Combination and Benchmark of RL Models

Dane Wang Cody Houff Etienne Sudre

I. Introduction

Atari games have been a long-standing benchmark in the reinforcement learning (RL) community for the past decade. This benchmark was proposed to test general competency of RL algorithms. In this environment, the frames of the game are inputted into the algorithms as observations. The RL community turned toward Deep Learning and Neural Networks in 2013 leading to multiple breakthroughs in algorithm development. In this project, we wanted to understand what part some of those algorithms played in the development of the current state of the art algorithms. To do so, we benchmarked several advanced and classical algorithms on the same Atari Game, Space Invaders, where a laser cannon tries to defend Earth against invading aliens.

To better understand the properties and performances of different algorithms, we extracted base algorithms from Rainbow (an algorithm that combines six different improved DQN algorithms). We would also like to explore new combinations and create our own RL models, so we implemented three combination algorithms that have not been originally explored. We tested Rainbow, these baselines algorithms, and our customized algorithms on Cartpole which is a simple but classic environment for RL.

II. Related Works

There has been a lot of work done in the area of deep reinforcement learning. From classic DQN to advanced Alpha zero [20], many researchers proposed new ways to improve the performance of the RL algorithms. In this section, we are going to discuss some of them and we will discuss the algorithms we worked with.

A. Deep Q Network (DQN)

In order to use deep learning and neural networks with reinforcement learning, Deep Q Networks were developed based on the Q-learning algorithm allowing the estimation of the Q function. This function estimates the values of state action pairs in a reinforcement learning concept. This work was the first to produce an end-to-end evaluation of a state action pair without hand-crafted features precomputed on the input. This networks apart from taking directly the image of the Atari games it is trying to master also introduce two main ideas. The first idea is the use of a replay buffer, by storing every transitions and training the network on a random selection of all the previous transitions, it is possible to increase the data efficiency of the learning process as well as smooth the distribution of transitions. Indeed after some learning steps, the agent starts to develop a behaviour leading it to certain region of the state space more than others providing an unbalanced distribution of the states over the state space. The second idea introduced by DQN is the use of a target network during the update of the network weights. This target network is the same as the evaluation network but is not updated as frequently. It starts with the same weights, it is not updated for a few training steps and is then brought back to the evaluation network weights. This algorithm achieved state of the art performance on several Atari games and even went beyond the human performance on some games.

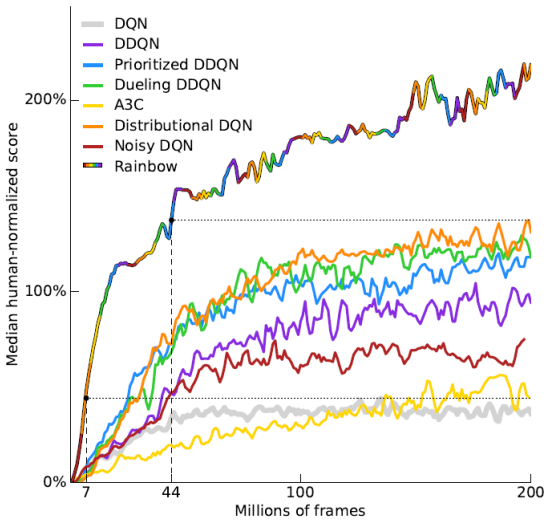

Fig. 1. Rainbow performances

B. Rainbow

Another interesting algorithm we found is Rainbow[9]. It is an algorithm that combines six different improved DQN models and achieve better performance. The six models that are used are Double DQN, Dueling DQN, Prioritized Replay, Multi-step Learning DQN, Noisy Net, and Distributional DQN.

Since these models are built based on the same framework, they really work well together. The evaluation of the rainbow and each individual algorithm are shown below in Figure 1. Rainbow really made a big improvement compared with each individual model as shown in the figure.

A lot of our work is based on the rainbow and the six DQN algorithms it consists of. The explanation of these six improved DQN algorithms and what improvements they made to the original DQN is presented below.

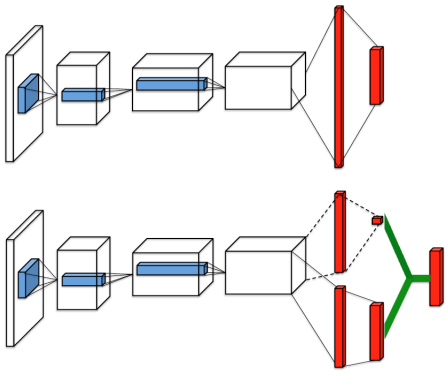

Fig. 2. Network structure of DQN (top) and Dueling DQN (bottom)

C. Double Deep Q-Learning

As discussed in class, one of the main problems of DQN is that the same Q target network is used to both estimate the next Q value and choose the next action. This will result in a higher estimation of the Q value and limit its performance. DDQN [8] decouples the process of value estimation and action selection. The Q target network is still responsible for the Q value estimation, but we will use the Q network to determine the next action that is going to be applied.

D. Dueling Deep Q-Learning

The main difference between Dueling DQN [22] and DQN is the network structure. The structure of the Dueling and DQN network is shown in 2 [22] Instead of getting the Q value directly, Dueling DQN splits it into two state functions V and A. V is used to evaluate the value of the state, and it is only related to the state. A is used to evaluate the value of an action, and it is related to both state and action.

Then our final Q value can be represented as the following formula 1 where θ is the shared parameters of the first few layers, α is the parameters of the fully connected layer for action, and β is the parameters of the fully connected layer for states.

$$Q(s, a, \theta, \alpha, \beta) = V(s, \theta, \beta) + A(s, a, \theta, \alpha) \tag{1}$$

For some specific situations, the Q values of the system only depend on the state, not the action. By splitting the state and action, dueling DQN can give a more accurate estimation for some applications, and increase the stability of the model.

E. Multi-step (N-step) Deep Q-Learning

Traditionally, DQN uses the current reward and the estimated Q value of the next step to evaluate the loss which is represented as equation 2.

$$Loss = (r_j + \gamma \max_a \hat{Q}(s_{j+1}, a, \theta^-) - Q(s_j, a_j, \theta))^2 \tag{2}$$

However, multi-step DQN [1] will use the future N steps to calculate the loss. This could be represented as equation 3 below.

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha[\underbrace{(r_t + \max_a \hat{Q}(s_{t+1}, a; \theta')}_{\text{target}} - Q(s_t, a_t; \theta))]$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{TD-error}}$$

Fig. 3. TD error

$$Loss = \left(\sum_{k=0}^{N-1} \gamma^k r_{t+k} + \gamma^N \max_a \hat{Q}(s_{t+N}, a, \theta^- - Q(s_t, a_t, \theta))\right)^2 \tag{3}$$

In DQN, the deviation of the network parameters in the early stage of training is large, because we basically know nothing about the system. It will result in a large estimation of the target value which affects the performance of the network. However, multi-step learning appears to solve this problem. In multi-step learning, the immediate reward is exactly obtained by interacting with the environment, so the target value in the early stage of training can be estimated more accurately, thereby speeding up the training. [21]

F. Prioritized Replay

Prioritized Replay is built on top of experience replay buffers, which allow a reinforcement learning (RL) agent to store experiences in the form of transition tuples, usually denoted as $(s_t,\ a_t,\ r_t,\ s_t + 1)$. In contrast to consuming samples online and discarding them thereafter, sampling from the stored experiences means they are less heavily "correlated" and can be re-used for learning. [16]

Replay buffers sample the experiences randomly, but Prioritized Replay sample the data based on priority. The priority is determined based on TD error which indicates how far the current prediction function deviates from this condition for the current input. It can also be shown in the figure below.

If there is a large TD error on a prediction, it means that there is still a lot of room for improvement in the prediction accuracy, and the more this sample needs to be learned. In other words, this sample has a higher priority. With Prioritized Replay, the network can learn how to get an accurate prediction with fewer data needed and thereby speeding up the training.

G. Distributional DQN

In traditional DQN, we use a neural network to output an expected Q value to evaluate an action at a specific state. However, we would output the distribution of Q value in Distributional DQN [3]. It will be like if we choose action a, there is a 10% of chance that the reward is 100 and 20% receive an 80, etc. The total possibility will add up to 1.

With the singular expected Q value, there is still a lot of information ignored. For example, if two actions can get the same value expectation of 20. The first action will get 10 for 90% of the time and 110 for the rest 10% of the chance. The second action would receive 25 in 50% of the time and 15 for the other 50% of the time. Although the expectations are the same, in order to reduce the risk, you should choose

the latter action, and only output the expected value without seeing the hidden risk behind it.

Distributional DQN will provide a more thorough evaluation of the action and help the algorithms choose the safer action and improve their stability.

H. Noisy Net

During the training of an agent, one of the biggest challenges is to find a way to balance exploration and exploitation. In traditional DQN, we normally apply the ϵ-greedy strategy, that is, a random strategy is adopted with the probability of ϵ, and the exploratory nature of the system is increased by adopting a larger ϵ in the early stage of training, and the stability of the system is achieved by reducing ϵ in the later stage of training.

Noisy Net DQN provides another approach to this challenge. By adding some random noises (normally Gaussian noise) to the network, it will add some uncertainty to the output. Instead of always outputting the same result with the same input, the network may output a different result because of the influence of the noise. Therefore, the uncertainty caused by the noise could be used to increase the exploratory nature of the system. With a larger noise, the system is more likely to explore different options. We could just use the greedy strategy directly because noises already add randomness to the system, and we could control this randomness by controlling the amount of noise added. The noise will decrease during the training process because of the gradient descent, so the system will be more stable with more training.

In addition, a noisy net will normally result in a smoother learning curve compared with the ϵ-greedy strategy.[6]

I. Advantage Actor-Critic (A2C)

Unlike DQN and DDQN, which are based on Q values estimation, A2C [12] is based on a policy, it is a policy gradient-based method meaning that instead of learning the values for state action pair for a greedy policy, the network learns directly the policy and the output of the network is probabilities of each action being taken for a particular state. A2C is a different version of A3C which uses asynchronous training to do parallel training. In A2C the asynchronous part is removed and all the actors are using the same policy to compute the policy updates based on a critic which learns the value of each state, they are synchronized so that the aggregated update from all agent is not an aggregation of update of different policies. This actor-critic method surpassed the DQN performances on several Atari games.

J. Proximal Policy Optimization (PPO)

One issue present in policy gradient-based method is the size of the steps between a previous policy and its updated version, too big a step can lead to training instability and prevent the policy to converge to a optimal or near-optimal solution. In off-policy, the "policy" we try to learn is not the policy used to gather trajectories this problem is known and can be compensated by importance sampling. However in on-policy reinforcement learning, the trajectories are supposed to be collected by following the policy we try to improve however parallel asynchronous training could be a problem for this assumption. By taking this difference in account and by limiting the divergence between old and current policies, Trust Region Policy Optimization algorithm (TRPO) [19] can achieve monotonic improvement when performing policy iteration. In TRPO the constraint on the divergence of the old and current policy is a hard constraint, similar to a linear programming constraint and not a penalty. The idea behind proximal policy optimization PPO [18] is to generalize the ideas behind TRPO to make them easier to use and to apply them to a wider range of problem. For that instead of using the hard constraint on the divergence between the policies, they clip the ratio so that the update on the policy parameter is still controlled. Alongside this clip they add two penalties to the objective function, one aiming to penalise error of the value estimation and one penalising an insufficient exploration.

III. Methodology

A. Benchmarking

Before choosing which type of algorithm we wanted to test and potentially use for our custom agents later, we wanted to get a sense of the field. The current top state of the art papers benchmarked with Atari games are GDI-H3 [5], MuZero [17], EfficientZero [23], Ape-X [10], Agent57 [2], IMPALA [4], R2D2 [11], Rainbow, DDQN, and DreamerV2 [7]. Algorithms such as A2C, DQN, PPO, and Distributional DQN are some of the baselines researchers commonly use. We attempted to benchmark each of these papers and algorithms with one game to compare and contrast how each compared with the other over time (see Figure 4). The game we chose to use was Space Invaders as it was difficult enough to allow the complex algorithms time to form but not too complex as to require more than a few million steps to get good results.

We were especially interested in finding the best performing algorithm that can be trained on a personal computer and get good results with limited training. We found it took about 10 hours for 5 million steps so our limit for good performance evaluation was a maximum of 5 million steps for an algorithm. Some of these algorithms require a massive amount of computing to run and we had to keep in mind our hardware limitations. Additionally, some of these algorithms had very little documentation. The algorithms we were able to plot for benchmarking were A2C, DQN, PPO, Distributional DQN, IMPALA, Rainbow, and Ape-X.

B. Custom Algorithm

Next, we created our own customized DQN agents. As shown in the next section, Rainbow has the best performance among all algorithms we tested, so we choose to use it a main source. In the paper about Rainbow, the author showed the performance of the Rainbow and the performance of each based model. However, they failed to show the effects of combining only two or three algorithms out of those six. Will

a leaner combination of the top performing algorithms work better or at least be comparable to Rainbow? What is the difference in performance between two algorithms combined (custom agents) and six algorithms combined (Rainbow)? Therefore, for our custom agents, we would want to choose three algorithms among those six DQN models mentioned above. We will try different combinations and then compare with Rainbow and base models.

After careful consideration, we chose to combine and test Double DQN, Multi-Step DQN, and dueling DQN. Double DQN has been widely applied and showed great results. Multi-Step DQN should have a great performance in an early state. [21] Because of the time and GPU limitation, we will focus on the early state or a relatively simple training environment. We chose Dueling DQN because it allows the network to better differentiate actions from one another. We will choose two out of these three algorithms to combine for our custom agents.

When we implemented our custom agents, we combined the certain parts of each model. For DDQN, we used the part that chooses an action with one Q network and estimates the state with the other Q network. For Multi-Step DQN, we extracted its loss calculation method. Instead of using one step, it calculates the loss with a few steps ahead. For Dueling DQN, we applied its network structure where it splits Q into two state functions. We implemented these agents based on this open-source Github project. [13]

Our original plan was to apply our custom agent to the space invaders environment. However, after a few tests, we realized it takes too long to design and test out a new custom agent on an environment that requires hours to get the results . Therefore, we decide to switch to a relatively simple environment - Cartpole. We ran each algorithm with a fixed amount of frames and compared their performances. The results are shown in the next section.

IV. Results

A. Benchmarking

Figure 4 shows the performances of algorithms trained on Space Invaders within a short timeframe. The x-axis is the number of steps, and the y-axis is the corresponding rewards gathered.

B. Custom Algorithm

Figure 5 shows the performances of Rainbow and other different DQN algorithms trained on cartpole. Figure 6 shows the performance of rainbow and our customized agents. Then, we also compare each customized agent with the algorithms it consists of to better show the difference. These results are shown as Figure 7, 8 and 9. More analysis of these figures will be included in the discussion section.

V. Discussion

A. Benchmarking

As shown in Figure 4, Rainbow has the best performance among all the algorithms we tested. Surprisingly, the other two advanced algorithms, IMPALA and Ape-X, do not outperform

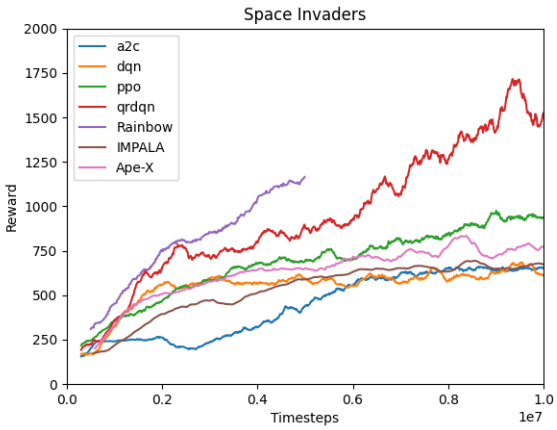

Fig. 4. Benchmark of algorithms on the Space Invaders environment. This figure was generated with code from [14] and [15].

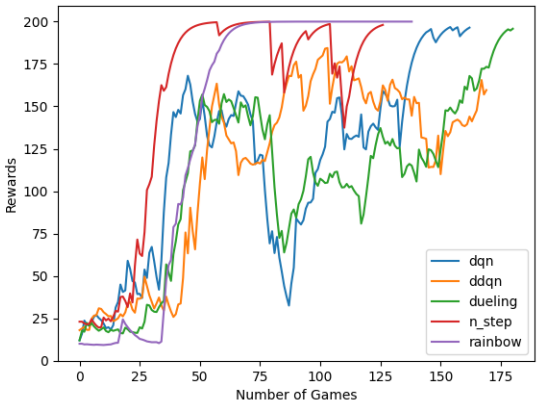

Fig. 5. Rainbow and other DQN algorithms performances

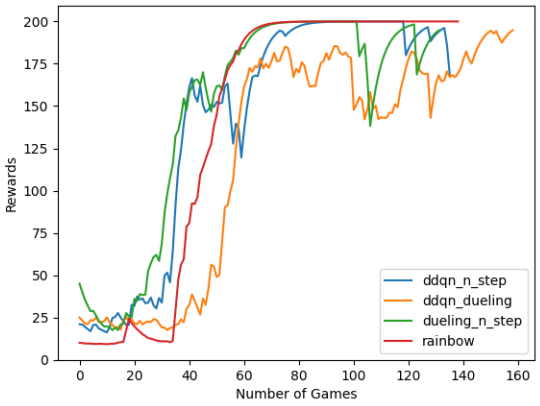

Fig. 6. Rainbow and customized algorithms performances

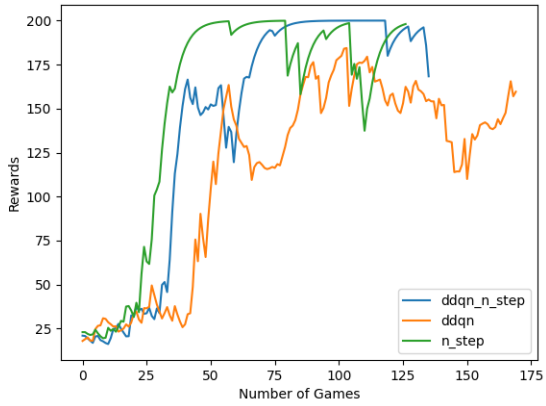

Fig. 7. N-step DDQN and its component algorithms performances

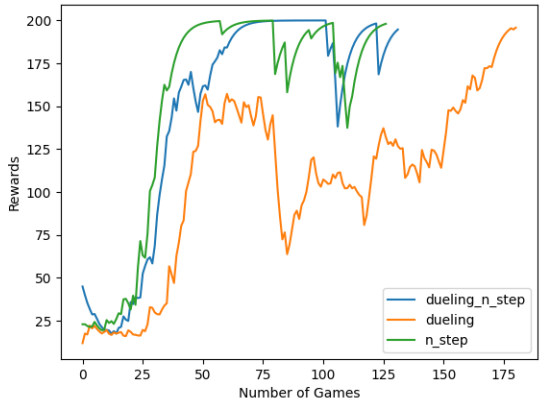

Fig. 8. N-step Dueling and its component algorithms performances

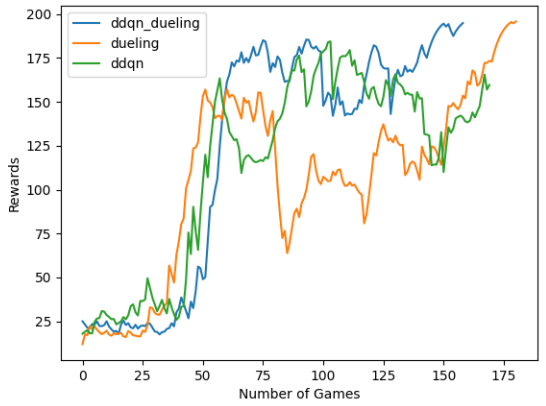

Fig. 9. Dueling DDQN and its component algorithms performances

other base algorithms as we expected. The potential reason is that some algorithms take a long training time to improve the performance, and they may not perform well at the early training state. Then other algorithms have a similar result.

B. Custom Algorithm

From Figure 6, we can see the strengths of the Rainbow algorithm. Even though it doesn't have the fastest rise, it is the most stable and smooth algorithm. It reaches and keeps the maximum scores without any variation. Rainbow definitely has the best overall performance. The next algorithm that impresses us is N-step DQN. It rises quickly and is able to reach the maximum score within the given frame limitation. Even though it varies a little bit after it converges, it still has the best performance among all base DQN agents. For the other three agents, they don't really achieve a good result. They can not converge and varies a lot during the training process. It shows that these three algorithms are not as stable as Rainbow and N-step DQN at the early stage of the training.

Then we compare the customized agents with Rainbow as Figure 7. It shows that agents consisting of N-step DQN both have a quick rise but still have some variations after the convergence. It means that none of the three DQN algorithms we selected is the reason for Rainbow's smooth learning curve. We believe it is the effect of noisy nets because all other algorithms focus on improving the training process by decreasing data needed or more accurate Q estimation. Instead of using ϵ-greedy, noisy nets adds some random noise or error to the network to balance exploration and exploitation. So noisy nets will affect the performance even after the convergence. Its paper states that it will result in a smoother training process, and our results prove this statement.

The last thing we did is compare customized agents with the corresponding component algorithms. Surprisingly, only the DDQN_Dueling algorithm outperforms its original algorithm and gets a more stable result. Unfortunately, it still didn't get to the maximum rewards (200 points). The other two agents didn't work as well as the original N-step DQN. They take a longer time to converge.

In summary, there are two main things we learned from this project. First of all, N-step DQN works really well for the early stage or a relatively simple environment. It has a very quick rise and convergence speed. Secondly, noisy nets will help smooth the learning curve, especially after the convergence.

However, there are still some limitations to our project. The main thing is our environment choice. Because of the time and resources limit, we can't apply some hard and complicated environments. It may limit the performance of some algorithms like DDQN or Dueling. Then we only tried a few combinations among the six DQN algorithms used. There is still more work to do to fully understand the affection of each model in Rainbow.

VI. Conclusion

In this project, we studied some advanced Deep Reinforcement Learning algorithms and compared their per-

formances with some base algorithms we learned in class. We specifically dug deep into the Rainbow algorithm which consists of 6 improved DQN models because of its outstanding performance.

Some combinations of Rainbow's component algorithms are implemented and evaluated in the cartpole environment. Unfortunately, we could only use this relatively simple environment because of the time and resource limit. Some algorithms like DDQN may not show their full potential because of the environment selection. That being said, we still generated some interesting conclusions from the combinations chosen. We have proved that multi-step DQN works really well in a simple environment, and noisy nets can help smooth the overall learning curve with its action selection strategy.

More work could be done in the future by trying different test environments and combinations of algorithms to fully understand the effect of each component algorithm in Rainbow.

References

[1] Kristopher De Asis et al. "Multi-step Reinforcement Learning: A Unifying Algorithm". In: *CoRR* abs/1703.01327 (2017). arXiv: 1703.01327. URL: http://arxiv.org/abs/1703.01327.

[2] Adrià Puigdomènech Badia et al. "Agent57: Outperforming the Atari Human Benchmark". In: *CoRR* abs/2003.13350 (2020). arXiv: 2003.13350. URL: https://arxiv.org/abs/2003.13350.

[3] Marc G. Bellemare, Will Dabney, and Rémi Munos. "A Distributional Perspective on Reinforcement Learning". In: *CoRR* abs/1707.06887 (2017). arXiv: 1707.06887. URL: http://arxiv.org/abs/1707.06887.

[4] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *CoRR* abs/1802.01561 (2018). arXiv: 1802.01561. URL: http://arxiv.org/abs/1802.01561.

[5] Jiajun Fan, Changnan Xiao, and Yue Huang. "GDI: Rethinking What Makes Reinforcement Learning Different From Supervised Learning". In: *CoRR* abs/2106.06232 (2021). arXiv: 2106.06232. URL: https://arxiv.org/abs/2106.06232.

[6] Meire Fortunato et al. "Noisy Networks for Exploration". In: *CoRR* abs/1706.10295 (2017). arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295.

[7] Danijar Hafner et al. "Mastering Atari with Discrete World Models". In: *CoRR* abs/2010.02193 (2020). arXiv: 2010.02193. URL: https://arxiv.org/abs/2010.02193.

[8] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[9] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *CoRR* abs/1710.02298 (2017). arXiv: 1710.02298. URL: http://arxiv.org/abs/1710.02298.

[10] Dan Horgan et al. "Distributed Prioritized Experience Replay". In: *CoRR* abs/1803.00933 (2018). arXiv: 1803.00933. URL: http://arxiv.org/abs/1803.00933.

[11] Steven Kapturowski et al. "Recurrent Experience Replay in Distributed Reinforcement Learning". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=r1lyTjAqYX.

[12] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[13] Jinwoo Park. *Rainbow is all you need*. https://github.com/Curt-Park/rainbow-is-all-you-need. 2019.

[14] Antonin Raffin. *RL Baselines3 Zoo*. https://github.com/DLR-RM/rl-baselines3-zoo. 2020.

[15] ray_project. *RL Experiments*. https://github.com/ray-project/rl-experiments. 2020.

[16] Tom Schaul et al. "Prioritized Experience Replay". In: abs/1511.05952 (2015). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[17] Julian Schrittwieser et al. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model". In: *CoRR* abs/1911.08265 (2019). arXiv: 1911.08265. URL: http://arxiv.org/abs/1911.08265.

[18] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[19] John Schulman et al. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[20] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[21] Richard S. Sutton. "Learning to predict by the methods of temporal differences". In: *Machine Learning* 3 (1988), pp. 9–44. DOI: https://doi.org/10.1007/BF00115009. URL: link.springer.com/article/10.1007/BF00115009.

[22] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.

[23] Weirui Ye et al. "Mastering Atari Games with Limited Data". In: *CoRR* abs/2111.00210 (2021). arXiv: 2111.00210. URL: https://arxiv.org/abs/2111.00210.