

Playing Coinrun with Q-Learning

Sunil Kumar

Uvic

BC, Canada

sunilkumar@uvic.ca

Abstract

We have seen Proximal Policy Optimization (PPO) a improve version of Policy Gradient method of Reinforcement Learning successfully working on environments like Coinrun in the paper "Quantifying Generalization in Reinforcement Learning" [3]. The paper explores the effect of known generalization techniques like dropout, batch, etc. and different NN architecture on PPO. As my project, I will be applying Q-learning on the same environment Coinrun and see how good is this technique. More specifically I will be applying a Deep Q-Network (DQN) and its variant with the intention of understanding the strength and weakness of the different Deep Q-Learning techniques.

1. Introduction

In the 1950s, behavioral psychology gave birth to the main idea behind Reinforcement Learning (RL), that all animals learn to perform better in specific tasks if they are given positive rewards after the completion of such tasks [14]. Keeping this idea in mind, RL has become a new sub-discipline of Artificial Intelligence (AI). RL is different than existing Machine Learning techniques. For example, RL is different than Supervised Learning because the feedback in RL is not complete, and most rewards are delayed, i.e., a reward is given after doing multiple steps instead of each step. It is also not an Unsupervised Learning because the goal of RL is not to find a hidden pattern in unlabeled random data but to maximize the reward solely. Initially, the applications of RL were limited to simple low-dimensional problems. However, with the increase in computational power in the last decade researchers can work on large data-sets.

The current decade has seen rapid growth in Artificial Intelligence. For instance, Google's DeepMind team created the AlphaGo algorithm which won against Lee Sedol during a 5-game tournament in the game of Go [12], setting a new landmark in AI research. Later they developed AlphaGo Zero [13] using only Self-Play with RL, which does not use any dataset or human knowledge and it performed

better than original AlphaGo. Then, there is OpenAI team (largely funded by Elon Musk) made AI for Dota2¹ 1v1 in 2017² which defeated the human world champion. Further, they built OpenAI Five³ which as of 13 April 2019 have won against human world champion team and many other pro teams in a 5v5 game. Also recently, DeepMind came up with AlphaStar [1] which is the first AI system to defeat a top professional player at the game of StarCraft II⁴. I have a passion for games and Computer Science, and the research on AI for games has really pumped me up.

Currently, RL is a very hot field of study. Hence, many RL based environments have been created to help researchers develop and compare their RL algorithms. For example, StarCraft II Learning Environment [17], Boomarm⁵, OpenAI's Gym, Atari 2600 Games [7, 8] etc. Coinrun is one of such environment developed by OpenAI's team in late 2018 as part of their experiment for the paper 'Quantifying Generalization in Reinforcement Learning' [3].

The goal of my project will be to understand how different Q-Learning techniques are actually works by implementing as many as I can before the project deadline. It will be interesting to understand the reasons behind OpenAI's decision to use Policy gradient methods instead of Q-Learning. I will be taken help from different papers to apply Q-Learning techniques on Coinrun environment. For example, DQN was applied to play Atari games [8] and a standard way of applying DQN was purposed later [10].

2. Theoretical Background

Reinforcement Learning problems are commonly expressed using Markov Decision Processes (MDP). A MDP is a model consisting of a set of all possible states S , all

¹<https://en.wikipedia.org/wiki/Dota.2>

²OpenAI Dota 2 1v1 Bot, 2017. <https://openai.com/the-international/>

³OpenAI Five Final, April 2019. <https://blog.openai.com/openai-five-benchmark-results/> <https://openai.com/blog/openai-five-finals/> <https://openai.com/blog/how-to-train-your-openai-five/>

⁴https://en.wikipedia.org/wiki/StarCraft_II_Wings_of_Liberty

⁵Pommerman: AI research into multi-agent learning, 2018. <https://www.pommerman.com/>

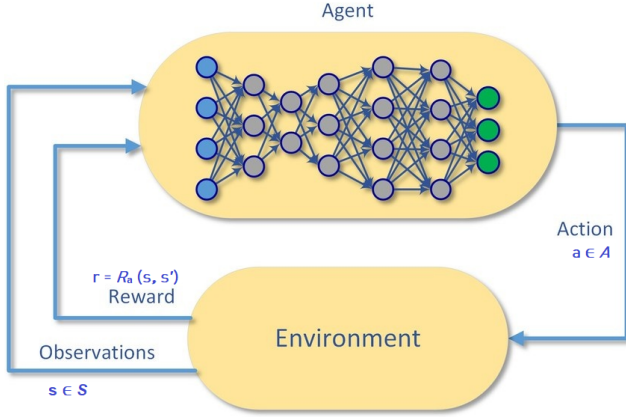


Figure 1. An illustration of the interaction between the agent and the environment through states/Observation, actions and getting reward based on the action. Note when the agent is a Neural network it becomes Deep Reinforcement Learning

possible actions A and a reward function $R_a(s, s')$ for each possible state transition T . States are memory-less and contain all relevant information about the past (also known as the Markov Property) [14].

Reinforcement learning deals with how an agent finds an optimal policy, $\pi(s, a) \in \Pi$, i.e., a series of action a at state s , which maximize the expected cumulative reward

$$\sum_{t=0}^n \gamma^t R_a(s_t, s_{t+1}) \quad (1)$$

by interacting with the environment, where R_a is the reward received for action a , which happened at time step t and the state changed from s to s' . γ is a parameter, $0 \leq \gamma \leq 1$ called the discount rate [14]. The concept of discounted reward is similar to the concept of money in finance i.e. getting 100 \$ tomorrow are more valuable than getting 100 \$ one year later. Note that each time the agent interacts with the environment, we obtain a data point $\langle s, a, r, s' \rangle$.

Many deep learning based solutions have been proposed to find the best policy for MDPs, such as Monte Carlo search [2], Policy Iteration [15] and Q-learning [19]. Let us see a few of these in detail.

2.1. Deep Reinforcement Learning

Deep Learning [6] is a technique of using a Neural Network model to solve a complex problem like speech recognition, visual object recognition and so forth. When Deep Learning is combined with reinforcement learning, then it is called Deep Reinforcement Learning [5].

The two major types of Deep RL are Model-based learning and Model-free learning. In Model-based learning, the agent is required to learn a model which describe how the environment works from its observations and then plan a

solution using that model. Model-free learning is when an agent can directly derive an optimal policy from its interactions with the environment without needing to create a model beforehand. The most common methods for Model-free learning are Q - Learning and Policy Gradient. Both of these methods have many variants available today like DQN, DDQN, TRPO, PPO, TRPPO, etc.

2.1.1 Q-Learning⁶

First of all, Q function is defined as the expected value of the sum of future rewards by following policy π .

$$Q^\pi(s, a) = E[R_t] \quad (2)$$

Here $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$. The idea of Q Learning is that if we have a good approximate of Q function for all state and action pairs, we can obtain the optimal policy $\pi^*(s) = \operatorname{argmax}_a Q(s, a)$.

Now we have data points $\langle s, a, r, s' \rangle$ to approximate Q function. The Bellman equation will help us relates the Q functions of consecutive time steps

$$Q^{\pi^*}(s, a) = r + \gamma \max_{a'} Q(s', a') \quad (3)$$

So, bellman equation become our target value and our predicted value will be the $Q(s, a)$ (initially random values). This give us the loss function

$$L = \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \quad (4)$$

For simple environments, we do tabular Q-Learning where we use a table SXA , which can be initialized with random values and as the agent's interaction with the environment we can update the estimated value of Q function in that table. This process is also known as Boltzmann exploration. But for a problem with larger state-action search space, we use a Deep Neural Network to approximate the Q function. This is called Deep Q-Network i.e. DQN. I am hoping of playing Coinrun with the native DQN or with a slightly improved version of it like Double DQN or Dueling DQN.

2.1.2 Policy Gradient

While the goal of Q Learning is to approximate the Q function and use it to infer the optimal policy $\pi^*(s) = \operatorname{argmax}_a Q(s, a)$, Policy Gradients (PG) seeks to directly optimize in the policy. By using a Deep Neural Network to directly predict probabilities of actions at a particular state. To come up with the best policy, we choose a set of actions and then adds randomness to those actions. If the new total

⁶ Some part of the text is taken from <https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html>

output reward is better, then we increase the likelihood of those actions. The same thing can be translated from the improved loss function of PG:-

$$L^{PG}(\theta) = \hat{E}_t \left[\log \pi_{\theta}(a_t | s_t) \hat{A}_t \right] \quad (5)$$

Here \hat{A}_t is the advantage function which defined as the difference of two terms Discounted Reward (Eq. 1) and Baseline Estimate of reward at a point. So, +ev \hat{A}_t mean we have collected higher reward than expected and hence the loss function help us increase the action probabilities of those actions and vice-versa. Hence, the Advantage term is used as the Critic to evaluate the goodness of an action hence corresponding gradient estimation method is known as Advantage Actor-Critic (A2C). Here the policy network acts as the Actor.

Generally, an environment has multiple possible actions with one ultimate final goal for which reward is given. This causes a problem known as Sparse Rewards System [9] which arises mainly due to too many actions needed before receiving the reward, which makes it really hard by just doing random action to get a successful reward with a +ev gradient to decent toward the optimal solution. To tackle this sparse reward system, we do reward shaping (manually design rewards for some +ev actions) which does make it easy to converse to a solution. But there are downsides of reward shaping also. 1st, it's a custom process it has to be re-done for every new environment; for example, all Atari games require separate personalized optimal Reward Shaping. So this is also not scalable.

2ndly, Reward Shaping also gives raises to "The Alignment Problem"⁷ in which agent will find a lot of ways of getting the small useless reward without doing the ultimate goal. And lastly, this reward shaping is optimized based on human behaviors which might not be optimal for many environments for example in the game of Go where we actually want to surpass human knowledge.

3. Contribution of the paper "Quantifying Generalization in Reinforcement Learning" [3]

1. New environments for experimenting for other researchers i.e. Coinrun and Coinrun-platform.
2. They showed that the number of different training levels required for a good generalization is much larger than the number used by prior work on training in RL. Gave a metric based on experimenting with 3 different environments.

3. Generalization metric using the Coinrun environment. This could help future researchers who plan to use Coinrun to setup there technique accordingly.
4. They evaluate the impact of 3 different convolutional architectures (Nature-CNN [8], IMPALA [4] IMPALA Large) and methods of generalization including Dropout, L2 regularization, Data augmentation, batch normalization, and Stochasticity.

4. Contributions of this project

Project Goal: It's my first Reinforcement Learning implementation so I decided to start from scratch to have an intimate understanding of every step of the process of a DQN algorithm. I will be using Pytorch instead of tensorflow which is used by the paper.

I am using Coinrun environment which seems to have a different interface than other Arcade Learning Environments like OpenAI gym. Turn out Coinrun is a vector environment which means that it takes a list of inputs and return a list of states and rewards. I find out that this kind of environment can be helpful in policy gradient methods (like PPO2 method in the paper) for faster stepping and faster batch learning but not really usable for DQN. After searching a bit, In issues section, I get my hands on a scalar adapter class which helped me wrap the vector environment and use it as a Scalar environment with minor limitations. In DQN we do one step at a time so Scalar environment was needed. Note: maybe vector version can be used to fill the initial experience replay buffer fast but I did not try this method because it normally takes only 30 to fill up Experience Replay Buffer of size 10,000 to start the training with.

Coinrun is able to create levels with varying complexity. But for our experiments, we will be using fixed levels as showing in Fig. 2 and 3. This should significantly reduce training time. If time permits and everything goes well then we will run the same for random 5 levels or more.

4.1. Basic DQN

Basic DQN algorithm is shown in Figure 4:-
Points should be noted for Basic DQN:-

- Input to the Q-function Neural Network is the observation state of the game in the form of a state image
- Output of Q-function Neural Network is a 1-D array estimating reward earned for each possible actions given the current state.
- There are two Neural Networks Q-function and Target Q-function which is same as Q-function but frozen for 10,000 timestamps

We going to use a simple CCN architecture as shown in Figure Below. Which is very similar to the one used by

⁷<https://ai-alignment.com/the-reward-engineering-problem-30285c779450>



Figure 2. One is very simple level can be recreated using 'num-levels 1 -set-seed 8'



Figure 3. Another simple a little larger Level can be recreated using 'num-levels 1 -set-seed 0'

Algorithm 1 Deep Q-learning with experience replay

```

Initialize replay memory  $D$  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  $\varepsilon$  select a random action  $a_t$ 
    otherwise select  $a_t = \arg \max_a Q(\phi(s_t), a; \theta)$ 
    Execute action  $a_t$  in the 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 experience  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
    Sample random minibatch of experiences  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$ 
    Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
    Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the weights  $\theta$ 
    Every  $C$  steps reset  $\hat{Q} = Q$ 
  end for
end for

```

Figure 4. Basic DQN Algo. with Target network: Q-function that is fixed over many (10, 000) time-steps

Deepminds to play Atari games with 3 Conv. layers and 2 linear layers wrapped by relu activation functions.

```

self.conv1 = nn.Conv2d(c_in, 32, kernel_size=8, stride=4)
self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)

self.fc4 = nn.Linear(w_out * h_out * 64, 512)
self.head = nn.Linear(512, out_number)

```

Flow Diagram of DQN Algo. has been shown in figure 5. Note the learning starts after the minimum experience have been collected in the replay buffer.

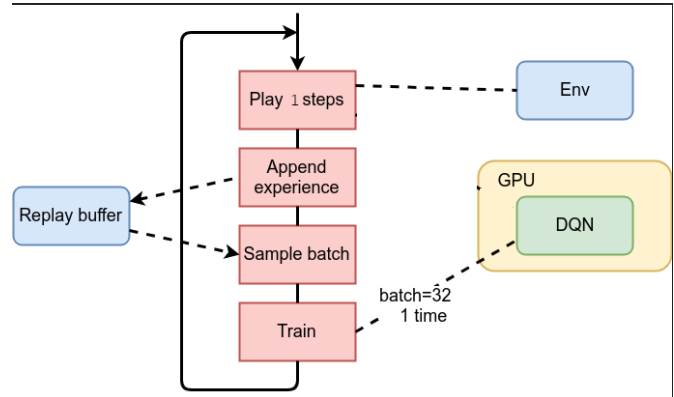


Figure 5. Flow diagram of DQN

Performance of the agent, when trained on simple levels, can be seen in Figure 6. We can see that training further than 300 episodes not really provide any progress. It was observed that early optimal point was never reached again hence early stopping the training can be beneficial. we will also see that the default reward system does not provide any use-full information about the nature of progress made by the system. To resolve this issue we will be using custom reward shaping.



Figure 6. Agent's performance with Simple DQN on 2 simple levels

4.1.1 Reward Shaping

Custom Reward Shaping can help give better results in Spare Reward Settings such as Coinrun environment where we have only one reward i.e. 10 points for collection the coin (Section 2.1.2). Different rules for reward shaping can be used to get better results. Following custom reward shaping have been implemented by me in the code:-

- Agent will be awarded -1 reward for each timestamp where he did not receive any other reward. this will motivate the agent to do an action which yields positive rewards.
- Award for obtaining the coin set to 1000 instead of 10.
- If Agent's action did not change the state of the environment then it is awarded -2. this will motivate the agent to perform impact-full actions.
- All levels start with an agent on the left and coin at the far right side of the level. Hence performing a forward jump can resolve many obstacles and gets the agent near to the reward hence. action 4 (forward jump) will be awarded 1 reward.
- death by any mean will result in -800 rewards.

A graph in Figure 7 show the impact of applying custom reward shaping on DQN performance. We can clearly see reward shaping help in quick conversion of the network to an optimal policy with a positive impact on overall performance.

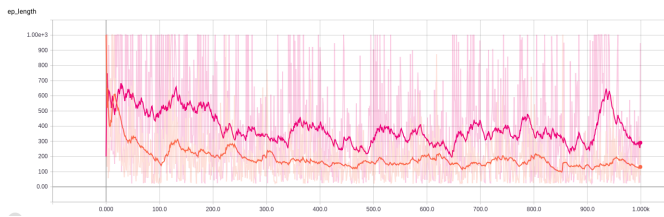


Figure 7. Orange with reward shaping enabled. Note as this experiment involve simple level with no agent's death possibility. the length of the episode can be representative of success of the episode

Challenges faced during this the whole DQN experiment:

- how fast should you decay the random movements. finally fixed it to from 1.0 to 0.1 decay over 25000 steps
- what should be the associated Adam learning rate. finally fixed it to 0.00025 for rest of the experiments.
- how big should be the memory. finally fixed it, start training from 10000 with a maximum of 100000 sizes.

- tested batch size 32 and 64. finally fixed with 64 because the reward is sparse and larger batch size have more change to have at-least data point with reward to help progress the network.
- writing a bug-free code which writes meaningful logs to track progress.

4.2. Double DQN

Double DQN method [16] is very similar to Simple DQN with minor changes. Double DQN algorithm is shown in Figure 8.

Algorithm 1: Double DQN Algorithm.

```

input :  $\mathcal{D}$  - empty replay buffer;  $\theta$  - initial network parameters,  $\theta^-$  - copy of  $\theta$ 
input :  $N_r$  - replay buffer maximum size;  $N_b$  - training batch size;  $N^-$  - target network replacement freq.
for episode  $e \in \{1, 2, \dots, M\}$  do
  Initialize frame sequence  $\mathbf{x} \leftarrow ()$ 
  for  $t \in \{0, 1, \dots\}$  do
    Set state  $s \leftarrow \mathbf{x}$ , sample action  $a \sim \pi_\theta$ 
    Sample next frame  $x'$  from environment  $\mathcal{E}$  given  $(s, a)$  and receive reward  $r$ , and append  $x'$  to  $\mathbf{x}$ 
    if  $|\mathbf{x}| > N_r$  then delete oldest frame  $x_{t_{min}}$  from  $\mathbf{x}$  end
    Set  $s' \leftarrow \mathbf{x}$ , and add transition tuple  $(s, a, r, s')$  to  $\mathcal{D}$ ,
      replacing the oldest tuple if  $|\mathcal{D}| \geq N_r$ 
    Sample a minibatch of  $N_b$  tuples  $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$ 
    Construct target values, one for each of the  $N_b$  tuples:
    Define  $a^{\max}(s'; \theta) = \arg \max_{a'} Q(s', a'; \theta)$ 
     $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-) & \text{otherwise.} \end{cases}$ 
    Do a gradient descent step with loss  $\|y_j - Q(s, a; \theta)\|^2$ 
    Replace target parameters  $\theta^- \leftarrow \theta$  every  $N^-$  steps
  end
end

```

Figure 8. Double DQN algorithm

Notice carefully for one major difference between the DQN and Double DQN is that the action is for target Q values are calculated using Q-function and not from the target Q-function. This is done to fix the overestimating the Q-values. Change in the code for this was small because we already have a target network which was not present in the vanilla DQN. The change is shown in Figure 9

```

q_next = self.target_Q_function(batch_next_state).detach()
if self.use_double_qlearning:
    best_actions = torch.argmax(self.Q_function(batch_next_state), dim=-1)
    next_Q_values = q_next.gather(1, best_actions.unsqueeze(1)).squeeze(1)
else:
    next_Q_values = q_next.max(1)[0]
target_Q_values = batch_reward + self.gamma * next_Q_values * (1 - done_mask)

```

Figure 9. Double DQN algorithm

Results with Double DQN compare to DQN with and without reward shaping are displayed in figure 10. The result was taken on one common medium size level. We can clearly see improvement in all factors like the length of the episodes and total episode reward. Further improvement can be seen using custom reward shaping. I figure that reward per timestamp would be a better representation of the performance of the algorithm. So the first graph shows the avg. reward collected per step.

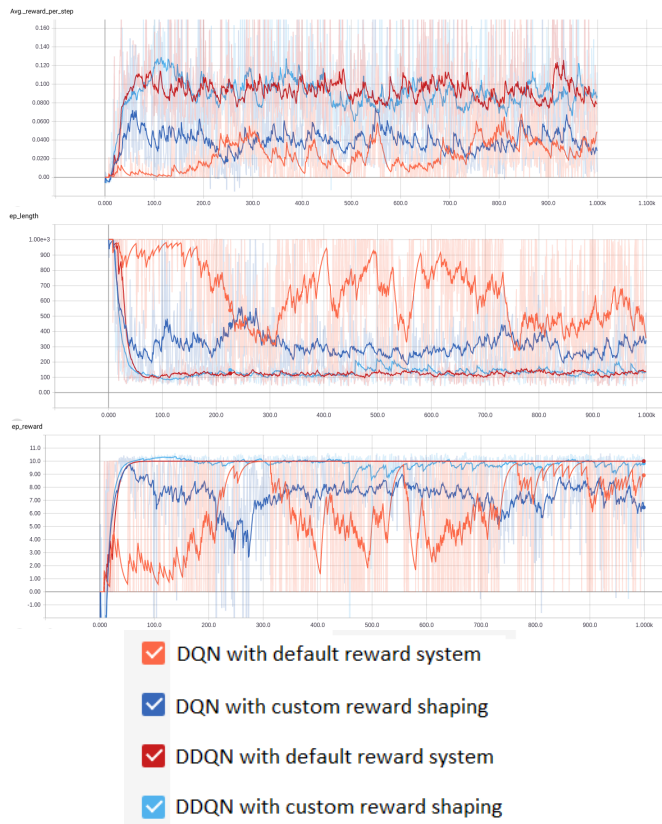


Figure 10. Comparison of Double DQN vs Simple DQN algorithm with and without custom reward shaping

5. Discussion and future directions

I found out the DQN for simple levels is able to converge with very few numbers of episodes 100-200. This proof that DQN is sample efficient. But the main problem I faced is that DQN is not able to stay converged to optimal policy for long and start degrading if trained further. Double DQN helped with the needed stability in the DQN and also provide better results. We have also seen custom reward shaping can speed up the time to get an optimal policy in many cases.

Problem with DQN seems to be its limitation to solve more complex levels. I tried multiple time to train for fixed 5 random levels using DQN and DDQN but none of them are able to get the required performance where we can say it has learned to play decently. Whereas the paper showed PPO able to learn even on the unbounded set of levels.

A further study from my project may include the following:-

1. I would love to know the limit of DQN by experimenting with the number of levels for which it can be trained to get decent results.
2. I would like to implement more variant of DQN and

see the impact comparison of such methods like prioritized DQN, distributed prioritized DQN, episodic memory DQN, asynchronous n-step DQN, and multiple DQN. From all of the above, I really want to try out Prioritized DQN when I have time as my own project which according to people really helps.

3. Impact of different architecture on DQN can be studied through experiments
4. If any of the above combinations of DQN give stable results then impact of generalization techniques (Dropout, L2 regularization, Data augmentation, batch normalization, and Stochasticity) also can be studied.

As for the weakness in the original method used in the paper, I would have liked to see the impact of LSTM on simple Coinrun environment as part of the paper. Final LSTM node is part of purpose IMPALA Network [4] and the reason of not using is not clear other than the logic that it probably impacts the speed negatively. Also, it would be interesting to compare the impact of newly purposed TRPPO algorithm [18] which claim to be an improvement of the PPO [11] technique used in the paper. Also, my experiment with DQN proves while DQN is able to learn to play simple level very quickly it is unable to learn complex levels or even fixed multiple random levels. It is not clear that maybe further training or with better network architecture it might be able to learn which could be considered as future work.

References

- [1] K. Arulkumaran, A. Cully, and J. Togelius. AlphaStar: An Evolutionary Computation Perspective. Feb. 2019.
- [2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, Mar. 2012.
- [3] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman. Quantifying Generalization in Reinforcement Learning. *arXiv:1812.02341 [cs, stat]*, Dec. 2018. arXiv: 1812.02341.
- [4] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.
- [5] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Belle-mare, and J. Pineau. An Introduction to Deep Reinforcement Learning. *Foundations and Trends in Machine Learning*, 11(3-4):219–354, 2018. arXiv: 1811.12560.
- [6] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.
- [7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. Dec. 2013.

- [8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.
- [9] M. A. Riedmiller, R. Hafner, T. Lampe, M. Neunert, J. Degraeve, T. V. de Wiele, V. Mnih, N. Heess, and J. T. Springenberg. Learning by playing - solving sparse reward tasks from scratch. *CoRR*, abs/1802.10567, 2018.
- [10] M. Roderick, J. MacGlashan, and S. Tellex. Implementing the deep q-network. *CoRR*, abs/1711.07478, 2017.
- [11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
- [12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.
- [13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, Oct. 2017.
- [14] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, volume 16. 1988.
- [15] P. Thomas. Bias in Natural Actor-Critic Algorithms. In *International Conference on Machine Learning*, pages 441–448, Jan. 2014.
- [16] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.
- [17] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Kttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekeremo, J. Repp, and R. Tsing. StarCraft II: A New Challenge for Reinforcement Learning. Aug. 2017.
- [18] Y. Wang, H. He, X. Tan, and Y. Gan. Trust region-guided proximal policy optimization. *CoRR*, abs/1901.10314, 2019.
- [19] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.