MarI/O 2.0

Edward Zhang
University of California, Los Angeles
edzhang@ucla.edu

Stephanie Doan
University of California, Los Angeles
stephaniekdoan@ucla.edu

Jack Zhang
University of California, Los Angeles
dvorjackz@gmail.com

Abstract

Game-playing Reinforcement Learning agents frequently strive for human-level performance through learning methods that are generalizable across many games. In 2013, Google Deepmind developed the milestone Deep Q-learning algorithm that found success across a collection of Atari 2600 video games, reaching and sometimes exceeding the human benchmark [1]. Super Mario Bros is much more complex than most of the aforementioned Atari 2600 games in both its gameplay and observation space, making this an interesting extension of the DQN algorithm. Using state-of-the-art techniques such as double DQN's [2], dueling networks [3], and prioritized replay buffer, we trained an agent that can outperform the human baseline on Super Mario Bros for NES. Code for this project can be found at: *https://github.com/dvorjackz/MarioRL*.

1 Background

1.1 Super Mario Bros.

Super Mario Bros is a popular video game made for the Nintendo Entertainment System in 1985. The player assumes the role of Mario who walks, runs, and jumps to navigate through a complex environment and reach the finish line. Along his path, Mario collects coins and avoids or destroys enemies that can end his life upon contact. The game involves several complex platforming tasks across a diverse set of levels and stages, requiring precise button inputs to successfully conquer each level.

Super Mario Bros is classified as computationally hard [4], its difficulty stemming from the playing board being arbitrarily large. The player sees only a portion of the entire environment based on Mario's current position, so obstacles and enemies enter and leave the frame in a smooth side-scrolling platform. The seemingly infinitely long and unpredictable path of each environment and the complexity of the player's task poses an interesting challenge to explore with Reinforcement Learning.

1.2 Deep Q Learning Algorithms

We approach this task with Deep Q-learning, which first requires understanding of the traditional Q-learning algorithm [5]. Q-learning is a model-free method that directly samples from the environment to learn the optimal optimal policy in a finite Markov Decision Process. It is an off-policy method that updates its Q-values by choosing the greedy action along the target policy, while following a behavior policy that encourages exploration often via an ϵ-greedy strategy. The Q-value is updated at every time step from the current state, towards the value of the greedy action from the target policy.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \underset{a}{max} Q(s_{t+1}, a) - Q(s_t, a_t))$$

Q-learning is efficient and simple in less complicated environments with fairly small action spaces, but complicated games such as Super Mario Bros poses the challenge of a large state-action environment that would constrain memory and make it difficult to sample enough state-actions.

In 2013, Deepmind developed the Deep Q-learning algorithm, which uses a convolutional neural network to approximate the Q-value function for every state [1]. Value approximation by a neural network can cause instability, which Deep Q Networks address with two techniques. Firstly, target Q-values are calculated with a separate network whose weights are fixed and periodically synchronized with weights of the primary Q-network, stabilizing the training process. Secondly, experience replay stores and randomly samples experience tuples (s, a, r, s') from an experience replay buffer, eliminating likely correlations between these experiences. Prioritized replay has been found to further improve performance by replaying important transitions more frequently[6].

In our study, we also explore Double Deep Q-learning, which prevents overestimation of the agent's action value in a stochastic environment. Standard DQNs end up repeatedly choosing actions with the highest values instead of searching for a new highest value, preventing it from finding the optimal policy. DDQN decouples the action choice from the target Q-value calculation, reducing the likelihood for overestimation. Algorithm 1 shows the DDQN method as provided by Hasselt et al. [2] in 2015.

Algorithm 1: Double Deep Q-learning (Hasselt et al., 2015)

Initialize primary network Q_Θ, target network $Q_{\Theta'}$, replay buffer D, τ « 1;
for *each iteration* **do**
  for *each environment step* **do**
    Observe state s_t and select a_t ~$\pi(a_t, s_t)$;
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$;
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer D;
  end
  for *each update step* **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1})$ ~D ;
    Compute target Q value: $Q^*(s_t, a_t) \approx r_t + \gamma Q_\Theta(s_{t+1}, argmax_{a'} Q_{\Theta'}(s_{t+1}, a'))$;
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\Theta(s_t, a_t))^2$;
    Update target network parameters: $\Theta' \leftarrow \tau * \Theta + (1 - \tau) * \Theta'$;
  end
end

In our study, we also employ the Dueling DQN architecture to improve policy evaluation for similar-valued actions [3]. In this method, we illustrate the Q-function $Q(s, a) = V(s) + A(a)$ as the sum of function $V(s)$ and an advantage function $A(a)$, dependent on the different actions one can take from a state. The Dueling DQN decouples the calculation of the advantage function from the value function, recombining them into the Q-value at the final layer of the neural network. This technique achieves higher robustness for Q-value approximation.

2 Methodology

2.1 Model Architecture

To train our agent, we used the standard Deep Q Network algorithm and explored the effects of Double DQN, dueling networks, and prioritized experience replay. We used the Stable Baselines library, which contains several implementations of common reinforcement learning algorithms [7]. Stable Baselines is built on top of Baselines by Open AI, which is a library for the popular RL framework Open AI Gym [8].

2.2 Environment

To simulate the game, we used gym-super-mario-bros, a custom environment built for Open AI Gym [9]. It offers eight worlds each with four stages, totaling 32 separate environments. Our state space from which we make observations is the raw pixel data from each frame of the game. The full NES action space of 256 actions is condensed into actions lists of 12 or fewer actions, with which we experimented with chose one the *simple movement* action set with 7 choices. These choices include moving left and right as well as jumping, among others.

2.3 Rewards

The agent is rewarded for moving to the right and progressing towards the goal. It is penalized for the passage of time and heavily penalized for death. Rewards are clipped between -15 and 15 to limit the scale of loss, thus capping the backpropagated gradient and providing more stable training. While clipping rewards between -1 and 1 is a common method used by papers such as Deepmind's 2015 Atari agent, we felt that the range of representation was not enough to differentiate between different rewards. For example, we needed death to produce a much more negative reward than passage of time.

In designing the reward function, we tried not to use highly engineered features, instead using features that felt natural from a human perspective. For example, we would not reward jump height (which would greatly speed up training for stages with tall pipes), since an actual human learning the game for the first time would not naturally be incentivized to jump high and there may end up being no tall pipes at all in any stage.

We chose to penalize passage of time only in the second half of training, because including this penalty for the entire training would cause the agent to devolve at times into learned helplessness, whereby it would choose to prematurely die in order to avoid suffering further penalties [10]. Because we valued time only under the context that the level has already been beaten (since we want Mario to beat the level as quickly as possible), we first trained without time penalty to ensure Mario beats the level. We later included time penalty to ensure Mario beats the level as quickly as possible. This is a minimally engineered decision since it is in the natural flow of a human learning the game to first attempt to beat the level before doing speed runs.

3 Experimentation

To discover hyperparameters for our most performant agent, we trained and evaluated performance with and without various enhancements, specifically the addition of Double DQN, prioritized replay buffer, and Atari environment wrappers.

3.1 Prioritized Experience Replay

Experience replay is a critical aspect of Deep Q-learning, stabilizing training and improving sample efficiency by reusing experiences. Prioritized experience replay is known to improve training efficiency by biasing the sample distribution on the TD error, optimizing this recycling of past experiences. This is especially important in environments like Super Mario Bros., where the large negative reward that is incurred upon dying is both sparse and has high importance.

To confirm that PER gave us better results than uniform experience replay, we trained our agent using both methods. Figure 1 illustrates the TD error, loss, and reward for a DDQN algorithm run with and without prioritized replay. With prioritized replay, we observe lower variance in TD error and faster convergence to zero, higher average reward, and drastically faster decrease of loss to zero, indicating higher stability in training compared to uniform experience replay.

3.2 Double DQN

We also experimented with Double DQN as opposed to standard DQN. In Figure 2, DDQN shows significantly higher loss but higher reward. Higher reward indicates further advancement of the agent through the obstacle course, indicating that Double DQN contributes to faster increase in performance during training due to the reasons outlined in our previous discussion.

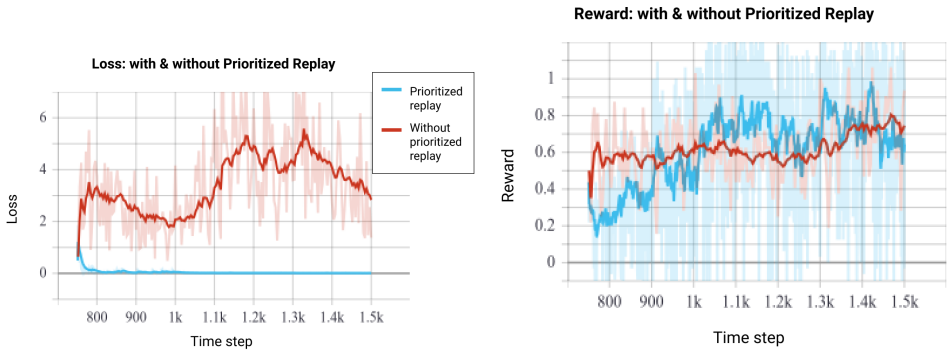

Figure 1: Loss and reward per time step with and without prioritized experience replay.

The DDQN TD error, loss, and reward are noticeably noisier than DQN's corresponding metrics. This can be attributed to the fact that DDQN trains two different Q estimates, making each training step less stable. However, the reward per timestep is clearly a significant improvement over DQN, and thus we used the DDQN algorithm in our final training process.

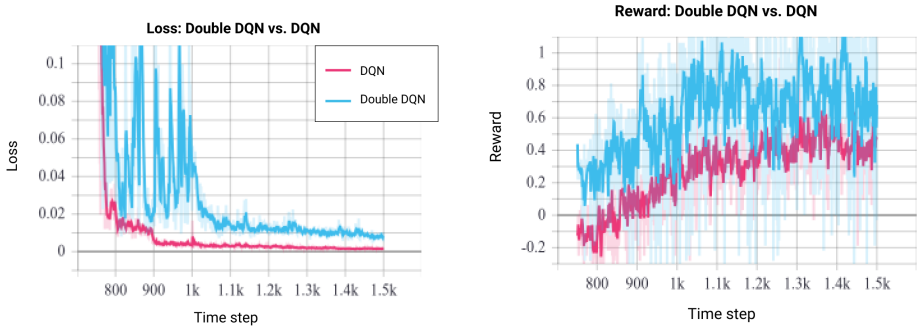

Figure 2: Loss and reward per time step for Double DQN and standard DQN.

3.3 Environment Wrappers

3.3.1 Warpframe

Super Mario Bros. has an enormous observation space. The NES has a resolution of 256×240 pixels. With each pixel having three channels ranging from $0 - 255$, we can see that our observation space has $\sim 1 \times 10^{12}$ different states. To improve training speed, we downsample our observation space by transforming the $256 \times 240 \times 3$ image into an $84 \times 84 \times 1$ grayscale image. We find that this provides a significant boost to training speed due to the vastly reduced input size while maintaining similar performance.

3.3.2 Framestack

The reinforcement learning agent initially observes only a single frame at each timestep. Using only a single frame, it is difficult to capture movement in the scene and thus gauge whether the player character is in danger. This is important in Super Mario Bros., as Mario's movement has momentum, which can cause him to slide into obstacles or enemies.

We attempt to capture velocity and speed information in the game by concatenating the previous four frames as input into our DQN. Intuitively, this allows our model to adequately capture movement information and Mario's movement states, such as jumping, falling, and fast or slow movement to the left or right. We find that this gives our model a significant performance boost, confirming our intuition that velocity information is vitally important to successfully playing Super Mario Bros.

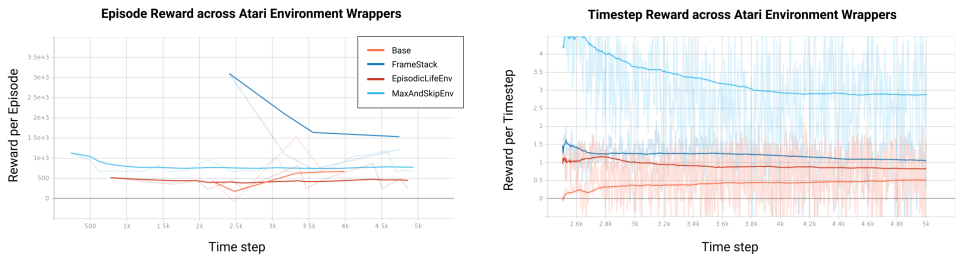

Figure 3: Reward per timestep and per episode for each environment wrapper used in isolation.

3.3.3 Frame Skipping

Using this wrapper, the agent only observes and selects an action every kth frame, repeating the selected action for the frames in between each observation. Since selecting an action requires a model prediction and is much more computationally complex that rendering a single frame, this effectively speeds up training by k times. Our project sets k to be equal to 8.

An additional benefit of skipping frames is that it allows an agent to reach a max height jump more consistently, which is the only way to progress past tall pipes, which was a major roadblock before we implemented frame skipping. A max height jump requires 24 consecutive frames of jump actions. Thus, without frame skipping, the agent would need to learn to select a jump action 24 times in a row, which is a highly improbable event that would take a long time for the agent to learn. By skipping 8 frames at a time, we only need the agent to select a jump action 3 times in a row, which is much easier for Mario to learn. A downside is that Mario's actions are less precise, but we found that this precision is not needed in order for him to complete the level.

Figure 4: An example of a tall pipe.

3.3.4 Episodic Life

In Super Mario Bros., the player initially has three attempts, or "lives", to complete the game. Coming into contact with an enemy or lethal obstacle, falling off the bottom of the level, or running out of time causes Mario to lose a life. At the start of each life, Mario is placed at the beginning of the level. Losing all three lives results in a game over.

The default configuration of our environment is such that the environment counts losing all three lives as a single episode. This can be problematic because it causes the agent to maximize the average reward for three attempts. Thus, an agent that progresses very far in the level on the first attempt, but fails very early on subsequent attempts, will be considered worse than an agent who only progresses a short distance in all three attempts.

A final modification to our environment is making each episode equivalent to the span of a single in-game life. This helps our model converge, as performance is dictated by the agent's the performance of a single attempt on the level rather than the average of three attempts.

4 Results

After experimentation, we trained one agent solely on Stage 1-1, Stage 1-2, one agent only on Stage 1-2, and another agent on random stages each for 2 million time steps. At the approximate halfway point of 1 million time steps, we observed that Mario learned to beat the level, after which it began optimizing its runs to avoid time penalties. Figures 5 and 6 illustrate the reward and loss over training for these respective agents; the continued reward increase and loss decrease suggest further improvements are possible with more training.

To evaluate the performance of our agents, we let each agent play for 100 episodes. The agent that was trained only on Stage 1-1 played Stage 1-1 100 times, and respectively with Stage 1-2 and random stages. To benchmark our performance, we also evaluated an agent which picks random actions and a beginner human player for 100 episodes each. Figure 7 shows the the relative performance of each model compared to these baselines.

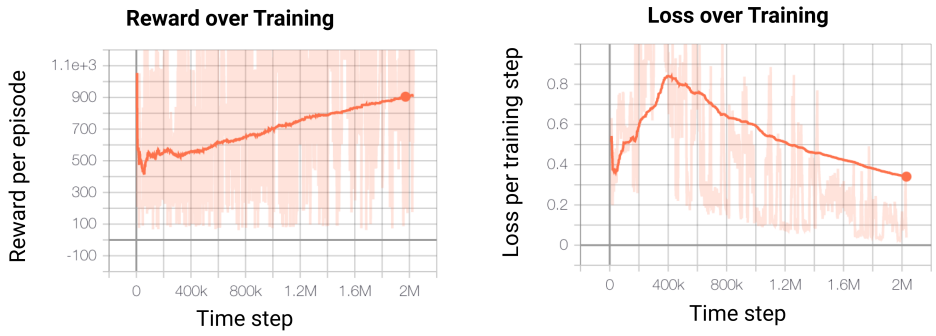

Figure 5: Reward and loss over the training period of our most performant agent trained on the same starting level, advancing Stage 1 to 2 and so on.

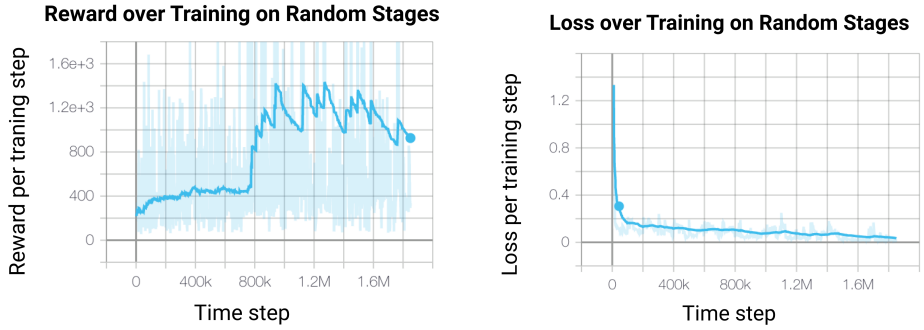

Figure 6: Reward and loss while training Mario agent on randomly stages of Super Mario Bros.

5 Discussion

While our agent was able to perform extremely well on the specific level it was trained on, it was unable to generalize to other levels. While our agent trained on Stage 1-1 was able to clear its stage with high consistency and achieve an average reward of 1700.70, when run on Stage 1-2, it visibly chose illogical actions and failed to pass the level, only averaging a reward of 874.18. Similarly, our agent trained on Stage 1-2 consistently cleared Stage 1-2 but was unable to make any significant progress on Stage 1-1.

However, the inextensibility of the model to unseen stages is not necessarily a shortcoming of our training. The goal of the project was to train an agent to play the specific game, Super Mario Bros. This means doing well on all existing levels, not necessarily on hypothetical platforming levels not actually present in the game. Thus, we also trained an agent on all stages (with stages picked randomly), with the goal of having it perform well on all stages in the game. Since there are 32 total environments, this would require a significantly higher number of training iterations than our one-stage agents. Unfortunately, we did not have the time nor computational resources to perform this training in a reasonable amount of time (one-stage agent training already takes approximately 48 hours). However, we were able to train the agent enough (36 hours) so that we could observe its continually increasing average reward and decreasing loss. This strongly suggests that upon training for the full number of iterations, Mario would be able to achieve decent performance on all levels. If we were actually to carry out this training fully in the future, we would also likely increase the complexity of our CNN network.

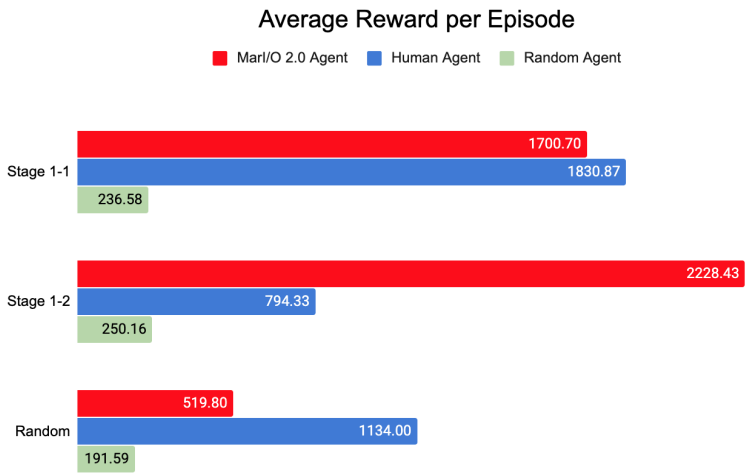

Figure 7: Average reward earned by our agent trained on 2 million timesteps and our agent trained on 1.85 million time steps on random stages for each episode, compared to a novice human baseline and a random agent.

Another notable realization during our experimentation was that our agent was prone to catastrophic forgetting [11], a phenomenon commonly found in neural networks where online incremental training would cause a network to lose its ability to predict points similar to previous training data. We found that our agent trained to consistently beat Stage 1-1, would eventually completely lose its ability to play Stage 1-1 when subsequently trained solely on Stage 1-2. Thus, for our generalized agent trained on all stages, rather than training on each stage sequentially, we picked a random stage for each training episode.

6 Conclusion

Through our study, we found that Double Q-learning with enhancements could surmount human-level performance on Super Mario Bros or the NES. Our agent was able to perform mechanically precise actions that easily outperform inexperienced human players. As shown in Figure 7, a human agent slightly outperforms the trained agent on Stage 1-1, which does not require precise movements. However, Stage 1-2 requires navigating tight crevices and carefully timed jumps, which the trained agent can perform easily after initially training on Stage 1-1. However, precision is an advantage of artificial agents over human skill, not specific to the Deep Q-learning algorithm.

The relatively poor performance of the agent trained on random stages chosen at episode start can be attributed to insufficient training, as there is high variance at around 2 million time steps (Figure 6), a trend seen at the start of the Stage 1-1 training before there is a steady increase (Figure 5). Our agents trained in both settings show potential to further improve with continued training.

DQN and its subsequent improvements have been shown to achieve good results in Atari 2600 games. Our study shows that these techniques show promise with more significantly more complex environments.

6.1 Further Research

We look to extend our study on Deep Q-learning by exploring recent enhancements to DQN, such as distributed agents like ApeX and Gorila DQN [12], and Recurrent Replay Distributed DQN (R2D2) [13] which leverages artificial recurrent neural networks, specifically Long Short Term Memory (LSTM) [14]. Never Give Up further builds upon R2D2 by using episodic memory to detect undiscovered parts of the game for exploration [15]. The emphasis on exploring for higher rewards would be especially beneficial for higher levels of Super Mario Bros, in which the desired path in the environment much more complicated than moving forward and jumping to avoiding obstacles.

We also briefly explored goal-based exploration as an alternative to simply selecting random actions. With Super Mario Bros, Pardo et al. developed a Q-Map algorithm that uses a CNN to select screen coordinates as goals from screen frames as input, and then follows that trajectory for several steps [16]. After 3 million timesteps, the Q-Map DQN was found to achieve higher rewards than a standard DQN that explores with random actions. We aimed to augment the Stable Baselines implementation to utilize Q-Map, but the training requirement was too extensive for our time frame, especially since the Q-Map DQN was found to require more computation and training time.

6.2 Final Remarks

We would like to thank Professor Lin Yang and the teaching staff for putting together this class and providing the opportunity for this project. Each member of our team contributed to the research, development, and analysis of this study.

References

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

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

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

[4] Greg Aloupis, Erik D. Demaine, and Alan Guo. Classic nintendo games are (np-)hard. *CoRR*, abs/1203.1895, 2012.

[5] Christopher J. C. H. Watkins and Peter Dayan. Q-learning, 1992.

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

[7] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[8] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[9] Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018.

[10] Vali Derhami and Zahra Youhannaei. Demonstration of learned helplessness with fuzzy reinforcement learning. 12 2008.

[11] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks, 2016.

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

[13] Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, and Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019.

[14] 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.

[15] 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, 2020.

[16] Fabio Pardo, Vitaly Levdik, and Petar Kormushev. Goal-oriented trajectories for efficient exploration, 2018.