THERE'S A PONG OF GOLD AT THE RAINBOW'S END: PLAYING WIMBLEPONG WITH RAINBOW

A PREPRINT

Abhilash Jain, 722074
Department of Computer Science
Aalto University
Espoo, Finland
abhilash.jain@aalto.fi

Dimitrios Papatheodorou, 721033
Department of Computer Science
Aalto University
Espoo, Finland
dimitrios.papatheodorou@aalto.fi

December 8, 2019

ABSTRACT

We employed the state-of-the-art DQN method Rainbow to learn how to play Wimblepong and put it against Simple AI, an agent that follows the ball closely, trying to make our agent generalize well so that it wins against others as well. We discuss about other methods, explain Rainbow in detail and talk about our extensive preprocessing and workflow. Our results show that using transfer learning by using a trained agent on the whole (preprocessed) image as a basis for continuing the training with an image not having the opponent paddle (cropped opponent) is the key in not only overcoming overfitting but making our agent better in every case.

1 Introduction

Wimblepong is a game based on ping pong which shares similarity to the well-known atari pong, where two players use paddles to hit a ball against each other, and the goal is that your opponent misses the ball. Reinforcement learning is a particularly good application for these kind of PvP games and the setup in this project is that we train an RL agent who plays against a SimpleAI agent that is actually and if-else statement following the movement of the ball. The RL environment is built such that the reward of winning is +10 and the reward of losing is -10. Some of the usual approaches to the RL agent deploy techniques like A3C, DQN and IQN [1, 2, 3] but in this project we would like to explore beyond this.

The RL agent we used is called *Rainbow* [4] by Hessel et al., which combines a lot of recent improvements in Deep Reinforcement Learning, such as Noisy Linear layers, Double DQN, Prioritized Memory Replay and more, to achieve high performance. Combining this with extensive preprocessing and transfer learning made the agent robust against any opponent.

2 Related work

In recent years, Reinforcement learning has seen a lot of developments setting new benchmarks each year. In this project we dwell into once such latest techniques (Rainbow) but before that understanding of the basic techniques is still paramount. We discuss here two such techniques we considered before ending choosing Rainbow, namely A3C and DQN.

2.1 A3C

Asynchronous Advantage Actor-Critic (A3C) algorithm was released by Google's DeepMind in June of 2016 and at that time it took the RL world by storm by setting new benchmarks. From 1 [5] lets try to understand the different parts of the algorithm:

- **Asynchronous**: In A3C there is a global network, and multiple worker agents which each have their own set of network parameters. Each of these agents interacts with it's own copy of the environment at the same time as the other agents are interacting with their environments. The reason this works better than having a single agent (beyond the speedup of getting more work done), is that the experience of each agent is independent of the experience of the others. In this way the overall experience available for training becomes more diverse.
- **Actor-Critic**: Actor-Critic combines the benefits of value-iteration methods(Q-learning) and policy-iteration(Policy Gradient) methods. In the case of A3C, our network will estimate both a value function V(s) (how good a certain state is to be in) and a policy π(s) (a set of action probability outputs). These will each be separate fully-connected layers sitting at the top of the network. Critically, the agent uses the value estimate (the critic) to update the policy (the actor) more intelligently than traditional policy gradient methods.
- **Advantage**: We use advantage estimate to allow the agent to determine not just how good its actions were, but how much better they turned out to be than expected. Intuitively, this allows the algorithm to focus on where the network's predictions were lacking. Formulated as, $A = R - V(s)$

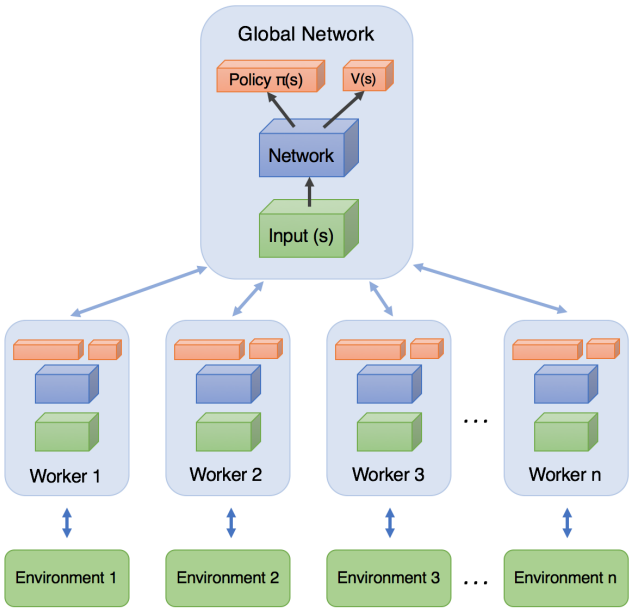

Figure 1: A3C high-level architecture

2.2 DQN

DQNs are the basis of a lot of modern state-of-the-art models in reinforcement learning by combining the idea of using deep neural networks as approximators utilizing their high capacity and expressive power. We discuss in depth about DQN [Figure 2], later on when explaining Rainbow, but it is important to know that DQN's came into spotlight, after this Stanford article [6] showcased its potential, by beating human-level performances on various types of games, including Pong.

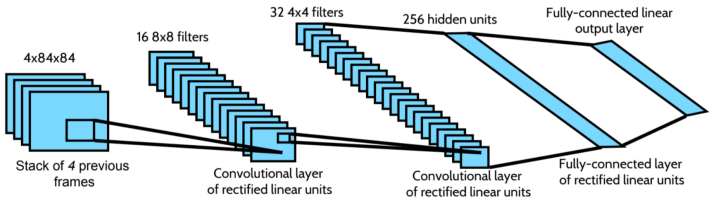

Figure 2: A DQN architecture

3 Approach and method

In our approach we use preprocessing step inspired mostly by Mnih et al. in "Human-level Control through Deep Reinforcement Learning" [7] and the Rainbow agent with all its components enabled. Rainbow is very sophisticated and shows state-of-the-art results so it was the way to go. Also since it has a lot of components, we could learn about a lot of things at once, which helped us a lot in seeing what's happening in the field. At first we wanted to also try A3C and compare the methods, but we couldn't find the time for that as well.

3.1 Preprocessing

The observation output of the environment is a $200 \times 200 \times 3$ image of the game frame at the current state. We used the stochastic frame skipping of the environment, skipping either 2, 3 or 4 frames uniformly at random, to avoid excess information. Each frame is transformed into grayscale, as the color is not needed, and downsampled by a factor of 75%, meaning that the frame becomes 50×50, as there's no need to keep all the pixels - we only need the essential information. This downsampling is a bit aggressive but seems to work by making the model data efficient. Furthermore, we threshold the downsampled image and normalize it so that the background is 0-valued and the paddles and ball are 1-valued. We provide the option of cropping out the opponent's side (12 rightmost pixel columns). On one hand, this may help the agent to focus only on it's paddle and the ball movement and not get distracted by the opponent (noisy information) or learn to mirror them. It's a good idea in terms of defence. On the other hand, knowing the opponent movements may help the agent find ways to win, which is a better offensive tactic. Our guess is that cropping the opponent also helps in reducing overfitting, so that our agent fairs better against agents it didn't train against. This process is repeated for every observation output o the environment.

Another part of the preprocessing is the temporal frame stacking. We keep a queue of window size 4 of incoming frames which are stacked into a $4 \times 50 \times 50$ tensor (stacked frames act like channels) and this our final preprocessed observation at a current step, ready to be used by our model. For example, if at one point we have an observation x_10, the stacked tensor is (x_7, x_8, x_9, x_{10}), and at the next point where we have the observation x_11, the stacked tensor is $(x_8, x_9, x_{10}, x_{11})$, thus the frames have a 3-frame overlap between them. This frame overlapping frame stacking helps the model understand the temporal aspect of the ball movement. An example for skipping 3 frames at a time can be seen in Figure 3

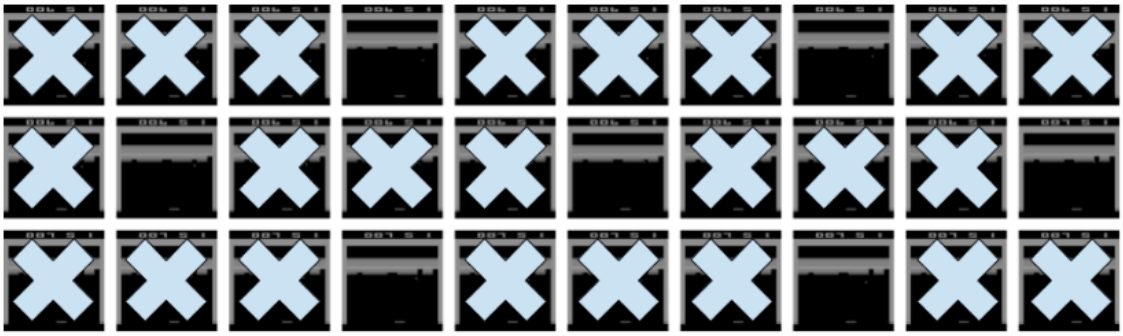

Figure 3: Example of skipped frames

In addition, we also provide the option to a posteriori increase the reward by a small factor if the ball is successfully reflected by the agent, and another option to clip the rewards.

Helpful blogpost: Preprocessing

3.2 Rainbow

Rainbow is an agent that extends the Mnih el al.'s vanilla DQN method [8] by combining a lot of different methods designed to tackle different problems in the area, thus the name, to achieve high data efficient and final performance. The components used are Double DQN[9], Dueling Networks[10], Prioritized Experience Replay[11], Multi-step learning [12, 13, 14], Distributional RL[15] and Noisy Nets[16].

To establish the notation at timestep t: $S_t \in \mathcal{S}$ is the observation spat out by the environment and preprocessed, $A_t \in \mathcal{A}$ is action made by the agent, R_{t+1} is the returned reward, $\gamma_{t+1} \in [0, 1]$ the discount factor of the reward,

$G_t = \sum_{k=0}^{\infty} \gamma_t^{(k)} R_{t+k+1}$ the discounted return where $\gamma_t^{(k)} = \prod_{i=1}^{k} \gamma_{t+i}$. The policy π defines a probability distribution over the action space for each state, and the agent wishes to find a good policy so that it can maximize the expected discounted return (through estimates). We also define the state-action function as $q^{\pi}(s,a) = E_{\pi}\left[G_t \mid S_t = s, A_t = a\right]$.

3.2.1 DQN

The basis of Rainbow is the DQN, which approximates policies $\pi(s,a)$ and values $q(s,a)$ using (deep) neural networks, either through the same network with different output layers or through different networks altogether, given the observations. The choice of actions at each steps are performed $\epsilon - greedily$ to promote exploration, and the transitions $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ are stored in the replay memory. The network is trained by some sophisticated gradient descent method, such as Adam, Radam, RMSprop, etc., on minibatches uniformly sampled from the replay buffer, to find the best parameters θ (the parameters of the main *online network*, the one used to make actions), minimizing the loss $\left(R_{t+1} + \gamma_{t+1} \max_{q'} q_{\bar{\theta}}\left(S_{t+1}, a'\right) - q_{\theta}\left(S_t, A_t\right)\right)^2$. $\hat{\theta}$ are the parameters of the *target network*, a periodic copy of the online network that is not directly optimized. For image input the layers used are convolutional.

The target network is necessary for stable optimization as it essentially prevents the network of "chasing its own tail" (something happening with DQN and not Q-learning, as it's updating multiple observations), by letting the agent acquire some experience before updating instead of updating after every move, making action predictions more robust.

Helpful blogpost: Vanilla DQN

3.2.2 Double DQN

In the loss function shown previously the max operation overestimates the values in a systematic way introducing maximization bias. This can become problematic considering that Q-learning involves bootstrapping (learning estimates from estimates). We can have unbiased Q-value estimates of the actions selected using the online estimator. We can thus avoid maximization bias by disentangling our updates from biased estimates. The loss becomes:

$$\left(R_{t+1} + \gamma_{t+1} q_{\bar{\theta}}\left(S_{t+1}, \text{argmax}_{a'} q_{\theta}\left(S_{t+1}, a'\right)\right) - q_{\theta}\left(S_t, A_t\right)\right)^2$$

.

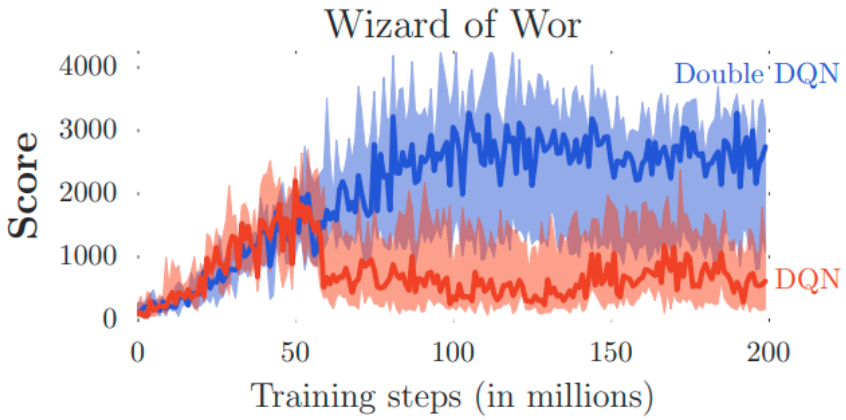

Figure 4: Double DQN overestimation fix

Helpful blogpost: Double DQN

3.2.3 Prioritized Experience Replay

Sampling uniformly from our memory is naive, as we want the probability mass to be centered more on transitions that are more useful in learning, so that they're sampled more frequently. Schaul et al. [11] propose such a method that approximates this "ideal" distribution by making the assumption/heuristic that newer transitions are more important and transitions' probabilities are relative to the last encountered *TD error*:

$$p_t \propto \left|R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta},}\left(S_{t+1} a'\right) - q_{\theta}\left(S_t, A_t\right)\right|^{\omega},$$

where ω is a hyper-parameter that determines the shape of the distribution. The updates are happening only when the transitions are encountered again. The newly inserted transitions have maximum priority to introduce this bias towards the most recent ones. In practice, this is done using *weighted importance sampling*:

$$w_j = (N \cdot P(j))^{-\beta} / \max_i w_i, \text{ where } P(j) = p_j^\alpha / \sum_i p_i^\alpha$$

This way, the probability of sampling transition j is controlled by the parameter α controlling prioritization-vs-uniformity that can help avoid overfitting. According to Schaul et al., estimation of the expected value with these updates relies on those updates corresponding to the same distribution as its expectation, and prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to. Weighted IS solves this and improves further by annealing the β parameter.

Helpful blogpost: Prioritized Experience Replay

3.2.4 Dueling Networks

The advantage value is the difference between the $Q(S_t, A_t)$ value and the state value $V(S_t)$. For an action it's measuring how much worse an action is in comparison to the best action in a state.

With Dueling DQN, you have two separate estimation streams after the convolutional encoder, one for the state value and one for the advantages. To calculate the Q values, the advantage is summed with the state value and the average value of the advantages is subtracted. The factorization of the action values is as follows:

$$q_\theta(s,a) = v_\eta\left(f_\xi(s)\right) + a_\psi\left(f_\xi(s), a\right) - \frac{\sum_{a'} a_\psi\left(f_\xi(s), a'\right)}{N_{\text{actions}}},$$

where $\theta = \{\xi, \eta, \psi\}$ are, respectively, the parameters of the shared encoder f_ξ, of the value stream u_η, and of the advantage stream a_ψ. The final term is basically the average of the advantages.

Helpful blogpost: Dueling Networks

3.2.5 Multi-step Learning

Q-learning accumulates a single reward and then uses the greedy action at the next step to bootstrap, but alternatively, forward-view multi-step targets can be used for faster learning. We define the truncated n-step return from a given state S_t as:

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

Then the loss of the multi-step DQN becomes:

$$\left(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}\left(S_{t+n}, a'\right) - q_\theta\left(S_t, A_t\right)\right)^2$$

Extra hints about multi-step learning in Rainbow: Multistep learning extras

3.2.6 Distributional RL

In this RL variant, instead of approximating expected returns, we approximate a returns distribution over a support z. Specifically, Bellemare et al.'s [15] proposal is a discrete support vector z of $N_{atoms} \in \mathbb{N}^+$ atoms, defined by $z^i = v_{\min} + (i-1)\frac{v_{\max}^{\text{atan}} - v_{\min}}{N_{\text{atoms}} - 1}$, $i \in \{1, \ldots, N_{\text{atoms}}\}$. The approximating distribution d_t at time t is defined on this support, with the probability mass $p_\theta^i(S_t, A_t)$ on each atom i, such that $d_t = (z, p_\theta(S_t, A_t))$. The goal is to update θ such that this distribution closely matches the actual distribution of returns.

To learn the probability masses, the key insight is that return distributions satisfy a variant of Bellman's equation. For a given state S_t and action A_t, the distribution of the returns under the optimal policy π^* should match a target distribution defined by taking the distribution for the next state S_{t+1} and action $a_{t+1}^* = \pi^*(S_{t+1})$, contracting it towards zero according to the discount, and shifting it by the reward (or distribution of rewards, in the stochastic case). A distributional variant of Q-learning is then derived by first constructing a new support for the target distribution, and then minimizing the Kullback-Leibler divergence between the distribution d_t and the target distribution $d_t' \equiv \left(R_{t+1} + \gamma_{t+1}\boldsymbol{z}, \boldsymbol{p}_{\bar{\theta}}\left(S_{t+1}, \bar{a}_{t+1}^*\right)\right)$:

$$D_{\text{KL}}\left(\Phi_z d_t' \| d_t\right),$$

where Φ_z is the L2-projection onto the z support, and $\bar{a}^*_{t+1} = \text{argmax}_a \, q_{\bar{\theta}}\left(S_{t+1}, a\right)$ is the greedy action with respect to the mean action values $q_{\bar{\theta}}\left(S_{t+1}, a\right) = z^\top \boldsymbol{p}_\theta\left(S_{t+1}, a\right)$ in S_{t+1}.

The target network update can be done without any changes and the parameterized distribution can be easily represented by a neural network with output size $N_{atoms} \times N_{actions}$ and the softmax activation across the actions dimension to ensure the output is a distribution (summation to 1).

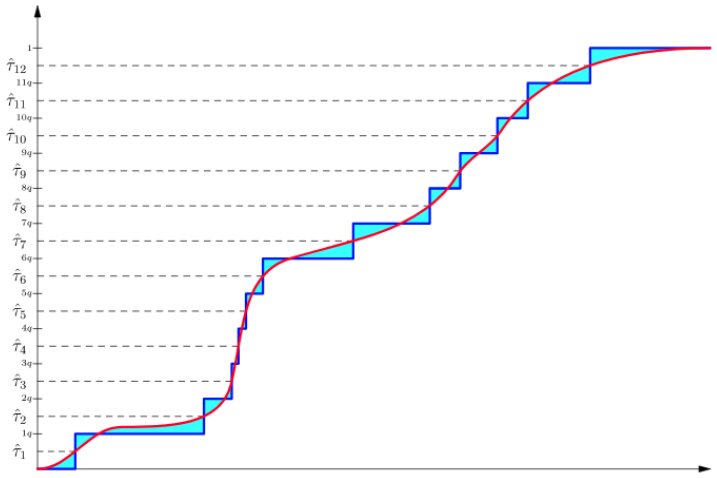

Figure 5: Approximation of the CDF of a distribution using 11 atoms

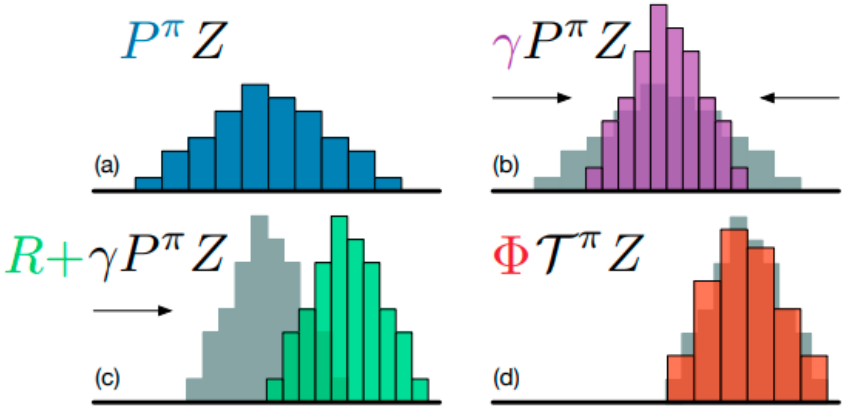

Figure 6: A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π, (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step

Helpful blogpost: Distributional RL

3.2.7 Noisy Nets

Instead of promoting exploration through predefined ϵ-greedy strategies, Fortunato et al. [16] propose a linear layer tha combines the deterministic and noisy stream as:

$$\boldsymbol{y} = \left(\boldsymbol{b} + \mathbf{W}\boldsymbol{x}\right) + \left(\boldsymbol{b}_{noisy} \odot \boldsymbol{\epsilon}^b + \left(\mathbf{W}_{noisy} \odot \epsilon^w\right)\boldsymbol{x}\right)$$

where ϵ^b and ϵ^w are random variables. Since it's a learnable layer, it can learn to ignore or not the noisy stream at different rates in different parts of the state space, allowing state-conditional exploration with a form of self-adaptation.

3.2.8 Combined agent: Rainbow

For rainbow all the above are combined. Specifically, the distributional loss is expressed using multi-step learning: $D_{\mathrm{KL}}\left(\Phi_z d_t^{(n)} \| d_t\right)$ where the target distribution is $d_t^{(n)} = \left(R_t^{(n)} + \gamma_t^{(n)} z, \quad p_{\bar{\theta}}\left(S_{t+n}, a_{t+n}^*\right)\right)$. Then the KL divergence used to prioritize transitions becomes $p_t \propto \left(D_{\mathrm{KL}}\left(\Phi_z d_t^{(n)} \| d_t\right)\right)^{\omega}$. This distributional loss is combined with the aforementioned Double Q-learning technique, by using the greedy action in S_{t+n} selected according to the online network as the bootstrap action a^{t+n}, and evaluating such action using the target network. Finally, using the dueling networks idea, we get a shared representation $f_\xi(s)$, fed into the value computation stream u_η with N_{atoms} output size, and into the advantage stream α_ξ with $N_{atoms} \times N_{actions}$ output size, then aggregated by adding them and subtracting the mean of the advantages across the actions, and then passed through a softmax layer, like this:

$$p_\theta^i(s,a) = \frac{\exp\left(v_\eta^i(\phi) + a_\psi^i(\phi,a) - \bar{a}_\psi^i(s)\right)}{\sum_j \exp\left(v_\eta^j(\phi) + a_\psi^j(\phi,a) - \bar{a}_\psi^j(s)\right)},$$

where $\phi = f_\xi(s)$, $a_\xi^i(\phi,a)$ output for atom i and action a, and $\bar{a}_\psi^i(s) = \frac{1}{N_{actions}} \sum_{a'} a_\psi^i(\phi,a')$.

Helpful blogpost: Rainbow

3.3 Experiments

Now we will explain the technical details of our experiments, our choices and our hyper-parameters, as well as our results.

3.3.1 Methodology and workflow

The details can be seen in the `main.py` file. A terminal interface of a lot of parameters is provided to either choose hyper-parameters or choice among training, evaluation, continuing training and loading models and saved memory. A replay buffer for the validation memory is created and filled with some transitions and then it chooses between evaluation or training. The first actions performed and the first transitions saved are not used for training, but act as a warm-up. After starting learning, the importance sampling weight β is annealed, the agent is trained every n steps (multi-step), and every once in a while there is an evaluation step when the agent is tested for 10 episodes and the results are plotted and the model is saved if it's better. Every some thousand frames, the target network is updated and a checkpoint is saved.

Our approaches consisted of training the agent without additional rewards or cropped opponent, adding a small reward if the ball gets successfully reflected, cropping the opponent throughout the whole training and cropping the opponent after training till convergence without cropping, essentially applying transfer learning. The latter was much more effective in reducing overfitting and generally achieving high performance against any agent we tested it against.

3.3.2 Hyper-parameters and design choices

We used more or less the same hyper-parameters used in the cited papers. This method has a lot of moving parts with a lot of parameters which makes hyper-parameter tuning very difficult. Below you can see a list of all the parameters and design choices of the model. A point that needs attention is the distributional min/max support values which should better encapsulate the exact range of the rewards so that there's no over/under-estimation. If, for example, one wishes to add extra reward in some cases, a good idea to clip the maximum/minimum reward to some range and use this range for support as well. The choice of the neural network architecture could also be extended to more layers and neurons, especially for the convolutional encoder, to increase the model capacity, but we wished for less to combat overfitting and have faster training. So it was more of a play-safe conservative choice, but was enough.

Hyper-parameters:

- Maximum training steps (not-skipped frames): 50M
- History length / temporal channels / deque window: 4
- Replay Buffer size: 1M
- Discount factor γ: 0.99
- Min history to start learning: 80K frames

- Target Network update interval: 8K
- Reward clipping: $None$
- Optimizer: Adam
- Adam learning rate: 0.0000625
- Mini-batch size: 32
- Noisy Nets σ_0: 0.5
- Target Network Period: 32K frames
- Adam ϵ: $1.5 \times 10 - 4$
- Prioritization exponent ω: 0.5
- Prioritization importance sampling β: annealing $0.4 \rightarrow 1.0$
- Multi-step returns n: 3
- Distributional atoms: 51 (C51 model)
- Distributional min/max values: $[-10, +10] \propto$ reward values
- Evaluation interval: 1K
- Evaluation episodes: 10
- Neural Network:
  - Convolutional Encoder:
    - CONV2D [in:4, out:32, kernel:5×5, stride:5], ReLU
    - CONV2D [in:32, out:64, kernel:5×5, stride:5], ReLU
    - flattened output: 256
  - Fully Connected Noisy Linear layers:
    - Value Stream: $256 \xrightarrow{\text{ReLU}} 512 \rightarrow 51 : v$
    - Advantage Stream: $256 \xrightarrow{\text{ReLU}} 512 \rightarrow 3 \times 51 : a$
    - $q = v + a - \text{mean}(a, \text{wrt:actions})$ (flattened a, v)
    - $q \xrightarrow{\text{Softmax}} output$

3.4 Results and performance analysis

In this section we discuss our experiments' results and the model general performance. As said before, the evaluation of the model and the resulting plots happen every 10,000 iterations. Thus, the plots below are an output of evaluation and not training output. The evaluation consists of 10 episodes so there's stochasticity/noise in the plots. Unfortunately, a higher number of episodes would make the process very slow. The general trend can be easily seen though. The plots are the mean reward and the Q values.

3.4.1 Simple approach

This is the first approach we tried and consists of the aforementioned workflow without adding any rewards or cropping the opponent. In Figure 7 and Figure 8 you can the Q values and rewards plot respectively for this approach. Generally speaking, the Q values take a deep in the first 100,000 iterations and then start rising, and both Q values and rewards converge after 4-6M iterations. This method, although performing very well against Simple AI with a winrate of 90%, to which it trained against, it performs poorly against anything else, barely managing to reach a 50% winrate against simple agents. The reason is the high overfitting in battling and winning Simple AI. A rendered play can easily give us hints about it as it may mirror the opponent, make actions depending on the opponent's movements instead of the balls and making offensive actions that only Simple AI would lose to.

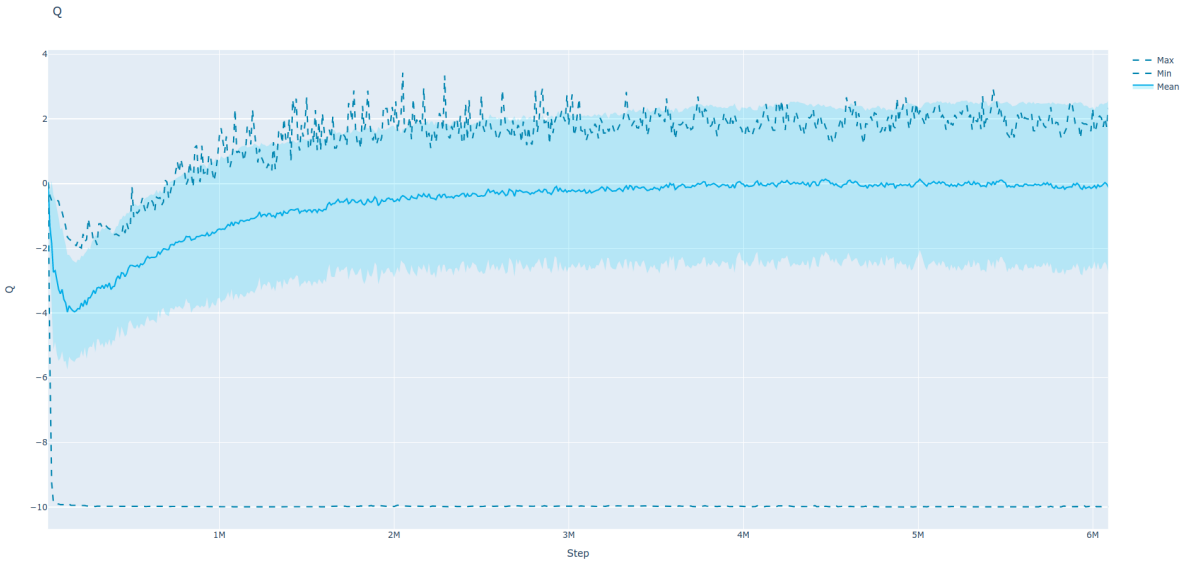

Figure 7: Q values for the Simple approach

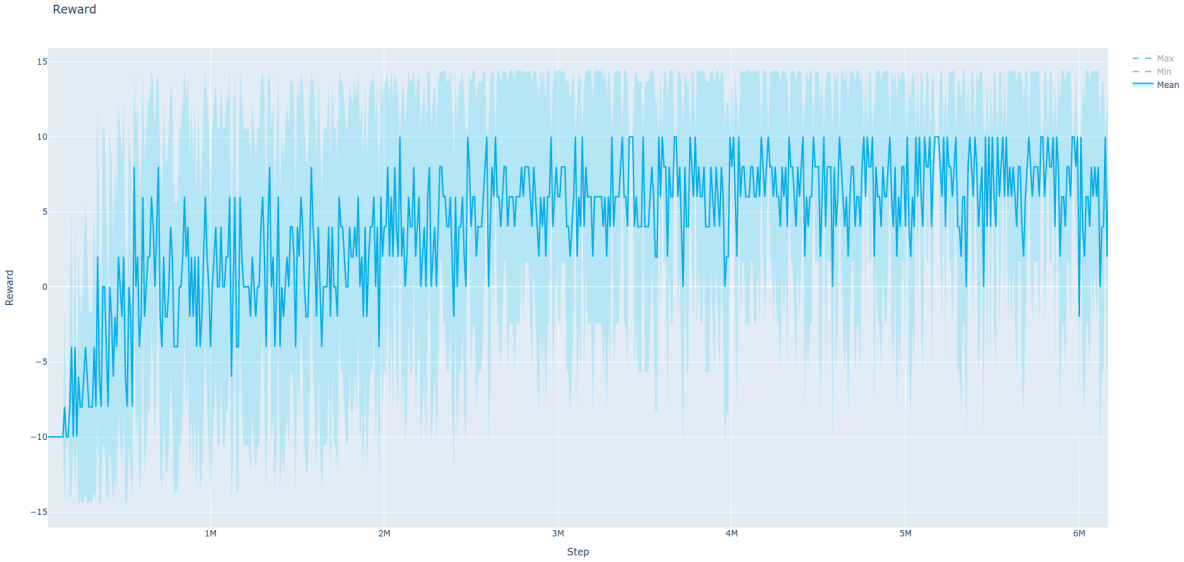

Figure 8: Rewards for the Simple approach

3.4.2 Extra hit reward approach

In this approach, during training we extract from the environment the information about if the ball was successfully hit by our agent and add 0.25 reward with a maximum of +1.25. The support range is changed as well to accommodate for that. This is done only during training. Results can be seen in Figure 9 and Figure 10. The training was less stable and the results were more or less the same, so no luck here. The agent just learned to be more defensive, but not more *successfully offensive*.

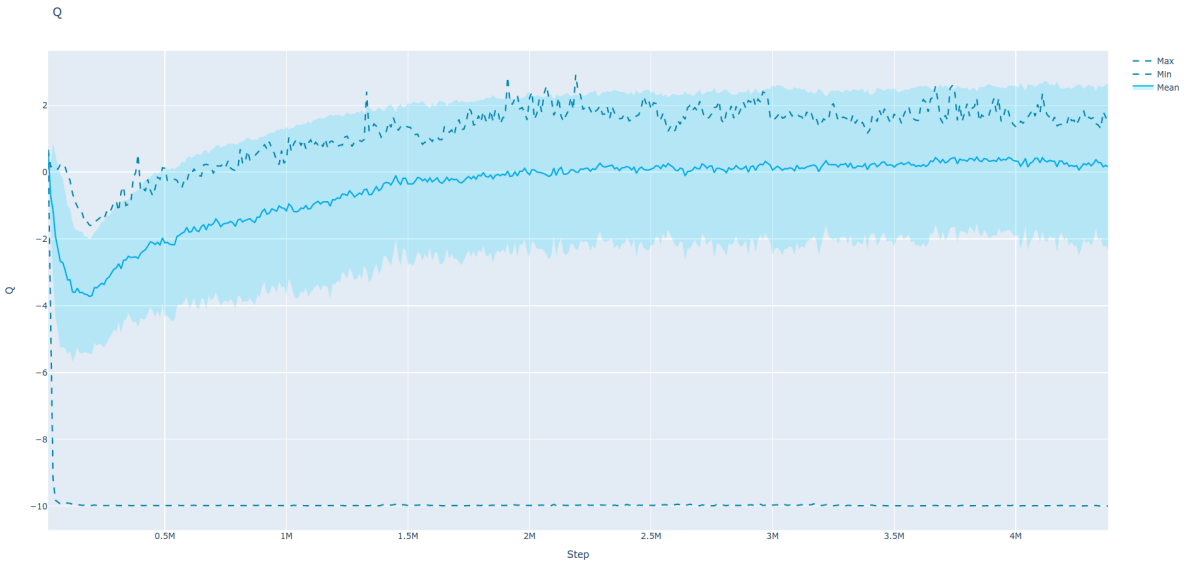

Figure 9: Q values for the Extra hit reward approach

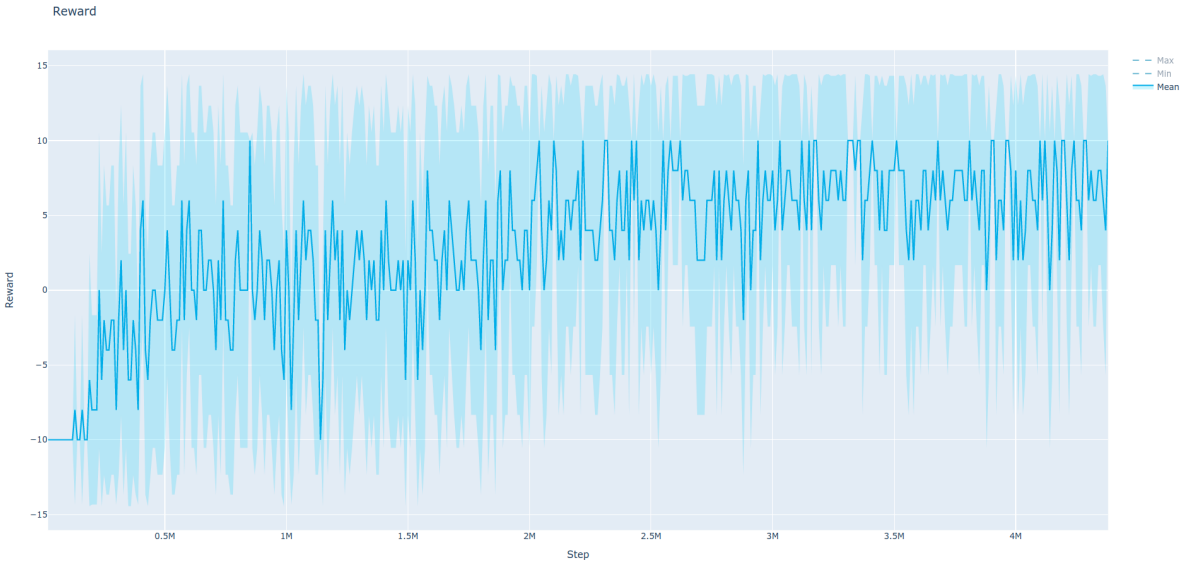

Figure 10: Rewards for the Extra hit reward approach

3.4.3 Cropped Opponent approach

In this approach, we crop the rightmost side of the image to crop out the opponent paddle during preprocessing, making the model to focus only on the agent's paddle and the ball. This method proved to significantly decrease overfitting, achieving 85+% winrate against other opponents, but only 70% against Simple AI since it doesn't overfit to it. The latter also makes the evaluation plots more noisy and the training more unstable and difficult as it has less information to win against Simple AI. For our goal though this isn't much of a problem as the general performance is greatly improved. You can see the plots in Figure 11 and Figure 12. This approach gave us insights on how to continue with an even better idea.

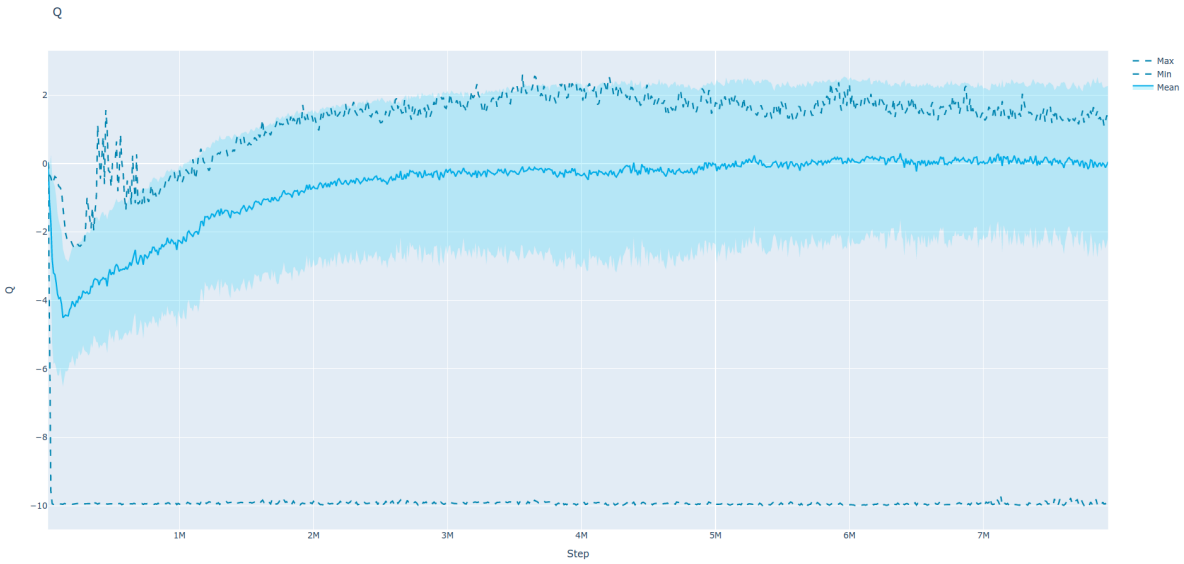

Figure 11: Q values for the Cropped opponent approach

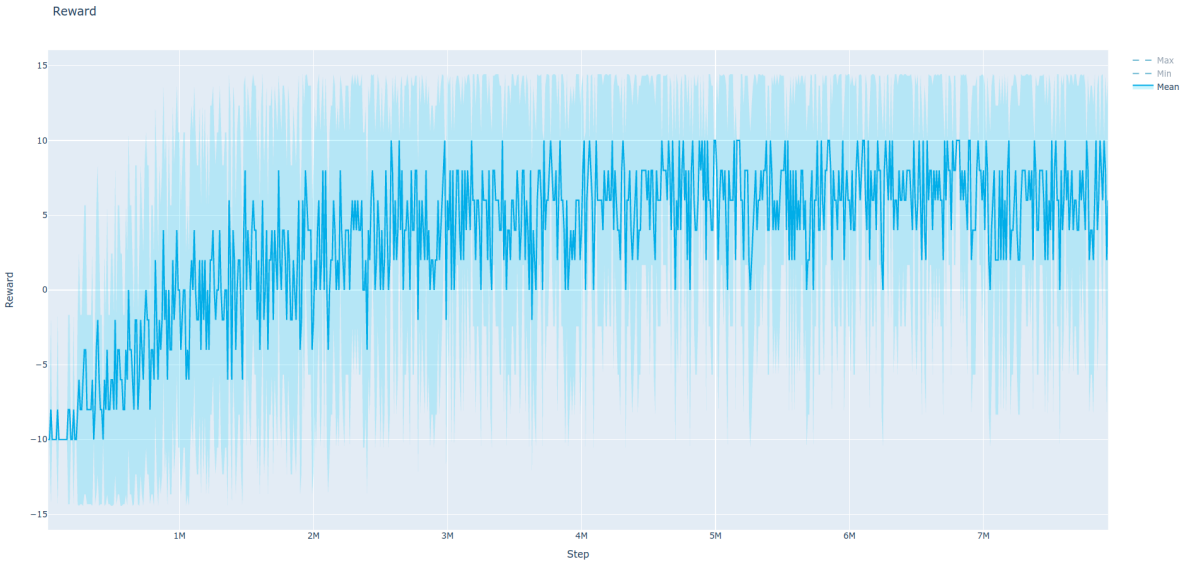

Figure 12: Rewards for the Cropped opponent approach

3.4.4 Simple + Cropped Opponent approach

In this approach, we essentially used a simple form of transfer learning by taking the agent trained with the Simple approach (without cropped opponent) and continuing the training with the Cropped opponent approach. This way the overfit agent could expand it's pong knowledge horizon further and get out of the overfitted trap without losing it's grip against Simple AI, but even becoming better at it. The winrate against anything we tried reached 95+% and the rendering of the agent showed that it makes very smart moves. This is our final and best agent. Plots can be found in Figure 13 and Figure 14 and are the plots starting right after the Simple approach plots. The rewards plot is still a bit noisy, but the real performance is great.

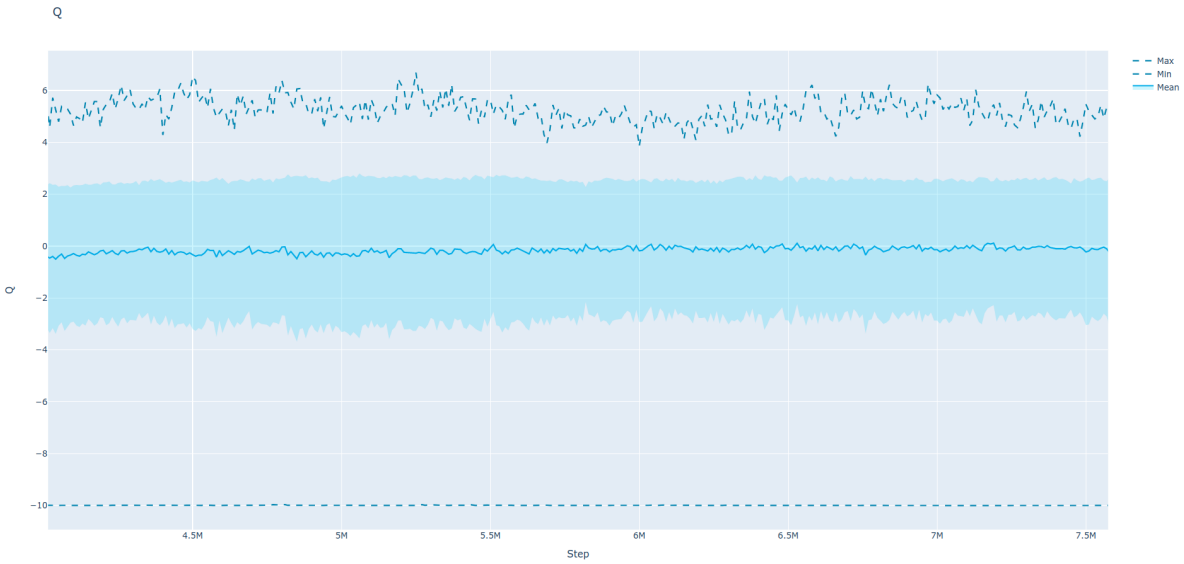

Figure 13: Q values for the Simple + Cropped Opponent approach

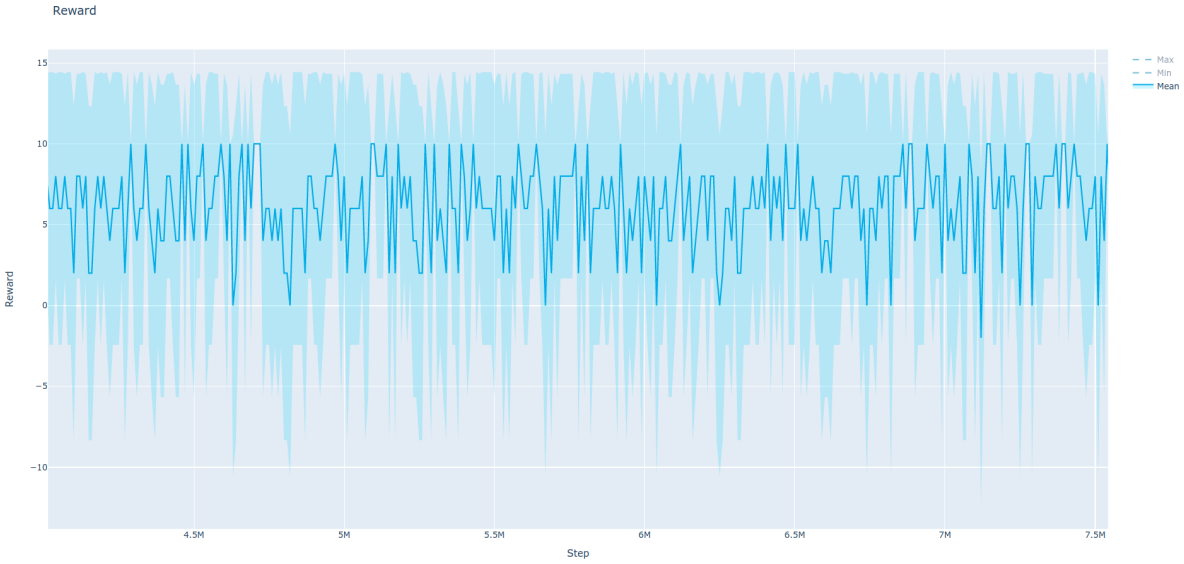

Figure 14: Rewards for the Simple + Cropped Opponent approach

Trying out the provided Battle Royale code against the provided agents get this Leaderboard in Figure 15.

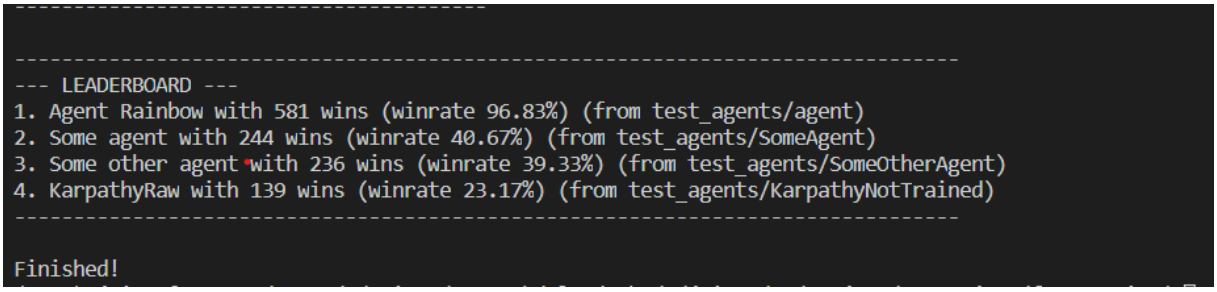

Figure 15: Simple + Cropped Opponent approach Leaderboard

4 Conclusion

This project taught us a lot about the state-of-the-art methods in the area of reinforcement learning. Extensive experiments with the Rainbow agent proved very fruitful, as the training procedure is quite powerful in the sense that, the agent quickly learns key behaviours to play against the SimpleAI and generalize that performance very well with the correct preprocessing and transfer learning. Our final agent is ready for the Mass Battle Royale!

A special mention to the Git repository of Kaixhin, from which we reused most of the code about Rainbow [17].

References

[1] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[3] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. *CoRR*, abs/1806.06923, 2018.

[4] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. 2017.

[5] Simple reinforcement learning with tensorflow part 8: Asynchronous actor-critic agents (a3c). *https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2*.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[9] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[10] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015.

[11] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[13] Kristopher De Asis, J. Fernando Hernandez-Garcia, G. Zacharias Holland, and Richard S. Sutton. Multi-step reinforcement learning: A unifying algorithm. 2017.

[14] J. Fernando Hernandez-Garcia and Richard S. Sutton. Understanding multi-step deep reinforcement learning: A systematic study of the dqn target, 2019.

[15] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning, 2017.

[16] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2017.

[17] Rainbow github repository. *https://github.com/Kaixhin/Rainbow*.