 8 presented the training process evolution. in the first 100 episodes, when the pure exploration phase the agent achieved some positive reward which really contribute to the initial learning process. We can notice during all learning process the variance of the reward is relatively high which indicate an stable learning process. Furthermore we can see around the 1000^{th} episode the reward deteriorate and the network forgetting everything, the average reward significant low and with high variance. Additionally we can say that actual learning take effect around the 400^{th} episode which indicate for long learning process. We set the network weights for the evaluation stage with the network weight on the episode with the highest average reward and the lowest variance (episode 933)

2.6 Evaluation

We evaluate the acrobot agent on 10 test episodes, we gain average reward of -121.3 ± 19.8.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013.

[2] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *CoRR*, vol. abs/1502.03167, 2015.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” *CoRR*, vol. abs/1511.05952, 2015.