Rainbow - DQN

Dominik Schmidt
Vienna University of Technology
e11809917@student.tuwien.ac.at

Abstract

Rainbow is a model-free off-policy reinforcement learning algorithm based on DQN, which includes some of the many proposed improvements to DQN. These include double DQN to decrease action value overestimation and dueling DQN to improve generalization over actions. Prioritized experience replay and multi-step bootstrapping lead to faster training and higher sample efficiency, while noisy layers enhance exploration of the environment. Finally, distributional RL replaces the estimate of the expected return with an estimate of the whole return distribution, aiming to make the value estimation problem more tractable.

In this paper, we summarize some of the core concepts of value-based reinforcement learning and review the individual components of Rainbow as well as the integrated agent. We then present some further experimental results regarding Rainbow's general performance and sample efficiency. Finally, we provide a complete implementation of Rainbow. Our implementation achieves highly competitive results, exceeding the original results in 40% of games while using 20x less data and training significantly faster.

1 Introduction

The fundamental goal of reinforcement learning is learning how to behave in an unknown environment in order to maximize some scalar reward signal. In each step of this sequential decision-making task (see figure 1), the agent observes the environment's current state, performs some action, and then receives the next state as well as a scalar reward.

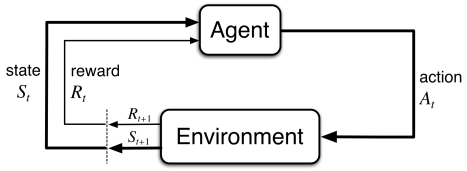

Figure 1: Agent-environment interaction [from Sutton and Barto, 2018]

Depending on the size and complexity of the environment, this task can range from being fairly easy to being almost intractably difficult.

Some of the main challenges in RL are [Sutton and Barto, 2018]:

- *Credit assignment problem*: Given a non-zero reward at some time step t, it may be difficult to discern which (if any) of the agent's preceding actions led to receiving this reward.
- *Delayed or sparse rewards*: Many environments produce non-zero rewards only very infrequently or much later than the actions that contributed to receiving the reward.

2nd TUML Seminar (2020/21).

- Agents need to explore their environment to learn how to behave optimally. Yet, to do so, they may need to temporarily deviate from what they currently consider optimal behavior. The need to balance exploration and (reward) exploitation is referred to as the *exploration vs. exploitation dilemma*.
- *Data efficiency*: Current RL algorithms need vast amounts of data to perform reasonably well. Generating such data is easy when working with small simulated environments but can become difficult when RL is applied to real-world tasks such as in robotics.

While challenging, being able to solve the reinforcement learning problem in a diverse set of environments is very useful since many practical problems such as scheduling, resource allocation, and control problems can be formulated as RL tasks.

1.1 A Brief Primer on Reinforcement Learning

The agent-environment interaction is typically formalized as a Markov Decision Process [Sutton and Barto, 2018]:

Definition 1.1 (Markov Decision Process)**.** An MDP is a 4-tuple $\langle \mathcal{S}, \mathcal{A}, R, P \rangle$ where

- $\mathcal{S}, \mathcal{A}$ are sets of states and actions respectively
- $R\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function that assigns each transition from state s_t to state s_{t+1} via action a_t a real valued reward $r_t = R(s_t, a_t, s_{t+1})$
- $P\colon \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the transition probability function that specifies a conditional probability $p(s_{t+1}|s_t, a)$ of transitioning into state s_{t+1} after executing action a in state s_t.

Definition 1.2 (Return)**.** The return $G_t = \sum_{k=t+1}^{T} R_k$ is the sum of all collected rewards from time step t until the end of the episode. [Sutton and Barto, 2018]

A policy is a function of the environment's current state, telling the agent which action to perform. We differentiate between two different kinds of policies:

Definition 1.3 (Policy)**.** A deterministic policy is a function $\mu\colon \mathcal{S} \to \mathcal{A}$ that maps each possible state to an action. A stochastic policy is a function $\pi\colon \mathcal{S} \times \mathcal{A} \to [0, 1]$ that defines a probability distribution over actions for a given state. [Achiam, 2018]

Definition 1.4 (Value Functions)**.** The state-value function $v_\pi(s)$ represents the expected return when starting in state s and following the policy π. The action-value function $q_\pi(s, a)$ gives the expected return when starting in state s, executing action a, and following the policy π. Both functions satisfy their respective, recursive Bellman equation:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_{a\sim\pi}[r(s, a) + \gamma v_\pi(s')] \\ q_\pi(s, a) &= \mathbb{E}[r(s, a) + \gamma \mathbb{E}_{a'\sim\pi}[q_\pi(s', a')]] \end{aligned} \tag{1}$$

We denote the optimal policy[1] (the policy that achieves the maximal return) by π_*.

Using this notation, we can thus concisely express the goal of reinforcement learning as finding some policy π that achieves the value of the optimal policy π_* as closely as possible.

Algorithms in reinforcement learning can roughly be grouped into two categories. *Policy-based RL* algorithms directly learn a policy that maximizes the given reward function. *Value-based* algorithms estimate the optimal action-value function $Q_*(s, a)$ and implicitly derive a policy, typically by finding the action that maximizes the action-value function, i.e. $\mu(s) = \max_{a\in\mathcal{A}} Q(s, a)$.

2 Related Work

This section covers some of the orthogonal and more recent directions of research in deep reinforcement learning.

[1] The optimal policy is not necessarily unique, but all optimal policies achieve the unique optimal value function v_*. [Sutton and Barto, 2018]

2.1 Model-based RL

Model-based reinforcement learning algorithms explicitly learn a model of the environment and then use this model to derive a good policy through planning [Sutton and Barto, 2018]. This can have several advantages, such as better data efficiency and explainability, but it can be difficult in environments with large and complex observation spaces [Moerland et al., 2021]. Due to these difficulties, much of the previous research in this area has focused on classical board games such as Checkers, Chess, and Go, where the observation space is comparatively simple. More recently, Schrittwieser et al. [2020] introduced MuZero, a model-based RL algorithm that achieves state-of-the-art performance on the Atari Learning Environment [Bellemare et al., 2013] by only modeling the parts essential for solving the RL task: the value function, the policy, and the rewards.

2.2 Policy-based RL

Policy-based RL algorithms explicitly represent and directly optimize the policy, usually in an on-policy fashion. Common policy-based algorithms include A2C/A3C [Mnih et al., 2016] which directly perform gradient ascent based on the policy gradient, and Proximal Policy Optimization (PPO), which optimizes a closely related surrogate objective instead.

2.3 Agent57 and R2D2

Agent57 [Badia et al., 2020a] and R2D2 [Kapturowski et al., 2019] are value-based reinforcement learning algorithms that follow in the lineage of Rainbow. Both are distributed RL algorithms, meaning that environment interaction is decoupled from the learner and its (prioritized) replay buffer. Similar distributed training strategies were also employed in Nair et al. [2015], Horgan et al. [2018], Espeholt et al. [2018] and Gruslys et al. [2018]. Like Espeholt et al. [2018], R2D2 also used recurrent neural networks to take advantage of longer state histories.

Agent57 builds off of the methods developed in Kapturowski et al. [2019] but combines them with two intrinsic reward based exploration mechanisms: the curiosity-based *Random Network Distillation* [Burda et al., 2018] and *Never Give Up* exploration [Badia et al., 2020b], aimed at increasing long and short term state-space coverage respectively. Additionally, a UCB bandit [Sutton and Barto, 2018] algorithm is used to control the amount of exploration and the value of the γ discount factor in each of the distributed actors.

3 Rainbow

Rainbow is a value-based reinforcement learning algorithm based on Deep Q-Networks, which is itself based on classical Q-Learning. Over the years, researchers have identified several possible improvements and extensions to Q-Learning, some of which the authors of Rainbow integrated into a single unified framework. They empirically demonstrated that this framework outperforms previous approaches on a large and diverse set of tasks, the Atari Learning Environment. [Hessel et al., 2017]

A slightly modified version of Rainbow introduced in van Hasselt et al. [2019] and referred to as *Data-efficient Rainbow* trades off reduced computational efficiency in exchange for significantly improved data efficiency.

3.1 Tabular Q-Learning

Q-Learning (see algorithm 1) is a value-based RL algorithm that explicitly maintains a table of estimates of $Q(s, a)$, for each action a and state s. Starting from an arbitrarily initialized Q, it repeatedly acts in the environment according to a policy derived from the current action value estimate while updating its estimates at each step [Sutton and Barto, 2018].

The derived policy is ϵ-greedy, meaning it generally takes strictly optimal actions with respect to the current action-value estimates, but may, with a small probability of ϵ, deviate from the optimal behavior to facilitate exploration of the environment.

More generally, Q-Learning falls into the class of temporal difference (TD) learning methods since it grounds value estimates for the current time step in estimates at future steps (bootstrapping). In

the algorithm below $Q(s, a)$ is also referred to as the TD-estimate, while $r_t + \gamma \max_a Q(s', a)$, is referred to as the TD-target. [Sutton and Barto, 2018]

Algorithm 1: Q-Learning for estimating $\pi \approx \pi_*$ [from Sutton and Barto, 2018]

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize lookup table $Q(s, a)$ arbitrarily with $Q(s', a) = 0$ for every terminal state s'.
for each episode **do**
  Initialize state S
  while S is not terminal **do**
    Choose next action A using ϵ-greedy policy derived from Q
    Take action A, observe state S', reward R
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$

While Q-Learning works well for small toy problems, it does not generally scale to problems with large or infinite state and action spaces since each action-value estimate $Q(s, a)$ needs to be explicitly stored as an element in the lookup table.

3.2 Deep Q-Network

More tractable approaches employ general function approximators such as neural networks to represent the value estimates. Even as the number of states becomes intractably large, neural networks with reasonable numbers of parameters may still be able to approximate the value function sufficiently well.

Algorithm 2: Deep Q-learning with Experience Replay [from Mnih et al., 2015]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
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 ϵ select random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta^-) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))$ with respect to θ
    Every C steps reset $\hat{Q} = Q$

Mnih et al. [2013] first introduced the use of convolutional neural networks in place of a lookup table. Their complete algorithm, called Deep Q-Networks (DQN; see algorithm 2), introduced two further changes to Q-Learning that are essential for good performance when using function approximation. *Experience Replay* detaches environment interaction and training steps by adding collected environment transitions to a large replay buffer which is sampled from during training. A second modification referred to as "fixed Q-targets" prevents an issue caused by the fact that each update step modifies both the Q-estimate and Q-target. This harmful correlation is prevented by introducing a second Q-target network which is only periodically updated.

3.3 Double Q-Learning

The TD-target equation in DQN, $r_j + \gamma \max_{a'} Q(\phi_{j+1}, a', \theta^-)$, includes a $\max$ operator over estimated action-values. This is not an issue when $Q = Q^*$, but during training, while value estimates

are still approximate and thus noisy, this leads to a systematic overestimation of action values since the maximum of a random sample is biased upwards.

While so-called "optimism in the face of uncertainty" [Kaelbling et al., 1996] is commonly known to be an effective exploration strategy, van Hasselt et al. [2015] showed that in the case of DQN, this systematic overestimation of action values does result in decreased empirical performance and present a simple modification to the DQN Q-target that solves this issue. Their modified Q-target disentangles the $\max$ operation into an action selection and action-value computation.

$$r_j + \gamma Q(\phi_{j+1}, \arg\max_a Q(\phi_{j+1}, a; \theta), \theta^-) \tag{2}$$

3.4 Dueling Q-Learning

Dueling Q-Learning is a small modification to the neural network that is used as the value function estimator. Instead of producing a value estimate for each action, the dueling architecture presented in Wang et al. [2016] separately estimates the state-value and advantage functions to compute the actual action-value estimate as $Q(s,a) = V(s) + [A(s,a) - \frac{1}{|\mathcal{A}|}A(s,a)]$. This improves the neural network's ability to generalize over actions, especially when the number of available actions is large.

3.5 Prioritized Experience Replay

Prioritized experience replay, introduced in Schaul et al. [2016], replaces the uniform sampling from the replay buffer by a sampling strategy that prioritizes samples by the magnitude of their TD-error (the absolute difference between the TD target and estimate). Intuitively, more informative transitions are then sampled more often and thus are used for training more often. Additionally, the authors suggest using importance sampling to partially correct for the bias introduced by the non-uniform sampling and demonstrate that their algorithm accelerates learning and improves results compared to vanilla DQN.

3.6 Multi-step Bootstrapping

N-step bootstrapping replaces the temporal difference target $r_j + \gamma \max_{a'} Q(\phi_{j+1}, a', \theta^-)$ with the n-step TD-target $r_j^{(n)} + \gamma^n \max_{a'} Q(\phi_{j+1}, a', \theta^-)$ where $r_t^{(n)}$ is the truncated n-step return $\sum_{k=0}^{n-1} \gamma^{(k)} r_{r+k+1}$. This effectively causes updates in value estimates to propagate through multiple states at once, thereby accelerating learning. [Sutton and Barto, 2018]

Interestingly, Fedus et al. [2020] showed that among the different components of Rainbow, the use of n-step returns is essential for taking advantage of larger replay buffers.

3.7 Noisy Nets

DQN uses ϵ-greedy exploration, meaning that at each time step, it may, with a probability of ϵ, take a random action. While this guarantees complete state coverage in the limit, it is not very efficient. A different approach suggested in Fortunato et al. [2019] is to employ controllable parameter noise to enable the neural network itself to control how much and in which directions to explore.

3.8 Distributional RL

The main idea behind distributional RL is to replace the estimate of the expected return of a state with an estimate of the entire return distribution. Distributional RL methods differ mainly in how they parametrize this estimate and compute the loss.

C51 [Bellemare et al., 2017], the variant used in Rainbow, models the return distribution as a distribution with discrete support and places the probability mass on a set of $N = 51$ atoms. Alternatively, QR-DQN [Dabney et al., 2017] estimates the 200-quantiles of the distribution. An advantage of the latter approach is that the approximate magnitude of the returns need not be known in advance.

4 Implementation Details

Due to the extensive computational requirements for training Rainbow, we opted to reduce the training duration from 200 million frames to 10 million frames. We demonstrate that we can still achieve highly competitive results through changes to hyperparameters and the neural network's architecture while reducing the effective training time from 7 days to around 8 hours on a single RTX 2080TI GPU.

4.1 Hyperparameter Tuning

We increased the learning rate by a factor of 4x. This significantly sped up learning when training for only 10M frames and considerably improved the final performance. We hypothesize that the reason Hessel et al. [2017] settled on the smaller learning rate was that the benefit of better convergence when using a lower learning rate outweighed the disadvantage of slowed learning when training for 200M frames.

The full set of used hyperparameters can be found in the Appendix B.

4.2 Q-Network Architecture

We replaced the small "Nature" dueling architecture with the large variant of the IMPALA CNN (see figure 2) [Espeholt et al., 2018] with 2x channels as modified in Cobbe et al. [2020]. Like in Cobbe et al. [2020] and Espeholt et al. [2018], we found that this significantly increased learning speed, sample efficiency, and final overall performance.

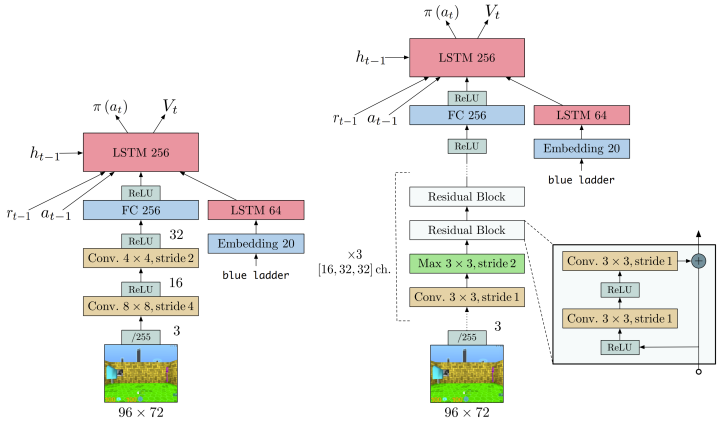

Figure 2: Small (left) and large (right) IMPALA-CNN network architecture. [from Espeholt et al., 2018]

4.3 Performance Improvements

We implemented several changes in order to decrease the effective wall-clock training time.

- Similarly to Cobbe et al. [2020], we increased the batch size from 32 to 512. As suggested in Stooke and Abbeel [2019], we accordingly adjusted the ϵ hyperparameter for the Adam optimizer to $0.005/L$ where L is the batch size.
- Since the environment interactions need to happen sequentially and can thus not be parallelized, we instead maintain $L/8$ instances of the environment and take one step of this vectorized environment for each training step. The number of environment instances is chosen so as to preserve the ratio of environment interactions and training samples.
- We used mixed-precision training as provided by PyTorch's `amp` package.
- We perform environment steps and training steps in parallel.

Overall, these changes together decrease the training time by a factor of 2.6, from 21 to less than 8 hours. Interestingly, the increased batch size also significantly improves performance on some games, a fact that was previously observed in Cobbe et al. [2020] and was attributed to decreased gradient variance and thus more stable training.

4.4 Distributional RL

We implemented both C51 [Bellemare et al., 2017] and QR-DQN [Dabney et al., 2017] but surprisingly did not see any improvement in overall performance with either. Our final implementation thus includes neither.

5 Results and Discussion

On the Atari Learning Environment, when trained for 10M frames, our implementation outperforms Google's "Dopamine" implementation, trained for 10M and 200M frames, on respectively 96% and 64% of games. Furthermore, we exceed results from Hessel et al. [2017] (at 200M frames) on 40% of games and average human results on 72% of games. For complete scores and learning curves, please refer to Appendix A.

Notably, we can observe an almost linear improvement in scores over the training duration in many games such as TimePilot, WizardOfWor, Alien, Amidar, Asteroids, Phoenix, MsPacman, and Berzerk. This suggests that at least for those games, extending the training duration may further improve those results considerably.

Our implementation is available at `https://github.com/schmidtdominik/Rainbow` and includes a complete and highly customizable framework for preprocessing, training and evaluation. We further provide integrations for OpenAI `gym`, `procgen` and `gym-retro`. Pretrained models for all 50 tested Atari games are also included.

5.1 Q-Network Architectures and Ablations

As part of our initial hyperparameter tuning, we reproduced a small subset of the ablation studies from Hessel et al. [2017]. Furthermore, we compared a number of different neural network architectures to represent the Q-estimate: the "Nature" architecture from Mnih et al. [2013], the "Nature"-dueling architecture from Wang et al. [2016] and the small and large IMPALA-CNN networks with the number of channels scaled by a factor of 1-4x [Espeholt et al., 2018]. As can be seen in table 5.1, the full Rainbow outperforms all reduced variants. We also see that the larger IMPALA variants perform much better, both in terms of learning speed and final performance, thus matching the results from Cobbe et al. [2020].

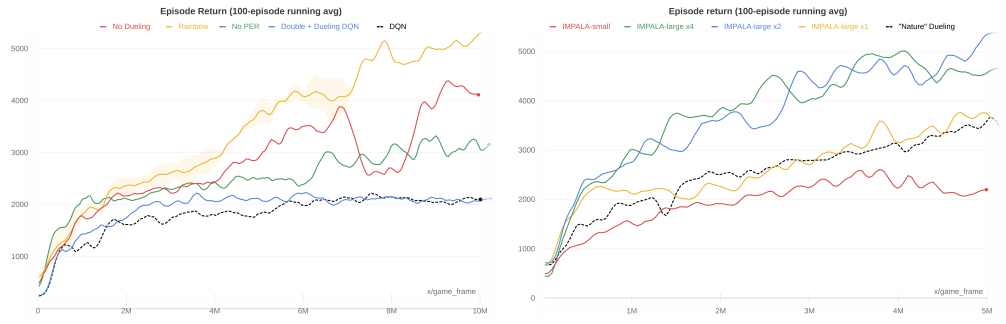

Table 1: Rainbow ablations (left) and Q-network comparisons (right) on Montezuma's Revenge.

5.2 Data-efficiency on Atari100k

We next evaluated the data-efficient variant of Rainbow [van Hasselt et al., 2019] on a set of 6 Atari games. This evaluation is performed against the Atari100k domain, meaning that the total data collected is limited to only 100k environment transitions. In figure 5.2, we can see that, at the expense

of wall-clock training time, the data-efficient variant achieves decent results, even with severely limited training data.

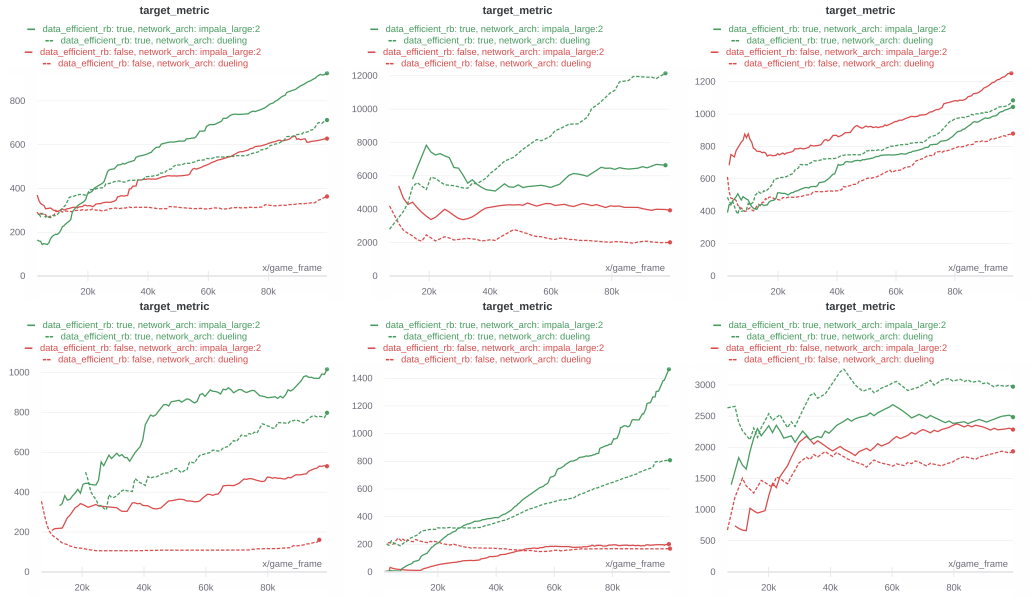

Table 2: Data-efficient Rainbow (DER) evaluated against Alien, BattleZone, MsPacman, Phoenix, Qbert and TimePilot in the Atari100k domain.

6 Evaluation Methodology

We evaluated our implementation against a subset of 50 Atari games, as described in appendix A.1, closely following the evaluation procedure from Mnih et al. [2015] and Hessel et al. [2017]. All evaluation runs lasted for 500k frames, and each of the individual episodes was no longer than 108k frames. The only change we made was that we performed the evaluation runs after training had concluded by periodically saving model checkpoints during training and then later loading them for evaluation.

7 Conclusion and Future Work

In this paper, we reviewed some of the fundamentals of value-based reinforcement learning and examined the individual components of Rainbow. We further provided an improved implementation of Rainbow that demonstrates competitive results, even when trained with considerably less data and for a shorter period of time.

Based on the gains achieved through the use of the IMPALA architecture, we would be interested in further investigating the benefits of scaling up the neural network architectures used in deep RL and integrating some of the modern many newer deep learning methods that have firmly established themselves in related fields such as computer vision.

In addition to newer curiosity-based exploration methods, we would also be highly interested in investigating the use of transfer and multi-task learning to learn to direct exploration into more informative directions.

8 Acknowledgements

We are very grateful to the TU Wien DataLab for providing the majority of the compute resources that were necessary to perform the presented experiments.

References

Josh Achiam. Spinning Up: Key Concepts in RL. https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#policies, 2018. Accessed: 2021-06-01.

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark, 2020a.

Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, and Charles Blundell. Never give up: Learning directed exploration strategies, 2020b.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013. ISSN 1076-9757. doi: 10.1613/jair.3912. URL http://dx.doi.org/10.1613/jair.3912.

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

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation, 2018.

Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning, 2020.

Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression, 2017.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.

William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay, 2020.

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, 2019.

Audrunas Gruslys, Will Dabney, Mohammad Gheshlaghi Azar, Bilal Piot, Marc Bellemare, and Remi Munos. The reactor: A fast and sample-efficient actor-critic agent for reinforcement learning, 2018.

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.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay, 2018.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning, 2019.

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

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. 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. *Nat.*, 518(7540):529–533, 2015. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

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, 2016.

Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based reinforcement learning: A survey, 2021.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning, 2015.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, and et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, Dec 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-03051-4. URL http://dx.doi.org/10.1038/s41586-020-03051-4.

Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning, 2019.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

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

Hado van Hasselt, Matteo Hessel, and John Aslanides. When to use parametric models in reinforcement learning?, 2019.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

A Full Results

Table 3: Learning curves for our Rainbow implementation for each of the 50 selected Atari environments. Each curve represents the 100-episode running average of episode returns.

	Uniform Random Policy	Human	Ours @10M	google/ dopamine @10M	DQN @200M	google/ dopamine @200M	Rainbow Paper @200M	PFRL Rainbow @200M
Alien	227.8	6,875.4	15,053.8	1,250.0	3,069.3	3,500.0	9,491.7	10,255.4
Amidar	5.8	1,675.8	2,083.4	350.0	739.5	2,400.0	5,131.2	4,284.6
Assault	222.4	1,496.4	10,898.4	1,500.0	3,358.6	3,250.0	14,198.5	15,331.9
Asterix	210.0	8,503.3	164,942.0	3,000.0	6,011.6	18,000.0	428,200.3	550,307.6
Asteroids	719.1	13,156.7	8,398.4	800.0	1,629.3	1,500.0	2,712.8	3,399.6
Atlantis	12,850.0	29,028.1	825,445.8	170,000.0	85,950.0	830,000.0	826,659.5	883,073.0
BankHeist	14.2	734.4	1,124.9	900.0	429.7	1,150.0	1,358.0	1,272.7
BattleZone	2,360.0	37,800.0	67,806.5	22,500.0	26,300.0	40,000.0	62,010.0	202,382.5
BeamRider	363.9	5,774.7	21,404.1	5,700.0	6,845.9	6,200.0	16,850.2	21,661.5
Berzerk	123.7	2,630.4	1,801.3	480.0		830.0	2,545.6	6,018.1
Bowling	23.1	154.8	28.9	40.0	42.4	40.0	30.0	62.3
Boxing	0.1	4.3	99.5	77.0	71.8	98.0	99.6	99.9
Breakout	1.7	31.8	681.7	55.0	401.2	110.0	417.5	317.8
Centipede	2,090.9	11,963.2	8,847.5	5,000.0	8,309.4	6,500.0	8,167.3	7,546.8
ChopperCommand	811.0	9,881.8	2,922.5	2,500.0	6,686.7	11,500.0	16,654.0	21,014.5
CrazyClimber	10,780.5	35,410.5	132,129.2	110,000.0	114,103.3	145,000.0	168,788.5	174,025.0
DemonAttack	152.1	3,401.3	110,345.6	3,000.0	9,711.2	16,000.0	111,185.2	100,980.6
DoubleDunk	-18.6	-15.5	-1.0	-18.0	-18.1	22.0	-0.3	-0.1
Enduro	0.0	309.6	2,216.6	1,700.0	301.8	2,200.0	2,125.9	2,281.9
FishingDerby	-91.7	5.5	33.6	20.0	-0.8	42.0	31.3	39.1
Freeway	0.0	29.6	33.0	32.0	30.3	33.0	34.0	33.6
Frostbite	65.2	4,334.7	4,983.4	2,700.0	328.3	8,400.0	9,590.5	11,046.3
Gravitar	173.0	2,672.0	2,847.3	210.0	306.7	1,250.0	1,419.3	1,387.2
Hero	1,027.0	25,762.5	19,903.0	14,000.0	19,950.3	47,000.0	55,887.4	34,234.5
IceHockey	-11.2	0.9	5.5	-5.0	-1.6	2.0	1.1	6.6
Kangaroo	52.0	3,035.0	14,280.0	6,000.0	6,740.0	13,000.0	14,637.5	13,726.5
Krull	1,598.0	2,394.6	9,143.3	6,100.0	3,804.7	6,100.0	8,741.5	7,844.4
KungFuMaster	258.5	22,736.2	22,552.4	19,000.0	23,270.0	27,000.0	52,181.0	54,835.5
MsPacman	307.3	15,693.4	5,322.8	2,400.0	2,311.0	4,000.0	5,380.4	5,277.3
NameThisGame	2,292.3	4,076.2	19,562.5	9,000.0	7,256.7	9,000.0	13,136.0	14,679.0
Phoenix	761.4	7,242.6	24,866.1	5,000.0		8,200.0	108,528.6	147,467.9
Pitfall	-229.4	6,463.7	-1.6	-3.0		-3.0	0.0	-3.4
Pong	-20.7	9.3	21.0	16.0	18.9	20.5	20.9	21.0
PrivateEye	24.9	69,571.3	360.6	0.0	1,787.6	22,000.0	4,234.0	101.7
Qbert	163.9	13,455.0	15,529.9	7,000.0	10,595.8	17,500.0	33,817.5	42,518.7
RoadRunner	11.5	7,845.0	57,881.1	35,000.0	18,256.7	55,000.0	62,041.0	67,638.5
Robotank	2.2	11.9	36.7	22.5	51.6	65.0	61.4	74.0
Seaquest	68.4	20,181.8	53,149.5	2,000.0	5,286.0	10,000.0	15,898.9	5,277.4
Skiing	-17,098.1	-4,336.9	-16,516.1	-22,000.0		-22,000.0	-12,957.8	-29,974.7
Solaris	1,236.3	12,326.7	2,510.2	1,400.0		2,200.0	3,560.3	6,730.1
SpaceInvaders	148.0	1,652.3	17,708.9	800.0	1,975.5	4,000.0	18,789.0	2,823.1
StarGunner	664.0	10,250.0	327,543.0	2,000.0	57,996.7	58,000.0	127,029.0	155,248.8
Tennis	-23.8	-8.9	23.8	-3.0	-2.5	0.0	0.0	-0.1
TimePilot	3,568.0	5,925.0	31,407.9	3,000.0	5,946.7	12,000.0	12,926.0	24,038.2
Tutankham	11.4	167.6	103.8	200.0	186.7	240.0	241.0	258.0
Venture	0.0	1,187.5	52.9	0.0	380.0	1,550.0	5.5	2.5
VideoPinball	16,256.9	17,297.6	202,204.1	40,000.0	42,684.1	450,000.0	533,936.5	292,835.7
WizardOfWor	563.5	4,756.5	18,406.7	2,800.0	3,393.3	7,500.0	17,862.5	21,341.2
YarsRevenge	3,092.9	54,576.9	102,667.8	10,000.0		45,000.0	102,557.0	93,877.2
Zaxxon	32.5	9,173.3	17,187.2	4,800.0	4,976.7	14,500.0	22,209.5	25,084.0

Table 4: Evaluation scores for each of the tested Atari games. Random and human scores are from Mnih et al. [2015] where available, otherwise from Badia et al. [2020a].

A.1 Environment selection

We evaluated our implementation against a subset of the Atari Learning Environment but excluded some games to reduce the computational burden. Excluded games are *Carnival*, *Pooyan*, *JourneyEscape*, *AirRaid* (no human or random scores were available for these), *Jamesbond*, *Riverraid* and *UpNDown* (these were not evaluated in the Rainbow paper), *Defender* and *Surround* (these are not available via `gym`), *Gopher* and *MontezumaRevenge*.

B Hyperparameters

B.1 Hyperparameters for Rainbow

This section lists the hyperparameters used in our experiments. Parameters that differ from the ones used in Hessel et al. [2017] are marked with an asterisk. As in previous work, the unit "frames" refers to the number of environment steps taken by the wrapped environment, including frame-skipping. For noisy-nets DQN we implemented the "factorized Gaussian noise" variant, with noise vectors generated on the GPU.

Parameter	**Value**
Discount factor γ	0.99
Q-target update frequency	32,000 frames
Importance sampling β_0 for PER	0.4
n in n-step bootstrapping	3
Initial exploration ϵ	1.0
Final exploration ϵ	0.01
*Exploration ϵ decay time	500,000 frames
σ_0 for noisy linear layers	0.5
*Learning rate	0.00025
*Adam ϵ parameter	$0.005/\text{batch size}$
Gradient clip norm	10
Loss function	Huber
*Batch size	512
*Parallel environments	64
Replay Buffer Size	1M transitions
Training starts at	80,000 frames
*Q-network architecture	IMPALA-large with 2x channels

B.2 Environment pre-processing hyperparameters

This section lists the settings for preprocessing environments from `gym`, `gym-retro` and `procgen`. All environments used a time limit of 108k frames (30 minutes of emulator time). Image downscaling was performed with area interpolation. For `gym` environments, we max-pooled consecutive frames and used 0-30 noop actions at the beginning of each episode as in Hessel et al. [2017].

Parameter	**Environment**	**Value**
Grayscale	gym	yes
	retro	no
	procgen	no
Frame-skipping	gym	4
	retro	4
	procgen	1
Frame-stacking	gym	4
	retro	4
	procgen	4
Resolution	gym	84×84
	retro	72×96
	procgen	64×64