GymNESium: Deep Reinforcement Learning for the NES

Hal Kolb HAL@KOLB.CO.UK
Department of Computer Science
Manchester Metropolitan University
Manchester, England, UK

Abstract

This paper presents Gym**NES**ium, a Gymnasium environment for modelling Nintendo Entertainment System (NES) games, demonstrated by training a bleeding-edge reinforcement learning (RL) agent to high level play in Mike Tyson's Punch Out!! (MTPO). The full spectrum of modern reinforcement learning (RL) optimisations, as outlined in Rainbow Deep-Q Networks (Hessel et al., 2018) and Beyond the Rainbow (Clark et al., 2024), are applied to MTPO, showing human-level performance with minimal walltime, on desktop PC hardware. The full code for the environments and agent is split between 3 GitHub repositories: Gym**NES**ium[1] the generic NES environment, Boxing Gym[2] the MTPO specific environment, and mtpo_rainbow_rl[3] the RL agent and runnable program.

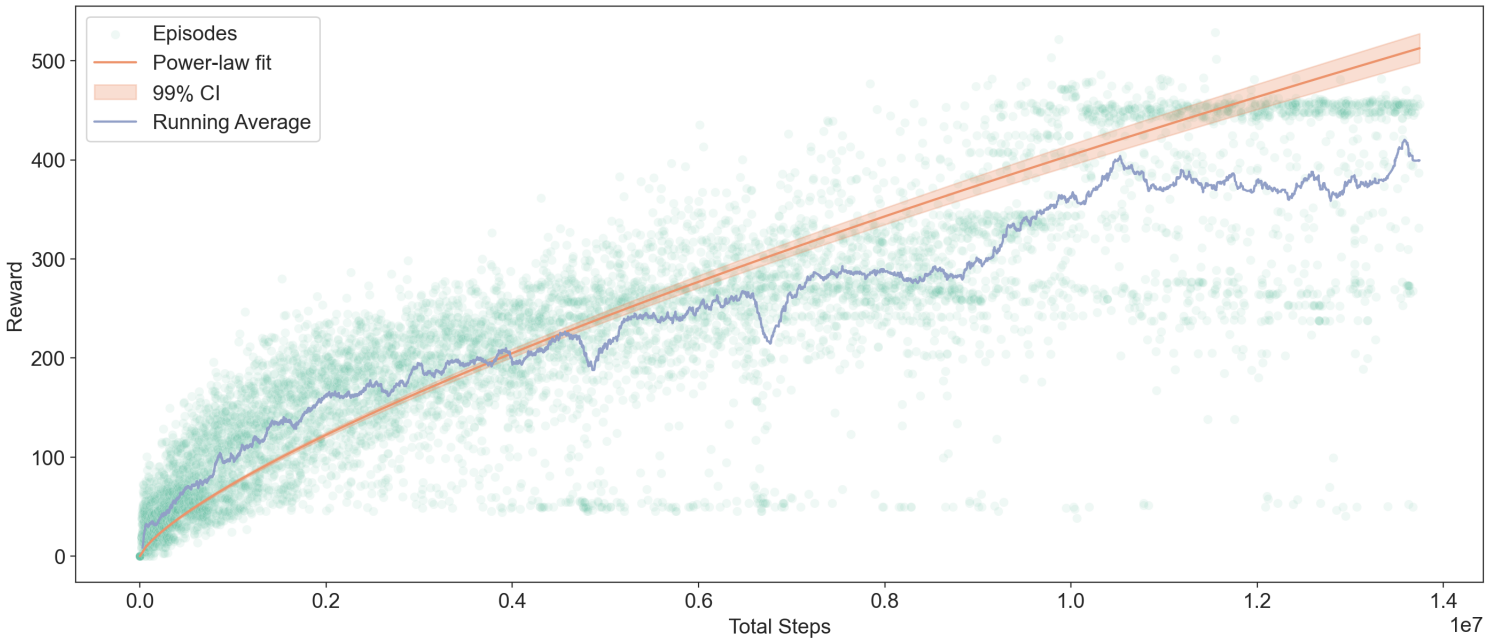

Figure 1: Total reward for each episode plotted against the number of total environment steps.

1. https://github.com/Hal609/GymNESium
2. https://github.com/Hal609/BoxingGym
3. https://github.com/Hal609/mtpo_rainbow_rl

1 Introduction

Deep reinforcement learning is a powerful machine learning technique capable of training extremely high level performance models, without the need for large data sets. RL agents optimise complex and uncertain environments through repeated interactions, evaluated with a reward function. As modern networks are becoming larger and larger and demonstrating that performance can continue to rise as they scale, the availability of high quality data is an increasingly limiting factor. The below figure from Villalobos et al. (2022) illustrates the scale of data required for modern models, predicting that by 2028 LLMs will be using 100% of available text data.

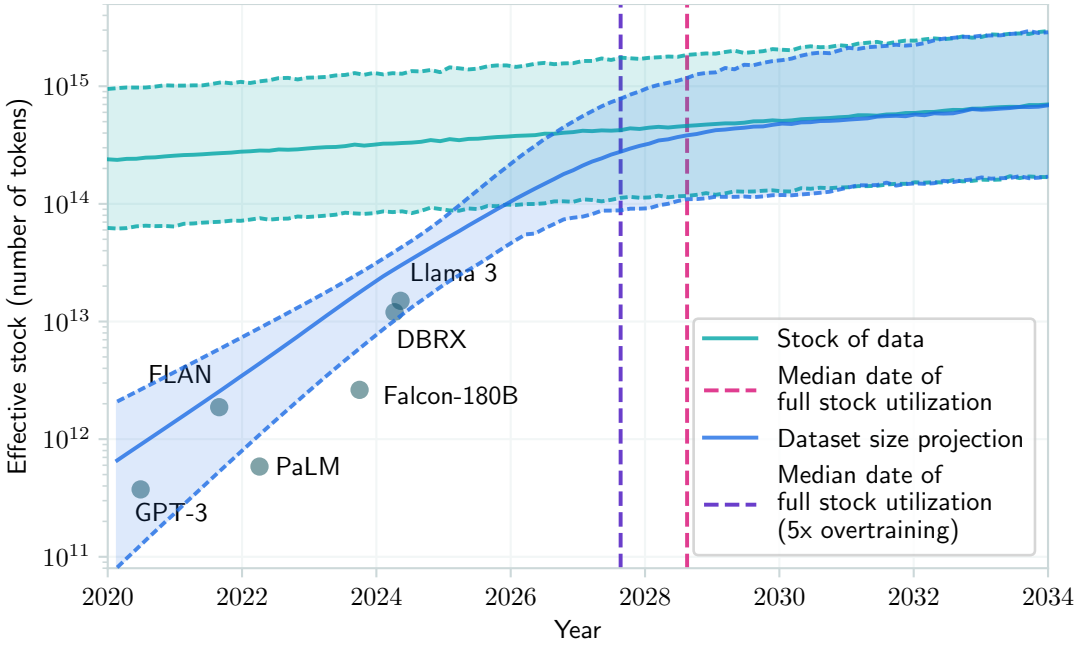

Figure 2: Projection of the amount of text data available to train LLM's from Villalobos et al. (2022)

.

RL agents are able to optimise by taking actions in an environment and receiving a set of rewards and punishments based on these decisions. This mimics how humans achieve high-level performance in novel and unfamiliar domains through evaluation and feedback. Achieving this level of generalised aptitude in a machine learning model is still a distant goal, but research continues to push the boundary of network generalisability. For example, in 2020, MuZero advanced the state of the art by achieving superhuman performance on "a range of challenging and visually complex domains, without any knowledge of their underlying dynamics"(Schrittwieser et al., 2020). The lack of reliance on preexisting knowledge of a domain's underlying dynamics is of particular interest, as it allows the agent to develop novel strategies not yet known to human players. Furthermore, supervised learning models such as LLMs aim to predict tokens (with the addition of self-supervision) based on available data. As the available training data is human-made, and thus limited to human ability,

it can become challenging to push models to exceed human performance. For RL models however, no such limitation exists as the models require no human examples to draw from. For instance, Shinn et al. (2024) surpassed GPT-4 on the HumanEval coding benchmark by using reinforcement feedback to improve the model's output.

One of the main ways MuZero was evaluated was by playing Atari games. Notably, Atari 2600 games are possibly the most popular benchmark for RL models, having been used to evaluate not just MuZero but also some of the most influential and widely cited RL advancements, such as: "PPO Algorithms" (Schaul, 2015), "Asynchronous Methods for Deep Reinforcement Learning" (Mnih, 2016), "Prioritised Experience Replay" (Schaul, 2015), "Human-level control through deep reinforcement learning" (Mnih et al., 2015) and "Deep Reinforcement Learning with Double Q-Learning" (Van Hasselt et al., 2016). This has been facilitated by the development of the Arcade Learning Environment (ALE) which uses Gymnasium and the Atari 2600 emulator Stella to provide a set of game environments for evaluating and training RL models (Bellemare et al., 2013).

The value of ALE as a benchmarking tool stems from serval factors, as outlined in the 2013 paper. First is variety. With over 500 games, spanning multiple genres, the platform has an expansive and diverse range of environments to be tested and trained on. While the environments are varied, they are restricted in scope by the limitations of the hardware, creating a more manageable evaluation domain. Secondly, the games have been created by independent parties which removes the potential for experimental bias. The last benefit they identify is that each domain is "interesting enough to be representative of settings that might be faced in practice".

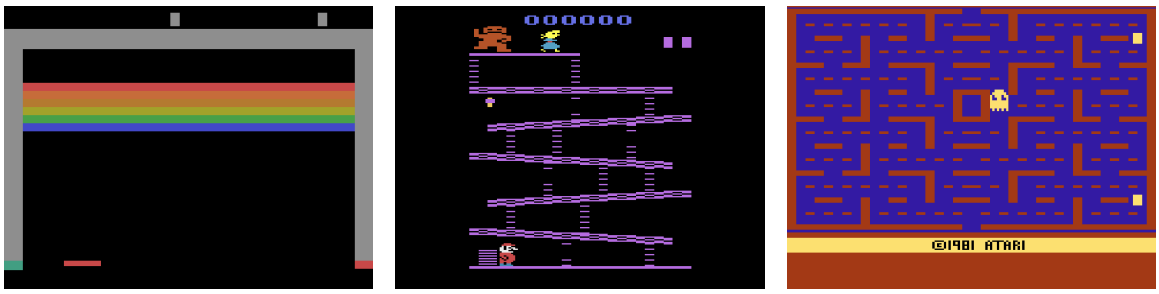

Figure 3: A selection of popular Atari 2600 games. From left to right: Breakout, Donkey Kong and Pac-Man.

Given the utility and prevalence of video game based environments it is worth noting that the 1977 Atari 2600 is an extremely underpowered console compared to what RL agents are currently capable of. For instance, Firoiu et al. (2017) created "Philip" (and its successor "Phillip II" (Firoiu et al., 2020)), a model capable of beating the worlds best players at the highly competitive *Super Smash Bros. Melee*, a fully 3D GameCube game from 2001. Other works such as "OpenAI Five" have achieved superhuman performance at modern competitive games like Dota 2 (Berner et al., 2019). While these are less general agents, it demonstrates the capability and interest in tackling more advanced games.

To bridge this gap I present Gym**NES**ium, a Gymnasium environment for training and evaluating models on Nintendo Entertainment System (NES)[4] games. While the full scope of the project aims to implement many NES titles, this initial release focuses on testing the environment with bleeding edge RL architecture applied to a single title: "Mike Tyson's Punch Out!!" (MTPO)[5]. I then demonstrate how modern RL optimisations enable human-level gameplay by training on desktop hardware, even with limited walltime.

4. The console was originally released under the name "Famicom" in Japan however, I will refer to it as the "NES" exclusively.
5. In 1990/91 after the contract to use Tyson's name and likeness expired, the game was rebranded to "Punch Out!!". I use the terms Mike Tyson's Punch Out!!, MTPO and Punch Out interchangeably throughout.

2 NES vs Atari

The Atari 2600, released in 1977, has a 1.19Mhz CPU, 128 bytes of RAM and outputs a 192x160 image with a 128-colour palette. The full catalog of officially licensed and released games consists of 470 titles, with most ROMs being ∼4KB (although 1990s "Fatal Run" managed to reach 32KB). The iconic CX40 controller has only 18 possible inputs. 4 cardinal joystick directions, 4 corner directions (e.g. up+left) as well as the, impossible to reach but still technically valid, inputs of up+down and left+right, and finally the red button next to the stick.

Comparatively, the NES has a 1.66 MHz, 2 KB of general purpose RAM plus 2 KB of video RAM and outputs a 256x240 image with 256 unique colours. The NES sports 1376 officially licensed games which vary greatly in ROM size, from Balloon Fight and Super Mario Bros. at 25 and 41KB respectively, to Mega Man 4 and Earthbound at 524KB. Finally, the controller has 8 buttons and no limitations on what can be pressed simultaneously (although you may have to bend some plastic to press left and right at the same time) giving 256 possible combinations.

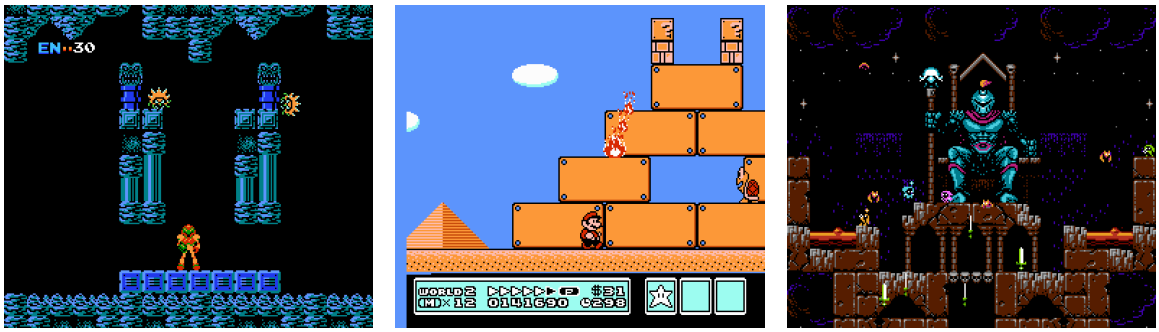

Figure 4: A selection of popular NES games. From left to right: Metroid, Super Mario Bros. 3 and Micro Mages.

The technical differences are summarised in the table below:

	Atari	NES
Release Year	1977	1983
CPU Clock	1.19MHz	1.66MHz
RAM Capacity	128 Bytes	2000KB+2000KB
Resolution	192x160	256x240
Colour Depth	128	256
Catalogue Size	470	1376
Game Size	4KB	25 to 524KB
Action Space	18	256

Figure 5: Atari 2600 and NES hardware comparison.

While this represents a meaningful step up in complexity it is not so significant that it infeasible to anticipate modern RL agents achieving some level of generality. In support

of this, I demonstrate how a moderately powered consumer desktop can train agents to high-level performance in Mike Tyson's Punch Out!! in around 48 hours.

3 Mike Tyson's Punch Out!!

Mike Tyson's Punch Out!! (MTPO) is a 1987 NES title where you play as protagonist "Little Mac" as he attempts to climb from the minor leagues of boxing up to beating Mike Tyson himself. The game consists of 14 unique fights against 10 unique opponents, divided into three circuits:

MTOP Circuits and Opponents

Minor Circuit:
- Glass Joe
- Von Kaiser
- Piston Honda (1)

Major Circuit:
- Don Flamenco (1)
- King Hippo
- Great Tiger
- Bald Bull (1)

World Circuit:
- Piston Honda (2)
- Soda Popinski
- Bald Bull (2)
- Don Flamenco (2)
- Mr. Sandman
- Super Macho Man

The Dream Fight:
- Mike Tyson

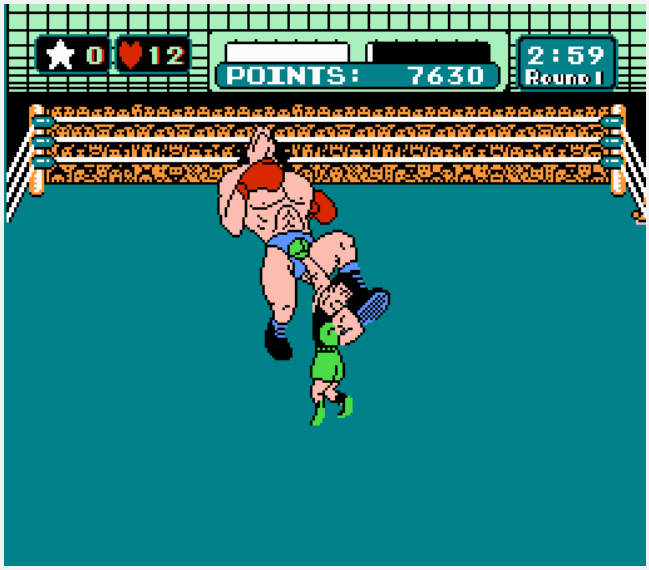

Figure 6: Screenshot from MTPO of the fight against "Super Macho Man".

4 Creating the Environment

4.1 Mathematical Formulation

To facilitate training a diverse set of RL agents, the environment is formulated as a Markov Decision Process (MDP) defined by the tuple (S_t, x_t, S^M, r), where S is the state, $\mathcal{X}$ is the set of decisions/actions, $S^M(S_t, x_t)$ is the transition function, and $R(S_t, x_t)$ is the reward function. The agent's objective is to learn a policy $X^\pi(S_t))$ that maximises the expected sum of discounted rewards $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t(S_t, x_t)]$, where $\gamma \in [0, 1)$ is the discount rate.

4.1.1 Game State

The state variable at time t (labeled S_t) encapsulates all the information about the current game position available to the model. Many RL techniques make use of RAM values to define the state. For example, Tom Murphy VII's generalised lexicographic ordering technique uses the entirety of the NES's general purpose RAM as it is only 2048 bytes (Murphy VII, 2013). A more sophisticated approach to defining the state with RAM values is used by Firoiu et al. (2017), where addresses which store important data, such as character positions or action states, are identified and read to form the state. Some of Punch Out's key memory values are shown in figure 7 below.

Label	Address	Symbol
Fight ID	`0x0008`	f
Opponent ID	`0x0001`	o
Opponent Type ID	`0x0002`	t
Current Round	`0x0006`	r
Clock (3 bytes)	`0x0305`	s
Mac's Health	`0x0391`	h_M
Opponent's Current Health	`0x0399`	h_O
Star Punches	`0x0342`	sp
Energy Level	`0x0392`	e
Hearts (two bytes)	`0x0323`	h_{10}
RNG (updates once per frame)	`0x0018`	rng
Counter (0 to 255)	`0x001E`	c
Opp Fight Pattern Init	`0x0030`	p_{init}
Opp Fight Pattern Timer	`0x0039`	p_t
Opp Actions ID	`0x003A`	a_O
Global Variable for Enemy Actions	`0x003B`	a_{global}

Figure 7: Notable RAM values in MTPO.

Thus the state variable could be written as a vector of these 8-bit values:

$$S_t = \left[f, o, t, r, s, h_M, h_O, sp, e, h_{10}, \text{rng}, c, p_{\text{init}}, p_t, a_O, a_{\text{global}}\right] \tag{1}$$

While this approach is effective in reducing the input space, it suffers from two mayor weaknesses. First is a lack of generality; locating and specifying the particular addresses for a given game is a lengthy and often tedious task which involves a human identifying which data are deemed important. This can be mitigated by supplying the entire contents of RAM however, this then eliminates the benefit of reducing the state space. The other limitation is that while the contents of RAM are heavily correlated with the game environment, they are not identical. There is information encoded in a game's sprites and palettes which cannot be viewed in RAM, such as an enemy telegraphing an attack. Furthermore, other information which is present in RAM is intentionally hidden from a player and not displayed on screen, such as the state of the games random number variable. A more general approach which

better matches the human play experience, is to take each full frame of video output as a matrix with dimensions:

$$\textit{number of colour channels}\ (c) \times \textit{horizontal resolution}\ (size_h) \times \textit{vertical resolution}\ (size_v)$$

Giving a state variable: $S_t = \mathbf{F}$. Here, $\mathbf{F} \in \mathbb{R}^{c \times size_h \times size_v}$ is the full frame tensor returned at each step of the emulator. However, the full resolution, unprocessed, frame tensor is usually not required to achieve optimal performance. Thus, for the Punch Out environment several filters are applied. First, the image is grayscaled using OpenCV's cvtColor function. Next the observation is cropped. Cropping is useful because the key information required to play Punch Out is contained in the small region of the screen showing the opponent and the player. Opponents rarely leave this region, and when they do it is for very specific attacks which can be recognised from the opponent's absence. I also implemented downscaling for the observation, but ultimately chose not to utilise it in favour of the aggressive crop.

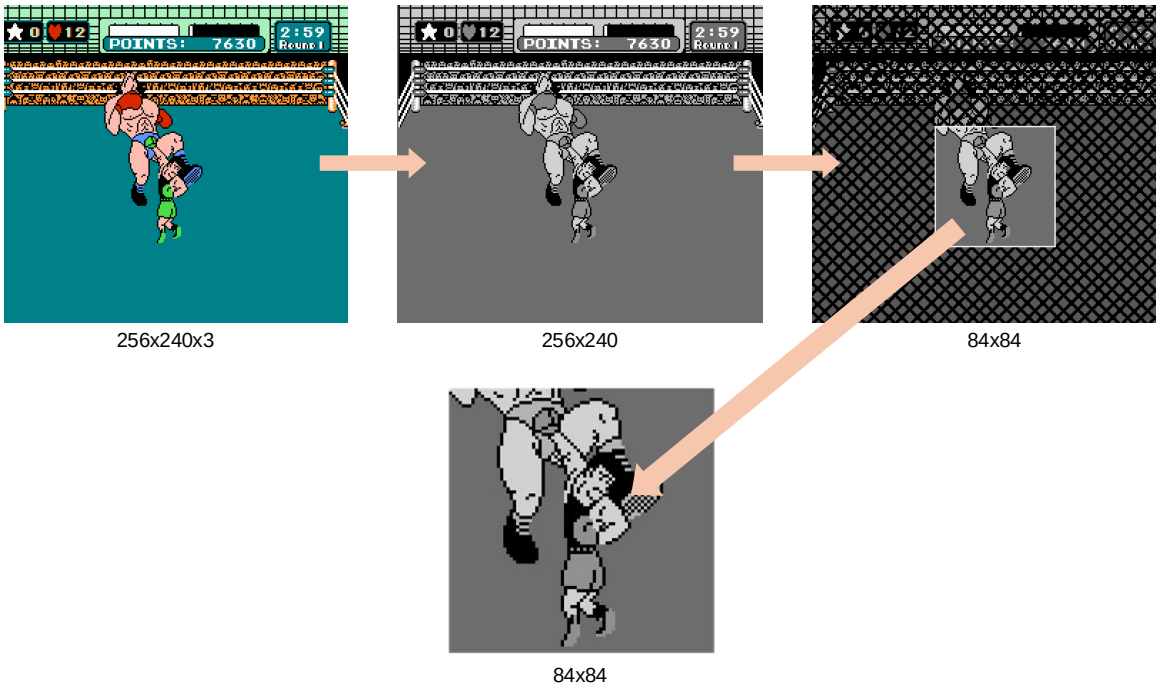

Figure 8: Visual illustration of how the observation is transformed.

Finally, 4 of these 84x84 processed frame tensors are stacked together using Gymasium's FrameStackObservation wrapper, to form a single observation. This allows the network to learn temporal relationships between frames and track important information such as moving objects. This gives a final state of:

$$S_t = \mathbf{O} \tag{2}$$

Where $\mathbf{O} \in \mathbb{R}^{4 \times 84 \times 84}$ is the processed observation.

4.1.2 Decision

The NES controller has 8 buttons, giving a total possible decision space of 256 unique inputs. However, the vast majority of these combinations do not produce unique action in most games. The decision space can be greatly reduced by instead mapping unique in-game actions to button combinations. For instance, in MTPO, the high-level actions available on each frame are:

	In-Game Action	Button	Controller Value
0	No action	-	00000000_2
1	Right punch	A	10000000_2
2	Left punch	B	01000000_2
3	Right uppercut	Up + B	10001000_2
4	Left uppercut	Up + A	01001000_2
5	Dodge left	D-Pad Left	00000010_2
6	Dodge right	D-Pad Right	00000001_2
7	Block	D-Pad Down	10000100_2
8	Star punch	Start	00010000_2

Figure 9: Possible actions/decision in MTPO.

This constrains the decision space to:

$$\mathcal{X} = \{0, 1, 2 \dots 8\} \tag{3}$$

I further reduce the number of actions with symmetry; right punch, right uppercut and right dodge are removed from the input space as they do not meaningfully differ from the left handed equivalents. This leaves only 6 actions for the agent to select from. At each step the agent selects a decision x_t following the requirement:

$$x_t \in \mathcal{X}, \quad \text{where } \mathcal{X} \in \{0, 1, 2, 3, 4, 5\} \tag{4}$$

Indexes 0 to 5 then map to the following controller values which are passed to the emulator: $\{0 : 00000000_2, 1 : 01000000_2, 2 : 01001000_2, 3 : 00000010_2, 4 : 00000100_2, 5 : 00010000_2\}$

4.2 Exogenous Information

The primary source of uncertainty in the model comes from the randomised actions of the in-game opponents. The most extensive source of information on MTPO's random number generation (RNG) function comes from the speedrunning community. In the interest of developing consistent strategies for tackling one of the game's most random opponents: 'King Hippo", users "Lucandor158" and "Zoxsox" conducted a thorough investigation into how the game determines random actions (Lucandor158 and Zoxsox, 2019). Based on this investigation they were able to develop a 100% consistent strategy for manipulating King Hippo, aptly named the "Hippo Manipo". MTPO selects the opponent's action based on an 8-bit random value at address 0x0018 which is updated every frame based on the bytes at 0x0019 and 0x001E.[6]

6. The relevant 6502 disassembly of the RNG function is available in appendix A.

The data at 0x001E stores an 8-bit frame counter, which is incremented each frame and loops back to 0 once the value exceeds 255. When updating RNG, the frame counter is loaded and the bits are shuffled to reduce the frame-to-frame consistency and predictability of the RNG. Next, the byte at 0x0019, which stores the current controller values (as shown in figure 14) is loaded and shuffled. Given that the RL agent will make occasional random decisions, using an epsilon-greedy approach, the RNG values will diverge as soon as any random input is made. From there even identical inputs will result in wildly differing opponent action. This means that the agent will be unable to predict the actions the opponent will take and will have to learn to adapt and respond to the uncertainty. Additionally, while the RNG value will rapidly diverge, I also set the RNG byte at 0x0018 to a random initial value each time the environment is reset to avoid early predictability.

4.2.1 Policy

The agent is trained using Q-learning, where a numerical "Q-value" is assigned to state-action pairs. When the network trains, it learns to approximate this Q function by sampling from its prior experiences, computing the gradient of the difference between prediction and reality, and then using backpropagation to update all parameters in the network. Once a the network has been fully trained, the policy simply picks the action x_t which maximises the expected reward, i.e. the action with the highest Q value for the current state S_t. This gives the policy:

$$X^{\pi}(S_t) = \underset{x_t \in \mathcal{X}_t}{\operatorname{argmax}} Q(S_t, x_t) \tag{5}$$

Where Q is the function approximated by the network. However, during training, two concurrent networks exist, one target network Q_{target} and one live network Q_{live}. The live network is updated at every gradient step and is used for selecting actions, while the target network is used to predict future rewards and is periodically updated to match the live network. Using a separate networks for estimating future rewards gives the live network a more stable target to aim for and improves the stability of training (Mnih et al., 2015). Furthermore, during training, the decision (or action) is chosen using an epsilon-greedy policy given by:

$$X^{\pi}(S_t) = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \underset{x_t \in \mathcal{X}_t}{\operatorname{argmax}} Q_{live}(S_t, x_t), & \text{with probability } 1 - \epsilon \end{cases} \tag{6}$$

Where $\epsilon \in [0, 1]$. Taking intermittent random actions ensures that the network balances between exploration and exploitation and ϵ is deceased throughout training to adjust this balance.

4.2.2 Reward Function

The contribution or reward function R determines the value of a particular decision in a state. For the Boxing Gym environment, the following information is used to define the reward:

$\Delta O_{id} =$ The change in the opponent ID value. Each fight has a unique ID ranging from 0-13.

$\Delta D =$ The change in the number of times the opponent has been knocked down.

$\Delta c =$ The change in the number of seconds on the clock

$\Delta H_{opp} =$ The change in the opponent's health.

$\Delta H_{mac} =$ The change in the player (Mac)'s health.

These are used to create the reward function:

$$R(S_t, x_t) = 15\Delta O_{id} + 2\Delta D + \Delta H_{opp} - 0.1\Delta c - \Delta H_{mac} \tag{7}$$

4.2.3 Objective Function

The Q function follows the Bellman equation and thus the network should learn to approximate:

$$Q_{live}(S_t, x_t) = R(S_t, x_t) + \gamma \max_{x_{t+1}} Q_{target}(S_{t+1}, x_{t+1}) \tag{8}$$

Where $\gamma \in [0, 1)$ is the discount factor. The discount factor reduces the value of predicted future rewards to ensure the policy favours imitate rewards (which are certain) over similarly sized future rewards. If γ is too high the agent may take poor actions, believing that they will give future rewards which never materialise, whereas if γ is too low, the agent will become short sighted and fail to learn actions that have better long-term value.

Therefore, the objective function aims to maximise the expected sum of discounted future rewards, giving:

$$\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} R(S_t, x_t) + \gamma \max_{x_{t+1}} Q_{target}(S_{t+1}, x_{t+1}\right] \tag{9}$$

5 Code

Existing work has been completed to model the NES as a reinforcement learning problem. Kauten et al. (2018) developed *nes-py*; a Python-based NES emulator and Gym interface, with the addition of a Gym environment for Super Mario Bros.(Kauten et al., 2017). Unfortunately, this work has some significant drawbacks upon which GymNESium improves. First, nes-py has not seen any major updates in several years and is only compatible with, the now deprecated, Gym. More critically however, the emulator backend (SimpleNES) only supports a subset of cartridge configurations (referred to as mappers) resulting in many games (MTPO included) being unable to run at all. Given this limitation, I instead chose to create a Gymnasium environment built upon the *cynes* emulator (Combey, 2024). The Gymnasium interface, GymNESium, implements a NESEnv class as a child of the Gymnasium environment class.

5.0.1 GymNESium

GymNESium is a fairly bare-bones abstraction layer which performs some checks on a ROM, loads the emulator and implements a NESEnv class. NESEnv provides some general-purpose methods such as those for saving and loading states and resetting the environment, as well as some abstract methods to be implemented for each specific game environment. This allows Gymnasium environments for any game to be set up quickly by inheriting from this class and implementing the abstract functions which then allows for straightforward training.

5.0.2 Boxing Gym

The MTPO environment (PunchOutEnv) in turn inherits from the NESEnv and contains the specifics for the Punch Out interface. The full code for this environment is available here: *Boxing Gym*. Critically, my initial attempts to train a RL agent suffered from significant limitations in the speed of execution. Based on the data gathered in these attempts, even with a highly optimised network architecture, tens of millions of frames of training are required to produce high level play. This is a particular challenge as emulation speed can easily be bottlenecked by single-threaded performance, even while overall CPU utilisation remains low. To ensure efficient use of resources, Boxing Gym leverages vectorisation to allow multiple parallel environments to be executed to train a single model.

5.1 Transition Function

In a Gymnasium environment the transition function is implemented by overriding the abstract “step" method from the gym.Env class. The transition function S^M takes the current state S_t, a given action x_t, and the exogenous information W_{t+1} and returns the new state. Generating randomness and transitioning the state is primarily handled by the emulator therefore Boxing Gym's step function advances the emulator a single frame with the chosen action x_t and returns the next frame as the observation.

The transition function also checks if a state is terminal and resets the environment if so. A state is considered terminal if the player (Mac) took any damage/got punched. This discourages the agent from taking actions which may result in getting hit, as at that point it is no longer able to gain any additional reward from continuing to play. The

small time penalty was included to discourage inaction or excessive dodging/blocking. The time penalty was deliberately set low so that continued play would almost always result in increasing rewards, as otherwise the agent will be incentivised to end the timer early by getting hit and resetting the environment. This proved effective and the agent quickly learned to land punches and dodge swings. However, one limitation of this approach is that more hits and more knockdowns can be gained by dragging out the fight for longer. This resulted in the agent prioritising technical knockouts (TKOs) where the player wins be default after 3 knockdowns. There are faster strategies, which involve well timed punches, that can get a KO much soon however, the agent deliberately avoided these in favour of the larger reward earned from a TKO. To tackle this, a second training run was performed where the reward function was updated to much more heavily penalise time, thus making a fast ending to the fight preferable. To avoid the possibility of the agent attempting to deliberately get hit to avoid a time penalty, the environment was updated to only reset at the end of a round. That way the only strategy to reduce the time penalty was to KO the opponent sooner.

5.2 Network Architecture

To test the environment the agent is implemented by directly utilising the *Beyond The Rainbow* network architecture from Clark et al. (2024), which combines each of the following innovations: Prioritised Experience Replay, N-Step, Distributional RL, Dueling DQN, Double DQN (Van Hasselt et al., 2016), Impala Architecture + Adaptive Maxpooling, Spectral Normalisation (SN), Implicit Quantile, Networks (IQN), Munchausen RL and Vectorisation. The network consists of several convolution blocks, each followed by a rectified linear unit (ReLU) activation layer. The diagrams below were created to illustrate the layers of the network.

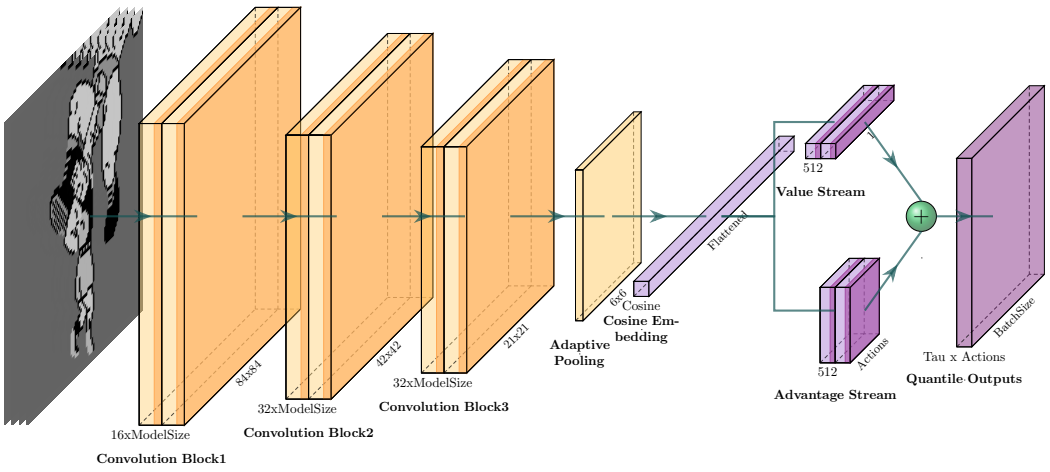

Figure 10: High level visualisation of the network layers.

Figure 10 above shows how the main blocks of the network fit together, as well as how duelling DQN(Wang et al., 2016) is used by splitting the network into an advantage and a

value stream. Each one of the convolution blocks contain several convolution layers, each made up of 32 channels. The below figure shows the steps involved in convolution block 1.

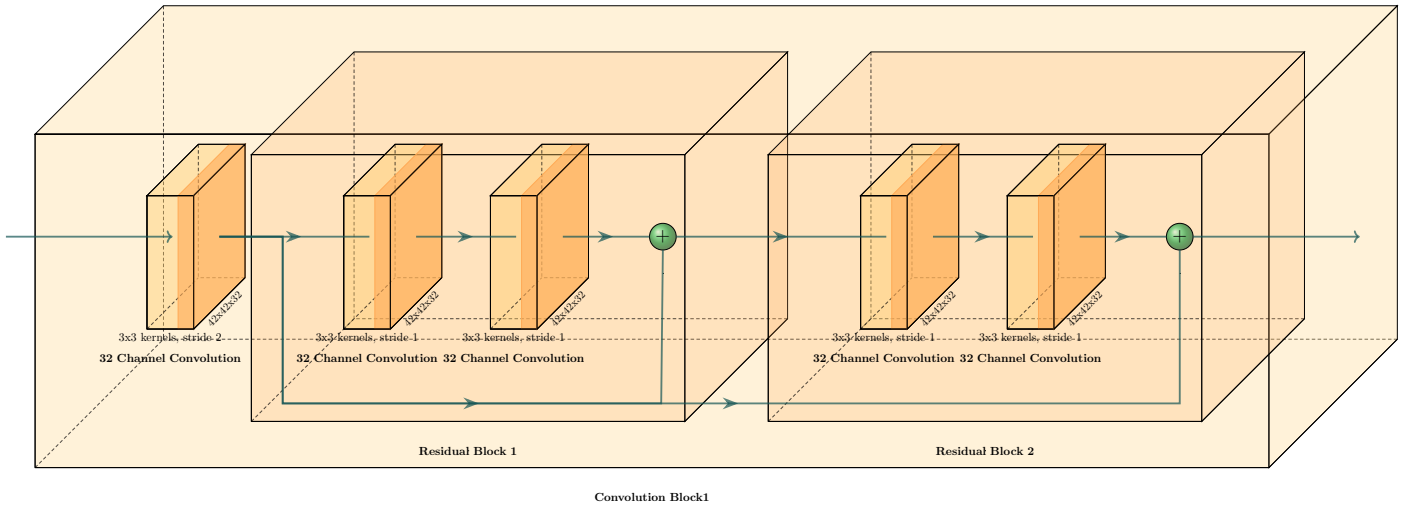

Figure 11: Visualisation of convolution block 1.

Note that the first 32 channel convolution has a stride of 2, which reduces the dimensions of the observation from 84x84 to 42x42 while the 32 channels change the dimensions to 42x42x32. The darker orange band after each convolution indicates a ReLU activation layer which sets any negative values in the convolution kennels to 0. The green plusses show where “shortcut" layers are used, which combine the output of the first convolution layer back into the outputs of each residual block.

6 Training

The network was trained on a desktop running an AMD Ryzen 5 5500, RTX 2080 Super and 32GB of RAM for 48 hours, running 64 parallel environments. The hyperparamters used during training are listed in appendix B. The training lasted for 13,743,232 steps over 7800 episodes

7 Evaluation

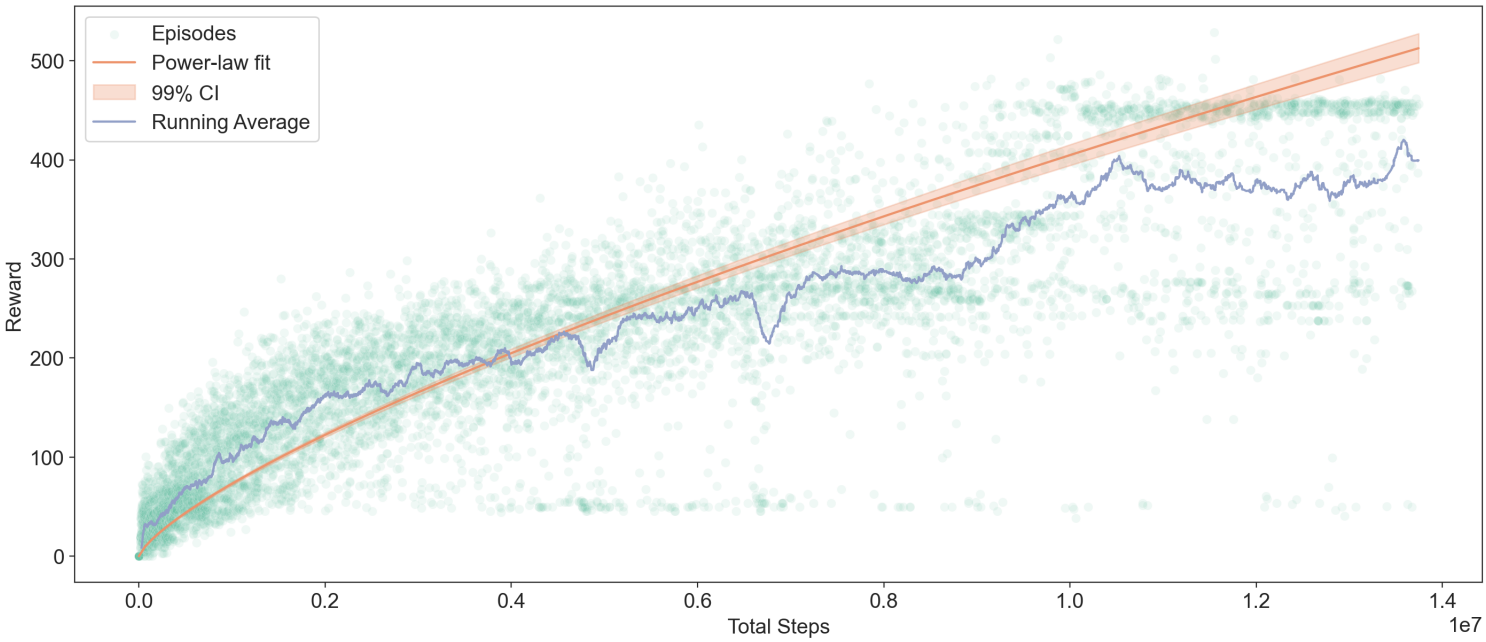

Figure 12: Total reward for each episode plotted against the number of total environment steps. The orange line shows a power law fit with a log-space R-squared value of 0.762 and the blue line shows a running average of the previous 100 rewards.

Each data point in figure 13 represents one complete episode, i.e. a full play through from the start of the first fight, up to when agent got hit, and the environment reset. The total steps on the x-axis show how many gradient steps had been taken up to that point in training. Over the course of the training the agent achieved single-episode reward of 528.5 on episode 7177 with the last 100 episodes having a mean reward of 399.5. The strength of the BTR architecture is evident in how consistently and rapidly the reward increases as the network is optimised. Given the trend at the time training finished, it is hard to assess if the agent would have continued to advance if given more time. A comparison worth mentioning however, is that the BTR authors showed increasing performance up to 200 million steps (compared to the 13 million used here) which suggests the rewards would have continued to rise.

Notably, the graph also shows stratification, with distinct horizontal lines of consistent reward. Running samples of the network at these points showed that these lines match up with when the agents reaches a new, unfamiliar, and more challenging opponent and quickly gets hit. These sharp difficulty spikes end many episodes, causing them to end with a similar reward. The below graph shows the reward achieved by an agent reaching, and then losing to, each opponent.

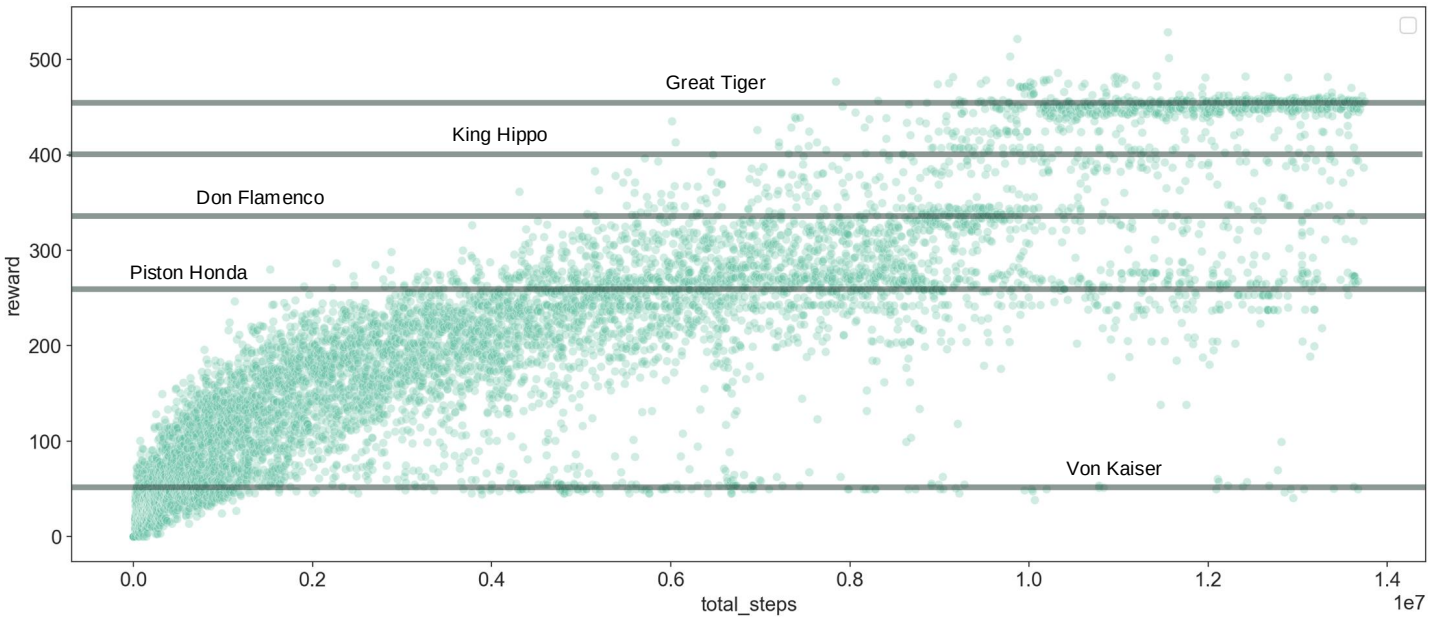

Figure 13: Total episode reward plotted against the number of total environment steps with lines to show approximately where the reward limit is if the agent is not able to beat a particular opponent.

The large number of episodes which terminate at “Great Tiger" cause the largest barrier however, a few episodes find success in progressing the fight.

As noted, this reward structure incentivises drawn out fights won by TKO. To encourage the agent to play more aggressively and finish fights sooner a second agent was trained with a new set of rewards. Given the change in rewards this second model cannot be directly compared with the first however, the second training process terminated early due to a hardware fault and was unable to beat “King Hippo". Despite this, the agent was able to constantly beat the first 4 opponents much more quickly than the first, and as such the model is included in the release for comparison.

8 Conclusion

The idea of an NES-based RL evolution environment shows promise with even low-compute, low-walltime training producing high-level results. Given this potential, further work is required to implement a more diverse set of game domains and a more produce a more thorough set of evaluations. Compared to the Atari lineup, the NES features many games with broader scope and a less clear definition of success than a simple score. While these present challenges they are also more representative of many real-world problems and so present an interesting benchmark for future models. With a larger set of game environments the generality of BTR, as well as other network architectures could be measured, with comparison between performance in ALE and GymNESium being of particular interest. As well broadening the set of domains, longer duration training would allow the full capability of BTR to be explored, as, while Great Tiger was ending many runs, the mean reward had

not yet plateaued. Developing a single agent capable of beating all 14 fights was an early goal which may require an increase to the scale of the network however, further testing is required.

Appendix A. RNG Disassembly

```
 Address   Hex       Disassembly      Comment
85E9      A5 1E      LDA $1E          ; Load frame counter
85EB      6A         ROR A            ; Rotate right
85EC      85 E8      STA $E8          ; Store result in $E8
85EE      6A         ROR A            ; Rotate again
85EF      85 E9      STA $E9          ; Store result in $E9
85F1      6A         ROR A            ; Rotate again
85F2      85 EA      STA $EA          ; Store result in $EA
85F4      6A         ROR A            ; Rotate again
85F5      29 08      AND #$08         ; Isolate bit 3
85F7      85 ED      STA $ED          ; Store intermediate value in $ED
85F9      A5 E8      LDA $E8          ; Load $E8
85FB      29 01      AND #$01         ; Isolate bit 0
85FD      05 ED      ORA $ED          ; Combine with $ED
85FF      85 ED      STA $ED          ; Store result in $ED
8601      A5 E9      LDA $E9          ; Load $E9
8603      29 92      AND #$92         ; Isolate bits 1, 4, and 7
8605      05 ED      ORA $ED          ; Combine with $ED
8607      85 ED      STA $ED          ; Store result in $ED
8609      A5 EA      LDA $EA          ; Load $EA
860B      29 04      AND #$04         ; Isolate bit 2
860D      05 ED      ORA $ED          ; Combine with $ED
860F      85 ED      STA $ED          ; Store result in $ED
8611      A5 1E      LDA $1E          ; Reload frame counter
8613      2A         ROL A            ; Rotate left
8614      85 E8      STA $E8          ; Store result in $E8
8616      2A         ROL A            ; Rotate again
8617      2A         ROL A            ; Rotate again
8618      2A         ROL A            ; Rotate again
8619      29 40      AND #$40         ; Isolate bit 6
861B      05 ED      ORA $ED          ; Combine with $ED
861D      85 ED      STA $ED          ; Store result in $ED
```

The data at 0x001E stores an 8 bit frame counter which is incremented each frame and loops back to 0 once the value exceeds 255. Here the frame counter is loaded and then the bits are shuffled to reduce the frame-to-frame consistency, and thus predictability of the RNG value. Next the byte at 0x0019 which stores the current controller values (as shown in figure 7) is loaded and shuffled:

```
 Address   Hex       Disassembly      Comment
861F   A5 E8          LDA $E8          ; Load $E8
8621   29 20          AND #$20         ; Isolate bit 5 of $E8
8623   05 ED          ORA $ED          ; Combine with the value in $ED
8625   85 ED          STA $ED          ; Store the result back into $ED
;Mix a specific bit (bit 5) of $E8 into $ED

```

```
8627  A5 D0          LDA $D0          ; Load $D0
8629  6A             ROR A            ; Rotate right
862A  85 E8          STA $E8          ; Store the result in $E8
862C  6A             ROR A            ; Rotate accumulator right again
862D  85 E9          STA $E9          ; Store the result in $E9
862F  6A             ROR A            ; Rotate right again
8630  29 98          AND #$98         ; Isolate bits 3, 4, and 7
8632  85 EC          STA $EC          ; Store the result in $EC
;Shuffle and isolate specific bits from $D0 (source of randomness)

8634  A5 D0          LDA $D0          ; Reload $D0
8636  29 01          AND #$01         ; Isolate bit 0 of $D0
8638  05 EC          ORA $EC          ; Combine with the value in $EC
863A  85 EC          STA $EC          ; Store the result back into $EC
;Add the least significant bit (LSB) of $D0 into $EC

863C  A5 E8          LDA $E8          ; Load $E8
863E  29 02          AND #$02         ; Isolate bit 1 of $E8
8640  05 EC          ORA $EC          ; Combine with the value in $EC
8642  85 EC          STA $EC          ; Store the result back into $EC
8644  A5 E9          LDA $E9          ; Load $E9
8646  29 04          AND #$04         ; Isolate bit 2 of $E9
8648  05 EC          ORA $EC          ; Combine with the value in $EC
864A  85 EC          STA $EC          ; Store the result into $EC
;Add bit 1 of $E8 and bit 2 of $E9 into $EC.

864C  A5 D0          LDA $D0          ; Reload $D0
864E  2A             ROL A            ; Rotate left
864F  85 E8          STA $E8          ; Store the result in $E8
8651  2A             ROL A            ; Rotate left again
8652  29 20          AND #$20         ; Isolate bit 5 of the accumulator
8654  05 EC          ORA $EC          ; Combine with the value in $EC
8656  85 EC          STA $EC          ; Store the result back into $EC
;shift 0x00D0s bits left, and bit 5 is added to 0x00EC.

8658  A5 E8          LDA $E8          ; Load  $E8
865A  29 40          AND #$40         ; Isolate bit 6 of $E8
865C  05 EC          ORA $EC          ; Combine with the value in $EC
865E  18             CLC              ; Clear the carry flag
865F  65 19          ADC $19          ; Add $19 to accumulator with carry
8661  85 19          STA $19          ; Store the result back into $19

8663  6A             ROR A            ; Rotate accumulator right
8664  85 E8          STA $E8          ; Store the result in $E8
8666  6A             ROR A            ; Rotate accumulator right again
8667  85 E9          STA $E9          ; Store the result in $E9
8669  6A             ROR A            ; Rotate accumulator right again
866A  29 02          AND #$02         ; Isolate bit 1 of the accumulator
866C  85 EC          STA $EC          ; Store the result in $EC

866E  A5 E8          LDA $E8          ; Load $E8
8670  29 20          AND #$20         ; Isolate bit 5 of $E8
8672  05 EC          ORA $EC          ; Combine with the value in $EC
8674  85 EC          STA $EC          ; Store the result back into $EC
8676  A5 E9          LDA $E9          ; Load  $E9
```

```
8678  29 08     AND #$08        ; Isolate bit 3 of $E9
867A  05 EC     ORA $EC         ; Combine with the value in $EC
867C  85 EC     STA $EC         ; Store the result back into $EC

867E  A5 19     LDA $19         ; Load $19
8680  2A        ROL A           ; Rotate accumulator left
8681  85 E8     STA $E8         ; Store the result in $E8
8683  2A        ROL A           ; Rotate accumulator left again
8684  85 E9     STA $E9         ; Store the result in $E9
8686  2A        ROL A           ; Rotate accumulator left again
8687  29 40     AND #$40        ; Isolate bit 6 of the accumulator
8689  05 EC     ORA $EC         ; Combine with the value in $EC
868B  85 EC     STA $EC         ; Store the result back into $EC
868D  A5 E8     LDA $E8         ; Load $E8
868F  29 04     AND #$04        ; Isolate bit 2 of $E8
8691  05 EC     ORA $EC         ; Combine with the value in $EC
8693  85 EC     STA $EC         ; Store the result back into $EC
8695  A5 E9     LDA $E9         ; Load $E9
8697  29 10     AND #$10        ; Isolate bit 4 of $E9
8699  05 EC     ORA $EC         ; Combine with the value in $EC
869B  85 EC     STA $EC         ; Store the result back into $EC

869D  A5 19     LDA $19         ; Load $19
869F  29 81     AND #$81        ; Isolate bits 0 and 7 of $19
86A1  05 EC     ORA $EC         ; Combine with the value in $EC
86A3  18        CLC             ; Clear the carry flag
86A4  65 ED     ADC $ED         ; Add $ED to accumulator with carry
86A6  85 18     STA $18         ; Store the final RNG value in $18
86A8  60        RTS             ; Return from subroutine
```

Appendix B. Hyperparameters

The hyperparamters used are identical to those outlined in Clark et al. (2024) shown in the table below.

Hyperparameter	Value
Learning Rate	$1e-4$
Discount Rate	0.997
N-Step	3
IQN Taus	8
IQN Number Cos	64
Huber Loss κ	1.0
Gradient Clipping Max Norm	10
Parallel Environments	64
Gradient Step	Every 64 Environment Steps (1 Vectorized Env Step)
Replace Target Network Frequency (C)	500 Gradient Steps (32K Environment Steps)
Batch Size	256
Total Replay Ratio	$\frac{1}{64}$
Impala Width Scale	2
Spectral Normalization	All Convolutional Residual Layers
Adaptive Maxpooling Size	6x6
Linear Size (Per Dueling Layer)	512
Noisy Networks σ	0.5
Activation Function	ReLu
ϵ-greedy start	1.0
ϵ-greedy decay	2M Frames
ϵ-greedy end	0.01
ϵ-greedy disabled	100M Frames
Replay Buffer Size	1048576 Transitions (2_{20})
Minimum Replay Size for Sampling	200K Transitions
PER Alpha	0.2
Optimizer	Adam
Adam Epsilon Parameter	1.95e-5 (equal to 0.05 batchsize)
Adam $\beta 1$	0.9
Adam $\beta 2$	0.999
Munchausen Temperature τ	0.03
Munchausen Scaling Term α	0.9
Munchausen Clipping Value (l0)	-1.0
Evaluation Epsilon	0.01 until 125M frames - then 0
Evaluation Episodes	100
Evaluation Every	1M Environment Frames (250K Environment Steps)

Figure 14: Training hyperparameters.

References

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.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019. URL `http://arxiv.org/abs/1912.06680`.

Tyler Clark, Mark Towers, Christine Evers, and Jonathon Hare. Beyond the rainbow: High performance deep reinforcement learning on a desktop pc, 2024. URL `https://arxiv.org/abs/2411.03820`.

Théo Combey. cynes - c/c++ nes emulator with python bindings, 2024. URL `https://github.com/Youlixx/cynes`.

Vlad Firoiu, William F. Whitney, and Joshua B. Tenenbaum. Beating the world's best at super smash bros. with deep reinforcement learning. 2017. URL `https://arxiv.org/abs/1702.06230`.

Vlad Firoiu, Max Shen, and Kyle Darling. Slippi-ai (phillip ii). 2020. URL `https://github.com/vladfi1/slippi-ai`.

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. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Christian Kauten, Luke Wood, Robert Clark, Elias Hasle, and Jun Jet T. gym-super-mario-bros: An openai gym environment for super mario bros, 2017. URL `https://github.com/Kautenja/gym-super-mario-bros`.

Christian Kauten, Lucas Schönhold, fo40225, Aymeric Bianco Pelle, and Devan Mallory. nes-py: Python3 nes emulator and openai gym interface, 2018. URL `https://github.com/Kautenja/nes-py`.

Lucandor158 and Zoxsox. Hippo rng. `https://docs.google.com/document/d/1haVqEMyMtdr-zXUzGHcERdyJAXHC0O08hAodKA7n-5I/edit?tab=t.0`, 2019. Accessed on January 2, 2025.

Volodymyr Mnih. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Tom Murphy VII. The first level of super mario bros. is easy with lexicographic orderings and time travel . . . after that it gets a little tricky. 2013.

Tom Schaul. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. Will we run out of data? limits of LLM scaling based on human-generated data. arXiv preprint arXiv:2211.04325, 2022. Accessed on.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.