Mastering Super Mario Bros with RainbowDQN

Jake Davies
Department of Computer Science
University of Bath
Bath, BA2 7AY
jd2229@bath.ac.uk

Shivam Sethi
Department of Computer Science
University of Bath
Bath, BA2 7AY
ss3618@bath.ac.uk

Benaiah Mutinda
Department of Computer Science
University of Bath
Bath, BA2 7AY
bm724@bath.ac.uk

Antony Rogers
Department of Computer Science
University of Bath
Bath, BA2 7AY
asr70@bath.ac.uk

Ethan Mansbridge
Department of Computer Science
University of Bath
Bath, BA2 7AY
ejm99@bath.ac.uk

Adam El Kholy
Department of Computer Science
University of Bath
Bath, BA2 7AY
aek57@bath.ac.uk

1 Problem Definition

Super Mario Bros Gym is an environment made publicly available by (Kauten, 2018) designed for compatibility with the OpenAI Gym framework. This project aims to develop an agent capable of completing the levels of the eponymous Nintendo Entertainment System (NES) game by comparing a variety of Reinforcement Learning (RL) algorithms. For this implementation, environment states are either processed frames from the framebuffer of the game emulator or a vector of features (such as Mario's position and score). The reward function provides rewards based on how far rightwards Mario has moved.

The environment has many different forms of the same game available for use, each with its own distinct levels. A large amount of training was conducted on the first level, with Mario being afforded only one life, and the remaining testing and training performed across a variety of levels in order to assess generalisability. One significant consideration was whether to use only a single level, with one life, or to use the full game (i.e. all levels) with three lives. We made use of both variations across different stages to ensure the validity of our results. The full selection of our environment options can be found in **Appendix: A.2.1**.

The reward function, computed after each state transition, is shown by (1), where Δx is only positive when Mario moves rightwards. The *death penalty* is 0 in most states, and -15 in states where Mario has died. The *clock time taken* represents the in game clock time between frames.

$$\Delta x + death\ penalty - clock\ time\ taken \tag{1}$$

This reward function essentially rewards our agent if it moves rightwards quickly, and penalises it for standing still, moving left, or dying. The reward is finally clipped between -15 and 15.

The action space for our agent is defined as all possible meaningful actions that can be made on an NES game controller. This includes doing nothing, pressing 'A', 'B', left, right, up, down, and certain combinations of the above actions. The comprehensive list of available actions can be found in **Appendix: A.2.1**. All actions can be performed in any state.

The transitions between states are deterministic. Mario will always perform the actions from the pressed combination of buttons, unless there is an obstacle in the way. For example, if there is a wall to the right of him, he will not move any further right.

2 Background

Solving problems with significantly large and complex state spaces relies on function approximation, where the agent doesn't learn an exact mapping of all states, but learns to group very similar states together. Deep reinforcement learning is a proven successful method, wherein the function approximator used for the optimal policy is a deep neural network.

The traditional approach used in a standard Deep Q-Network (DQN) has been to use a replay buffer of experiences and sample from them uniformly, reducing the recency bias that can occur when only using recent experiences (Mnih et al., 2013). The traditional DQN has two networks, where one is essentially a copy of the other that is updated far less often to increase stability. Advancements to prevent notable falloff in results were made soon after, with the Double-DQN being presented in (van Hasselt, Guez and Silver, 2015), which counters the overestimation introduced in DQN, offering better results with less falloff.

Research into how a standard DQN could train more efficiently also began soon after its introduction. One of the earlier, more impacting concepts was a Prioritised Experience Replay Buffer (PER), proposed in (Schaul et al., 2016), which instead samples experiences across a probability distribution of assigned priorities. In doing so, the paper demonstrates a 233% increase in the normalised mean metric with a standard DQN and a 131% increase with a DDQN. Performance is outlined to be at its greatest when rewards lack noise, which is the case in our Super Mario Bros environment.

There have been far more advancements made to DQN since, and research has shown that a combination of these improvements can result in an agent that is consistently far more successful on the Atari benchmark, (Hessel et al., 2017). This combined technique, known as RainbowDQN, incorporates:

- Double Deep Q-Network (DDQN) (van Hasselt, Guez and Silver, 2015),
- Prioritised Experience Replay (PER) (Schaul et al., 2016),
- Dueling Network architecture (Wang, de Freitas and Lanctot, 2015),
- Noisy Network architecture (Fortunato et al., 2019),
- Distributional Network architecture (Bellemare, Dabney and Munos, 2017), and
- N-Step learning updates (Sutton, 1988)

(Hessel et al., 2017) have already shown how successful this technique has been across numerous Atari games, and as Super Mario Bros is a very similar game sharing common properties (low resolution, platformer) we believed that RainbowDQN would be a very useful addition to our agent. We incorporate each improvement from RainbowDQN into our agent and assess results.

One of the biggest challenges in reinforcement learning is an agent's ability to learn a policy that generalises over various related environments, such as the levels in our game. Generalisation essentially asks - is our agent learning the skills needed to play the game, or simply fitting to complete the training data. Through training an agent with Proximal Policy Optimisation on their Procgen Benchmark, (Cobbe et al., 2020) have shown that some agents need to experience over 10,000 procedurally-generated video game levels to become capable of generalisation. In contrast, (Kirk et al., 2023) argue that procedural generation is an unfair way to benchmark generalisation for real-life environments. Since Mario is a simple game with a fixed number of levels, where some are visually similar, we believe policy transfer may be useful in order to generalise early on all the levels individually, and then transfer the agent to the full environment where it would complete the whole game.

Intrinsic reinforcement learning can be broadly defined as a branch of RL in which agents are motivated by an internal reward mechanism, as opposed to any exterior reward function, (Barto, 2013). Random Network Distillation (RND), (Burda et al., 2018), is a method of Intrinsic motivation which aims to overcome the issue of procrastination. Agents are motivated by rewards that encourage exploration, with high rewards for states that are unlike any seen historically. This technique solves the "noisy TV" problem (Mavor-Parker et al., 2022), and was shown to be extremely effective on another platformer game classed as a hard exploration problem, Montezuma's Revenge, (Burda et al.,

2018). We hypothesise that this should help Mario navigate through far more of the levels in our full environment. One weakness of this approach may be that our Mario agent does not utilise sparse rewards as in Montezuma's Revenge, it instead has rewards extremely often, to the point where they are almost too noisy. We additionally hope for RND to help solve this issue.

3 Method

3.1 Network Architectures

During development we experimented with two differing network architectures; a Deep Neural Network (DNN) and a Convolutional Neural Network (CNN). The DNN consists solely of linear layers and takes a feature vector from the environment as input. The feature vector includes information such as Mario's *x* and *y* position, score and state (*small*, *tall*, or *fireball*). The CNN takes a pre-processed frame-buffer as input which is then passed through multiple convolutional layers, following which a fully connected layer outputs the maximal action for the agent. Thus the DNN chooses an action based purely on the state of the game environment whereas the CNN considers the visual representation of the game at the time of decision.

We found that the CNN far outperforms the DNN (see **Appendix: A.10.2**) in testing as the information encoded visually is far richer than that given by the environment feature vector (for example, the DNN cannot observe an enemy Goomba via the feature vector). It is also possible that our DNN had some noise in the feature vector, such as the "coins collected" which may not be related to our reward function.

3.2 Image Preprocessing

In our implementation we preprocess the input image of the CNN in order to increase training speed and minimise noise (in this case useless visual information). The initial size of our input image is *[3, 240, 256]* (where 3 represents the RGB colour channels). We convert the image to greyscale to reduce the dimensionality of the input whilst preserving useful information, as RGB colour is relatively useless for our purposes. We then downsample the image to *[1, 84, 90]*, retaining useful features whilst reducing dimensionality (as the higher resolution image offers minimal added information) thus increasing our training efficiency. We also use the FrameStack wrapper to wrap 4 frames as a single data point, to encode temporal information, such as whether Mario is jumping. Finally we implement our own custom SkipFrame wrapper which leverages the fact that information does not change dramatically between individual frames. SkipFrame skips every *n* frames and aggregates rewards in the intermediary frames, thus sparing training resources at little cost to accuracy, resulting in a final dimensionality of *[4, 84, 90]* for our CNN input.

3.3 Double Deep Q-Network

For our solution, we began with Double Deep Q-Networks (DDQN) (van Hasselt, Guez and Silver, 2015). DDQN uses two networks; an online network, returning an action for a given state, and a target network, used for producing target updates. The target network is never updated and instead copies the online network to itself after a given number of steps to avoid a moving target problem potentially creating instability during training, (van Hasselt, Guez and Silver, 2015). Unlike traditional Deep Q-Networks (DQN), (Mnih et al., 2015), DDQN uses the Q-value returned from the target network corresponding to the best action returned by the online network, given the next state. This helps solve the overestimation bias in normal DQN. Each step of experience is stored in an experience buffer, which is then read in batches to perform updates to the online network. This is achieved by back-propagating the Huber loss between a Q value generated using the output of the target network, and the value generated by the online network. The pseudocode followed for our implementation can be seen in **Appendix: A.3.1**. During training, we updated our target network every 10,000 steps.

3.4 Categorical DQN

In our solution we implement a Distributional Network, one of the components of Rainbow DQN. Categorical DQN generates estimations for the distribution of returns instead of using the expected return. We model this using a variant of the Bellman equation. The value distribution model is created using a discrete distribution, which is supported by a set of atoms. For the atom size we use a value of 51, which performs well according to (Bellemare, Dabney and Munos, 2017) in their tests on the Atari 2600 games. **Appendix: A.5.3** shows the results we obtained when testing out Categorical DQN with a range of different v-min and v-max parameters. Our tests found that this variant failed to improve our agent significantly over time.

3.5 Proportional Prioritised Experience Replay

When stepping through an environment, learning from consecutive observations results in recency bias due to the strong temporal correlations between experiences, causing agents to "forget" about older experiences, (Mnih et al., 2013). Experience replay, introduced in (Lin, 1992) and utilised in (Mnih et al., 2013), breaks this temporal correlation by randomly sampling from past experiences, providing more efficient training.(Lin, 1992). However, random sampling leads to sample inefficiency by revisiting frequently observed data, leaving rarer experiences less attended to. To combat these flaws, our agent uses a prioritised experience replay buffer.

Prioritised Experience Replay (PER) is introduced in (Schaul et al., 2016). The key idea explored is how the importance of an experience is determined. The authors propose using the TD error (δ) to indicate the unexpectedness of an experience, which represents this importance. δs are stored as priorities along with experiences, which are sampled across a probability distribution of priorities, leading to experiences with higher δs being sampled more often. Importance sampling is used to compensate for the bias introduced by PER where updates no longer correspond to a uniform distribution, (Schaul et al., 2016). We linearly anneal β to reach 1 by the end of training to reduce the impact of these weights early on. We implement proportional PER as opposed to rank-based PER due to the higher reliability and stability shown in (Schaul et al., 2016), as well as the fact that our rewards are already clipped within the range of $-15 \leftrightarrow +15$. With our compute limitations in mind, we found that using an experience buffer size of 10,000 and a sample batch size of 32 was the best balance for outputting high quality results whilst keeping training speeds feasible. These hyperparameters also align with (van Hasselt, Guez and Silver, 2015).

Results showing the benefits of PER compared with uniform experience replay, as well as our grid search of PER parameters across our DQN and DDQN, can be seen in **Appendix: A.5.1**. Our results align with those outlined in (Schaul et al., 2016), with the highest performing parameters as follows: $\alpha = 0.6$, $\beta = 0.4$. Also note that all combinations of parameters when we use PER provide better results than using standard experience replay, further aligning with the original paper's findings.

3.6 Multi-Step Learning

Multi-step learning involves using a discounted set of future rewards over multiple steps, as seen in Equation (2) (Hessel et al., 2017), rather than using just the reward from the current step. Multi-step learning often leads to faster training, (Sutton and Barto, 2018). The amount of steps to look ahead is determined by hyperparameter *n*, which we experimented with and found a value of 5 to be appropriate for our agent. The experiments can be seen in **Appendix: A.5.2**.

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} \tag{2}$$

3.7 Dueling Networks

Standard Deep Q-Networks learn values based on the combination of a state and its action. However, as pointed out in (Wang, de Freitas and Lanctot, 2015), it may be that actions do not always affect the agent's environment in any relevant way (for the agent to learn from). The authors propose an architecture which allows for learning the value of states independently of the effects of actions. As explained in the paper, the typical single-stream Q-network used in DQN is split into two streams of fully connected layers for producing estimates for the value and advantage function, which are aggregated to output Q-values in the usual format. This split of the value of a state and the advantage of actions naturally provides generalisation benefits for the agent. Dueling networks are also powerful in allowing previous and future algorithms to easily be built in a modular fashion due to the same input and output format as in standard DQNs.

In our environment our agent, Mario, benefits from a dueling network due to the separation of value and action estimates. For example consider when Mario is in close proximity to an enemy. Separation of the value of the state independent of any actions allows for Mario to understand the difference between being close to the enemy (potentially negative) and the advantageous actions it can take when close to an enemy (potentially positive). This provides both generalisation support and stability in training. As mentioned in the paper, results are even more promising when combined with prioritised experience replay. We implement a Dueling Network following the same architecture as proposed by the authors, seen in **Appendix: A.3.2**, with a single output for the value stream

being aggregated with the Q-values for all actions in the action space of the advantage stream. The suggested aggregation, which we follow, is the addition of the centered mean of the advantage onto the value, (Wang, de Freitas and Lanctot, 2015), as shown in Equation 3.

$$Q(s,a) = V(s,a) + (A(s,a) - mean(A(s,a))) \tag{3}$$

3.8 Noisy Nets

We employed Noisy Networks (NoisyNet) to enhance the exploration capabilities of our agent. The core idea is to inject random noise into the network's weights, thereby encouraging exploratory behavior, (Fortunato et al., 2019). The agent can then learn to enhance or ignore this noise over time. Specifically, we utilized Factored Gaussian Noise due to its computational efficiency. The weights of the network are defined as $\theta = \mu + \sigma \odot \varepsilon$, where μ and σ are learnable parameters representing the mean and standard deviation respectively and ε is the noise drawn from a Gaussian distribution. We could calculate each noisy value independently, but the Factored Gaussian Noise model simplifies this by decomposing ε into only noise values for input and output features, where the noise for each value is calculated as $\varepsilon_{ij} = f(\varepsilon_i^{(row)}) \cdot f(\varepsilon_j^{(col)})$, where $f(x) = \text{sign}(x) \cdot \sqrt{|x|}$.

This approach allows for more effective exploration compared to epsilon-greedy approaches by diversifying the decision-making process. Additionally, we reset the noise at every step, ensuring that the exploration strategy is dynamic and adapts continually, allowing the agent to explore a wide range of strategies and responses in the game environment. This method proved instrumental in enhancing the Mario agent's performance by balancing exploration and exploitation efficiently.

3.9 Policy Transfer

Policy transfer is a way of incorporating and testing an agent's ability to generalise by reusing a learned policy on different environments, with or without extra training. To investigate generalisation, we attempted policy transfer for our agent in two different ways. We first attempted to train an agent on a majority of the levels and tested its ability to generalise on the unseen levels. We then measured its ability to transfer across the two environments, where the only differences were the number of lives, use of checkpoints, and more levels in the second case.

3.10 Hyperparameter Selection

In the given space of time it was not feasible to train an agent using a brute-force grid search on all possible combinations of hyperparameters. We followed the same coordinate descent approach as (Brittain et al., 2020), where we define a fixed set of arguments for a single parameter and then train our agent multiple times, iterating through the arguments using bash scripts. We could then select the best performing argument and carry that forward to future tests. To keep the tests fair, we fixed the random seed such that a bash script was required to reset the seed at the start of each test. Our full hyperparameter diagrams can be found in **Appendix: A.4**.

Whilst this technique fails to find the completely optimal agent, we argue that it drastically reduces the hyperparameter space for a reasonable approximation. For our full RainbowDQN we followed the same method of ablation testing as used by (Hessel et al., 2017), where each feature is turned on and off exclusively, to measure the impact of each standalone feature on our agent. Our final trained agent uses the best combination of all of the above.

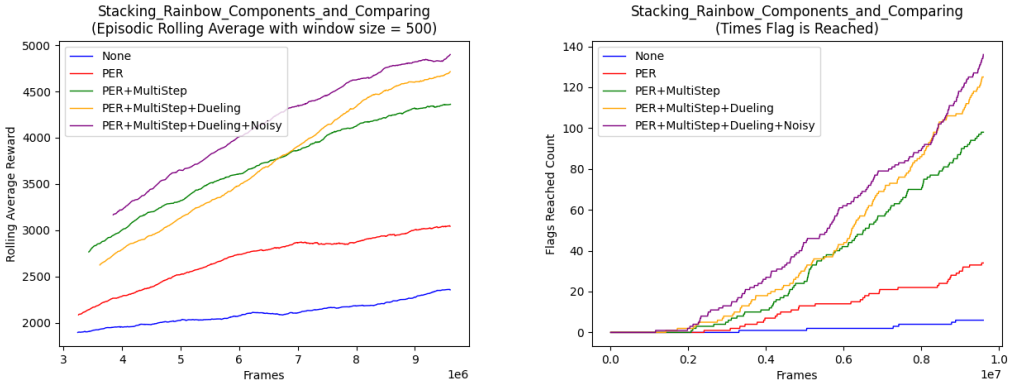
3.11 Random Network Distillation

Random Network Distillation (RND) utilises a target and predictor network to compute the agent's intrinsic reward. The target network remains fixed throughout the learning process, whereas the predictor network aims to predict the output of the target network at every step, updating its parameters given the loss (often mean squared error) between its output and that of the target network's. The intrinsic reward value is the aforementioned loss, which is used in conjunction with the common extrinsic reward to produce a final reward value. The parameters of the predictor network are updated after every step, i.e. during training. Thus the target network is distilled into our predictor network during the training process, with the effect that novel states are explored and learned by the predictor network, therefore losing their novelty (and the high reward associated) the more often they are seen, overcoming the noisy TV problem in traditional intrinsic RL. Our implementation of RND achieved a mixed performance for our given task, suggesting future improvements as discussed in **Results**.

4 Results

4.1 RainbowDQN

For baseline testing we utilise a randomly acting agent that achieves an average score of 250 per episode, with a large standard deviation as it may choose to go leftwards and wait out the clock (leading to negative overall rewards) or rightwards and typically die to the oncoming Goomba with a maximum score of 430.



(a) Episodic rewards

(b) Flags obtained

Figure 1: Stacking RainbowDQN Improvements

In Figure 1 we observe that our agent improves with the addition of each component of RainbowDQN besides the CategoricalDQN, which may be due to a variety of factors including its suitability to our problem or the environment. Overall, we drastically improve upon our DDQN baseline and produce an agent capable of mastering the first level that makes considerable progress in the second. Our agent managed to complete the first level in 20.8 seconds (52 time in-game time steps). This is a 5.2 second improvement on the time taken by a practiced human, and an improvement of between 19.2 and 39.2 seconds on the time taken by a novice human (LeBlanc and Lee, 2021). Our agent also outperformed the agent developed by LeBlanc and Lee by 9.2 seconds (LeBlanc and Lee, 2021), showing that our agent achieves very high performance in the problem space.

4.2 Random Network Distillation

We were able to incorporate Random Network Distillation into our agent with the result of decreased performance. This could be due to various factors, the most notable being a poor reward function. We stipulate that the reward function was inoptimal for our agent and believe a more sparse rewards function that incentives only the time taken would be more useful than receiving a reward for every rightwards step moved, thus adding incentive to wait in dangerous situations.

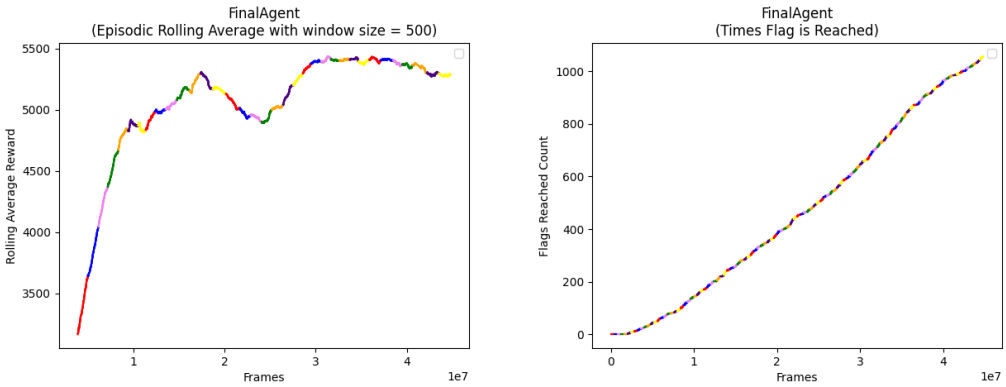
4.3 Generalisation

Our agent demonstrates a poor ability to generalise. We first trained the agent on an environment that provides 31 random levels, changing upon each death, during which our agent consistently underperformed. We then trained our agent on only the first four levels but again it was completely unsuccessful in learning an optimal policy even on seen data. We thus concluded that our agent would be unable to generalise given its pefrormance on mixed seen levels

This could be due to many factors, including our function approximator (CNN) potentially not being complex enough to learn the policy for the far more vast state space introduced by adding levels. Whilst we were unable to to have Mario master multiple levels, there is no reason this should not be possible given enough training time and compute power.

4.4 Final Agent

Our optimal combination of RainbowDQN components are shown in Figure 2 (also see **Appendix: A.9.1**), which includes all but categorical due to its negative impact on our agent's returns. We trained our agent over a two-day period, for a total **44,804,000** frames, to provide more representative results.



(a) Episodic rewards

(b) Flags obtained

Figure 2: Final RainbowDQN Agent

5 Discussion

In comparison to competing implementations, we did not restrict Mario to using a limited action space nor only a single level of the environment. These restrictions improve performance, but do not represent the true game, as limiting movement to only rightwards and upwards on specific levels significantly reduce the techniques that may be learned and make exploration itself artificially more effective.

We have chosen to solve a difficult problem with a complex action space on the full game environment, as well as using a computationally expensive CNN as our function approximator. Our improvements from RainbowDQN were successful and improved training performance and we have been able to run numerous experiments on different framebuffer types and different hyperparameter along with other modifications.

We also evaluated Mario's ability on harder levels in the game. Our agent performs strongly on what humans consider to be the "hardest level", world 8 stage 3. The agent does not master it given the same training time as the first level, however nor does it get stuck in a specific spot, thus we believe given more time it would successfully master the level. Results can be seen in **Appendix: A.10.1**.

6 Future Work

Exponential epsilon-decay is just one technique we can use to encourage our agent to explore. We have already implemented one opposing technique, noisy networks, but we could potentially investigate further contrasting methods, such as sinusoidal epsilon-decay, or rewards-based epsilon decay, which is argued to yield much faster learning, as it waits for proof that our agent has learnt before reducing the amount of exploration (Maroti, 2019). Novel exploration strategies have also been suggested for future investigation, (Burda et al., 2018).

It has been shown that supervised learning techniques that are not typically important in reinforcement learning, such as dropout layers, L2-normalisation and batch-normalisation, can have a significant impact on the ability for a policy to generalise, (Cobbe et al., 2019). In the future it may be insightful to retrain our RainbowDQN agent with our network architectures adapted to include these layers.

Methods in hierarchical reinforcement learning (Sutton, Precup and Singh, 1999) could also be useful to see if our agent could learn higher level skills, such as "sqaush the goomba" or "jump over a pipe". We argue that it may be possible that these skills could help an agent generalise, if it could recognise these object-interactions across environments, which may be worth exploring in the future.

7 Personal Experience

Overall the project was very enjoyable. It was fun training Mario and slowly making improvements by implementing more advanced techniques. We found modern research papers to be more accessible than we expected and some of the unexpected difficulties involved included our measurement of performance and small choices such as when to use steps and when to use episodes.

References

Barto, A.G., 2013. Intrinsic motivation and reinforcement learning. *Intrinsically motivated learning in natural and artificial systems*, pp.17–47.

Bellemare, M.G., Dabney, W. and Munos, R., 2017. A distributional perspective on reinforcement learning. 1707.06887.

Brittain, M., Bertram, J., Yang, X. and Wei, P., 2020. Prioritized sequence experience replay. 1905.12726.

Burda, Y., Edwards, H., Storkey, A. and Klimov, O., 2018. Exploration by random network distillation. 1810.12894.

Cobbe, K., Hesse, C., Hilton, J. and Schulman, J., 2020. Leveraging procedural generation to benchmark reinforcement learning. 1912.01588.

Cobbe, K., Klimov, O., Hesse, C., Kim, T. and Schulman, J., 2019. Quantifying generalization in reinforcement learning. 1812.02341.

Fortunato, M., Azar, M.G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C. and Legg, S., 2019. Noisy networks for exploration. 1706.10295.

Hasselt, H. van, Guez, A. and Silver, D., 2015. Deep reinforcement learning with double q-learning. 1509.06461.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2017. Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the aaai conference on artificial intelligence* [Online], 32. Available from: https://doi.org/10.1609/aaai.v32i1.11796.

Kauten, C., 2018. Super Mario Bros for OpenAI Gym [Online]. GitHub. Available from: https://github.com/Kautenja/gym-super-mario-bros.

Kirk, R., Zhang, A., Grefenstette, E. and Rocktäschel, T., 2023. A survey of zero-shot generalisation in deep reinforcement learning. *Journal of artificial intelligence research* [Online], 76, p.201–264. Available from: https://doi.org/10.1613/jair.1.14174.

LeBlanc, D.G. and Lee, G., 2021. General Deep Reinforcement Learning in NES Games. *Proceedings of the canadian conference on artificial intelligence*. Https://caiac.pubpub.org/pub/m4ue9ykj.

Lin, L.J., 1992. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University.

Maroti, A., 2019. Rbed: Reward based epsilon decay. *Arxiv* [Online], abs/1910.13701. Available from: https://api.semanticscholar.org/CorpusID:204960941.

Mavor-Parker, A., Young, K., Barry, C. and Griffin, L., 2022. How to stay curious while avoiding noisy TVs using aleatoric uncertainty estimation [Online]. In: K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu and S. Sabato, eds. *Proceedings of the 39th international conference on machine learning*. PMLR, *Proceedings of Machine Learning Research*, vol. 162, pp.15220–15240. Available from: https://proceedings.mlr.press/v162/mavor-parker22a.html.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. *arxiv preprint arxiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015. Human-level control through deep reinforcement learning. *Nature* [Online], 518, pp.529–33. Available from: https://doi.org/10.1038/nature14236.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2016. Prioritized experience replay. 1511.05952.

Sutton, R., 1988. Learning to predict by the method of temporal differences. *Machine learning* [Online], 3, pp.9–44. Available from: https://doi.org/10.1007/BF00115009.

Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction* [Online]. 2nd ed. The MIT Press. Available from: http://incompleteideas.net/book/the-book-2nd.html.

Sutton, R.S., Precup, D. and Singh, S., 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. intell.* [Online], 112, pp.181–211. Available from: `https://api.semanticscholar.org/CorpusID:76564`.

Wang, Z., Freitas, N. de and Lanctot, M., 2015. Dueling network architectures for deep reinforcement learning. *Corr* [Online], abs/1511.06581. Available from: `http://arxiv.org/abs/1511.06581`.

A Appendix

A.1 Video of agent learning

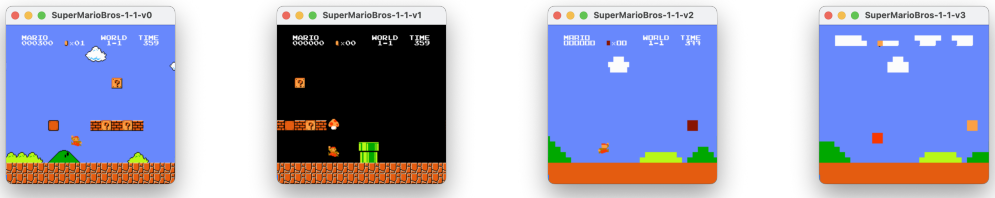
https://www.youtube.com/watch?v=AWjrj-vUKH8

A.2 Problem Definition

A.2.1 Environment Options

Table 1: Environment Options

	Single Level	Full Game	Random Stage Selection
Number of Lives	1	3	1
Number of Levels	1	32 (ordered)	up to 32 (any order)



(a) Version 0 (b) Version 1 (c) Version 2 (d) Version 3

Figure 3: Possible Versions of our Environment Framebuffer

Index	Action
0	NOOP (do nothing)
1	right
2	right, A
3	right, B
4	right, A, B
5	A
6	left
7	left, A
8	left, B
9	left, A, B
10	down
11	up

Table 2: Action Space of our Mario Agent

A.3 Background

A.3.1 DDQN

Initialise replay memory D to capacity N
Initialise action-value network $\hat{q}_1$ with parameters $\boldsymbol{\theta}_1 \in \mathbb{R}^d$ arbitrarily

Initialise target action-value network $\hat{q}_2$ with parameters $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$

Loop for each episode:
Initialise S
Loop for each step of episode:
Choose action A in state S using policy derived from $\hat{q}_1\left(S, \cdot, \theta_1\right)$
Take action A, observe reward R and next-state S'
Store transition (S, A, R, S') in D
For each transition $\left(S_j, A_j, R_j, S'_j\right)$ in minibatch sampled from D :

$$y = \begin{cases} R_j & \text{if } S'_j \text{ is terminal} \\ R_j + \gamma \max_{a'} \hat{q}_2\left(S'_j, a', \boldsymbol{\theta}_2\right) & \text{otherwise} \end{cases}$$

$\hat{y} = \hat{q}_1\left(S_j, A_j, \boldsymbol{\theta}_1\right)$
Perform gradient descent step $\nabla_{\theta_1} L_\delta(y, \hat{y})$
Every C time-steps, update $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$

Figure 4: DDQN pseudocode (Sutton and Barto, 2018)

A.3.2 Dueling Network Architecture

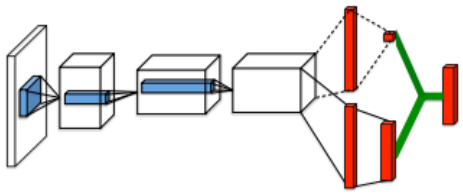


Figure 5: Dueling Network Architecture implemented, taken from (Wang, de Freitas and Lanctot, 2015). The value stream (single output) and advantage stream (output of size equal to the action space size) are aggregated to output Q-values for each action

A.4 Hyperparameter Tuning

A.4.1 Learning Rate Tuning (Across 600,000 steps)

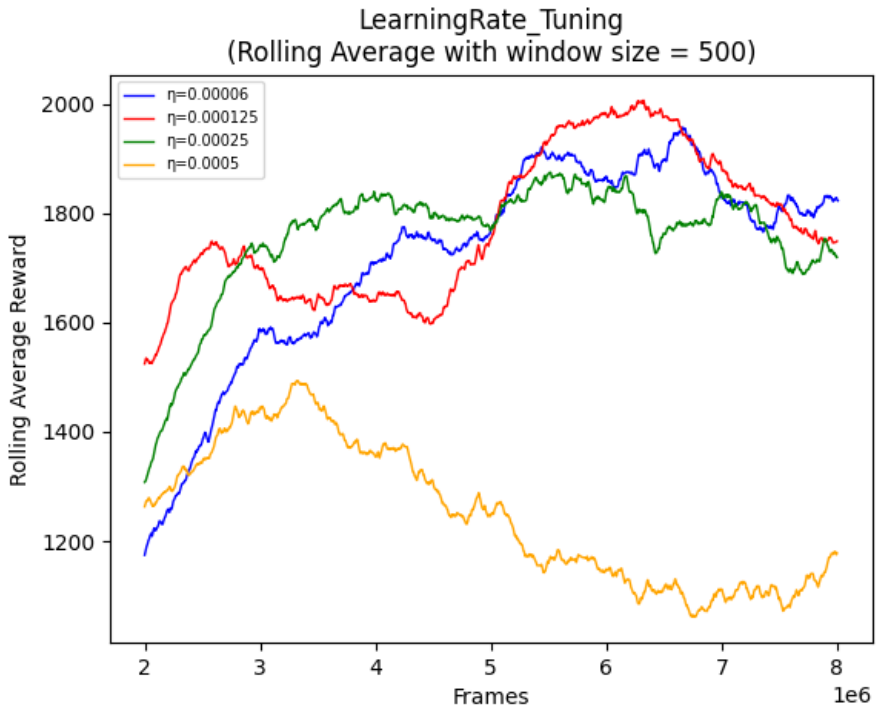
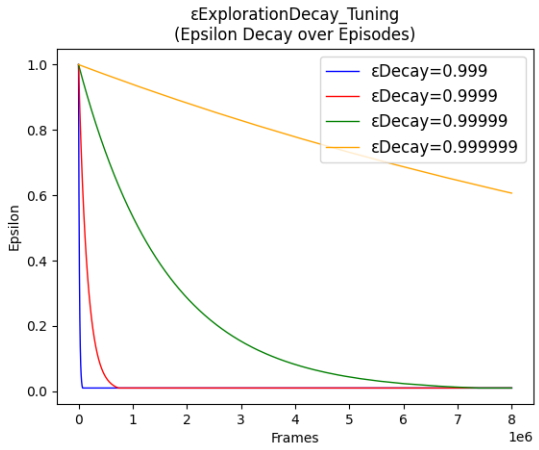
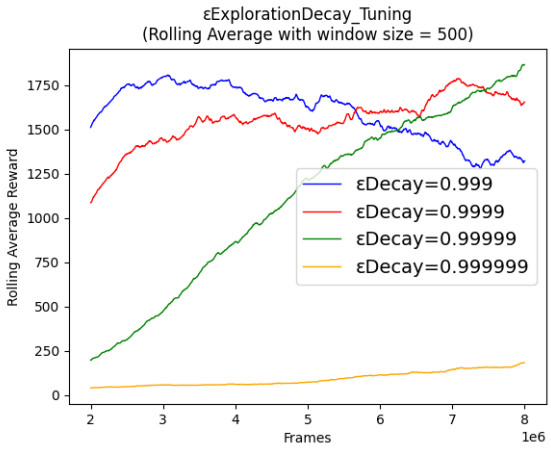


Figure 6: Learning Rate Tuning (η) - *Rolling Average 500*

A.4.2 Epsilon Decay Tuning (Across 600,000 steps)



(a) ϵ Decays Plotted



(b) ϵ Decays - *Rolling Average 500*

A.4.3 Discount Factor Tuning (Across 600,000 steps)

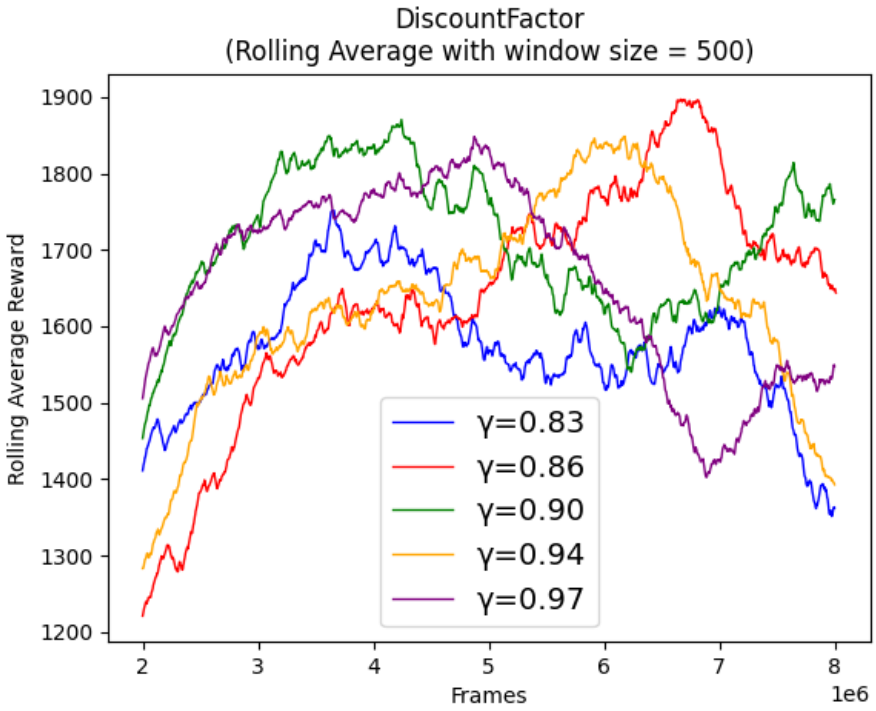


Figure 8: Discount Factor Tuning (γ) - *Rolling Average 500*

A.5 Adam (Optimisation) Tuning (Across 600,000 steps)

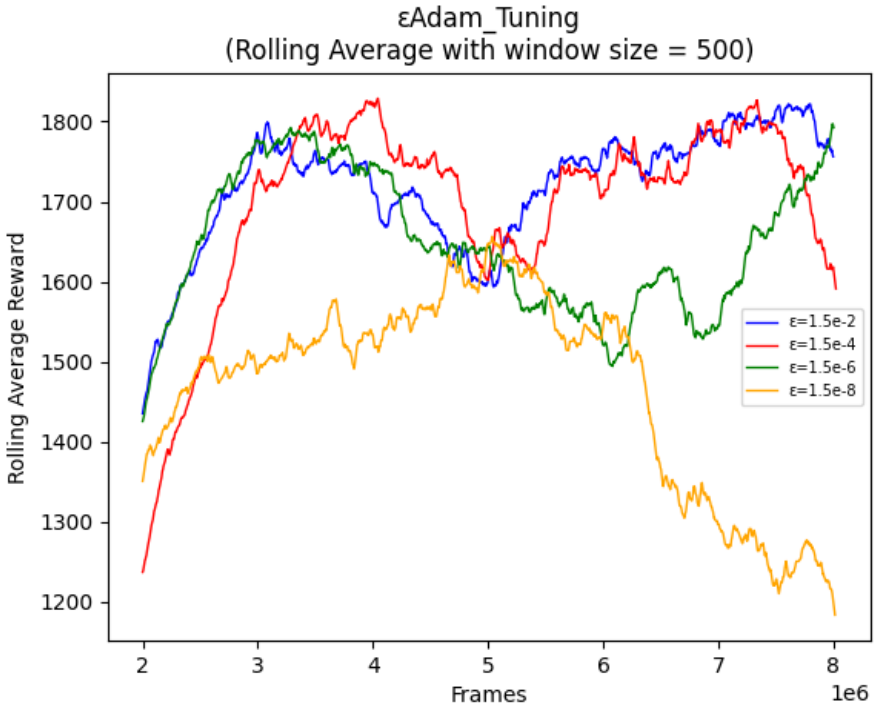


Figure 9: Adam (Optimisation) Tuning (ϵ) - *Rolling Average 500*

A.5.1 PER Tuning on DQN and DDQN (Across 600,000 steps)

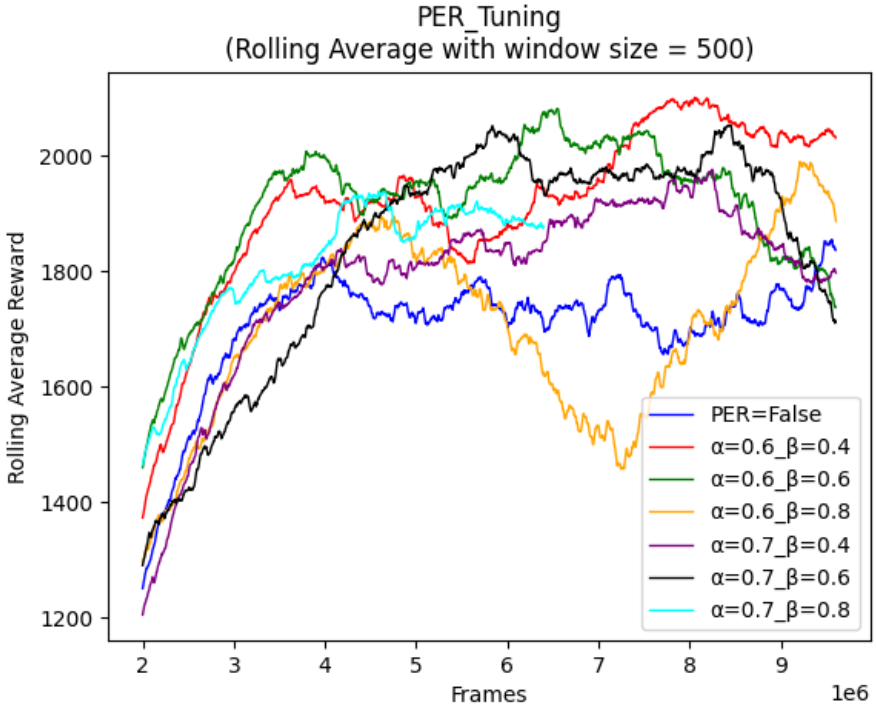


Figure 10: PER Tuning (α, β) **on DDQN** (Significantly Better Performance than DQN) - *Rolling Average 500*

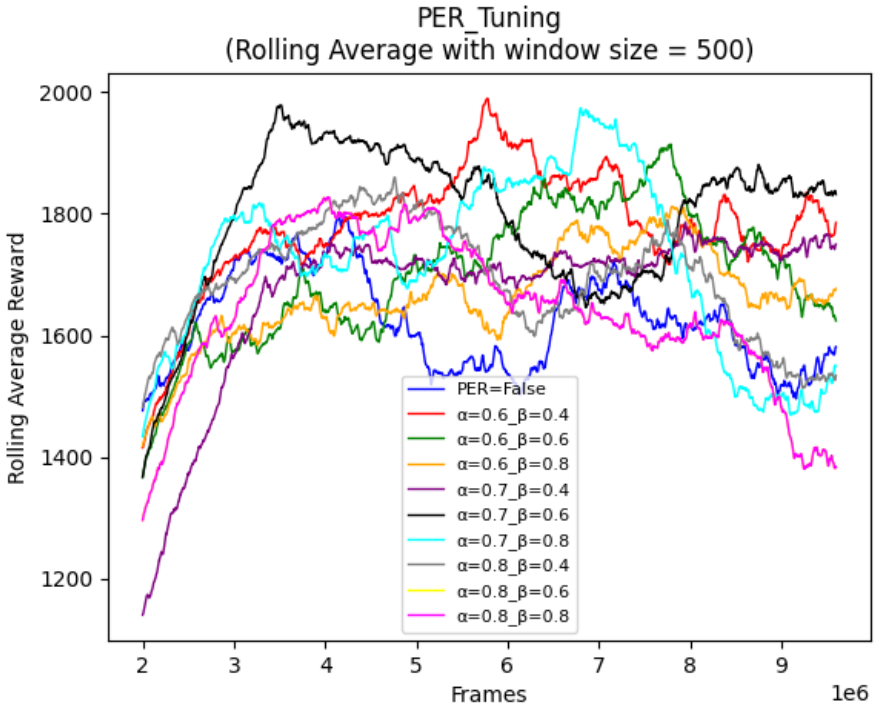


Figure 11: PER Tuning (α, β) **on standard DQN** - *Rolling Average 500*

A.5.2 MultiStep Tuning on DQN and DDQN (Across 600,000 steps)

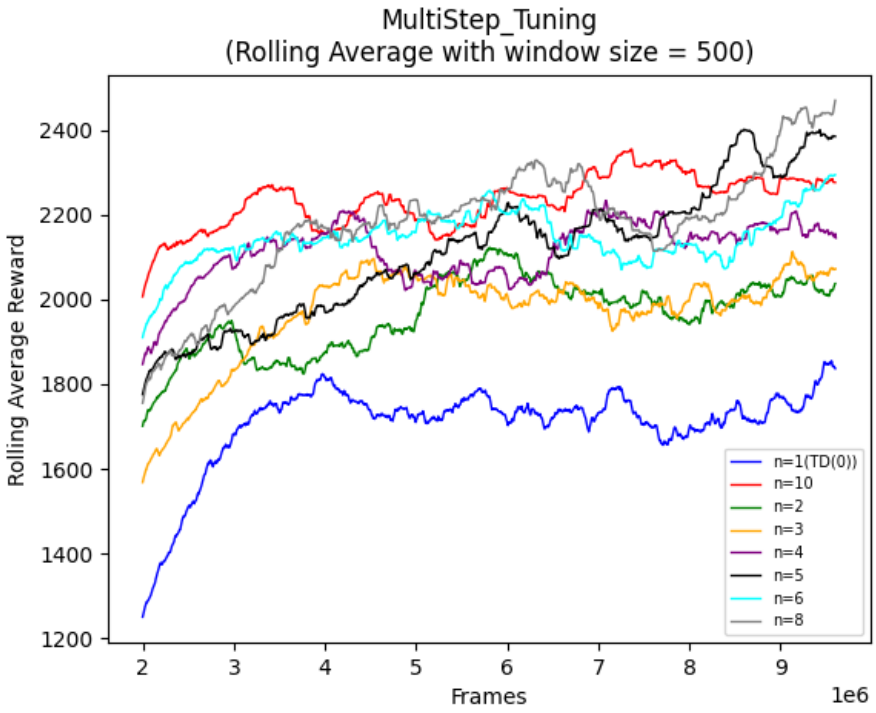


Figure 12: MultiStep Tuning (n) on **DDQN** (no dropoff compared to DQN) - *Rolling Average 500*

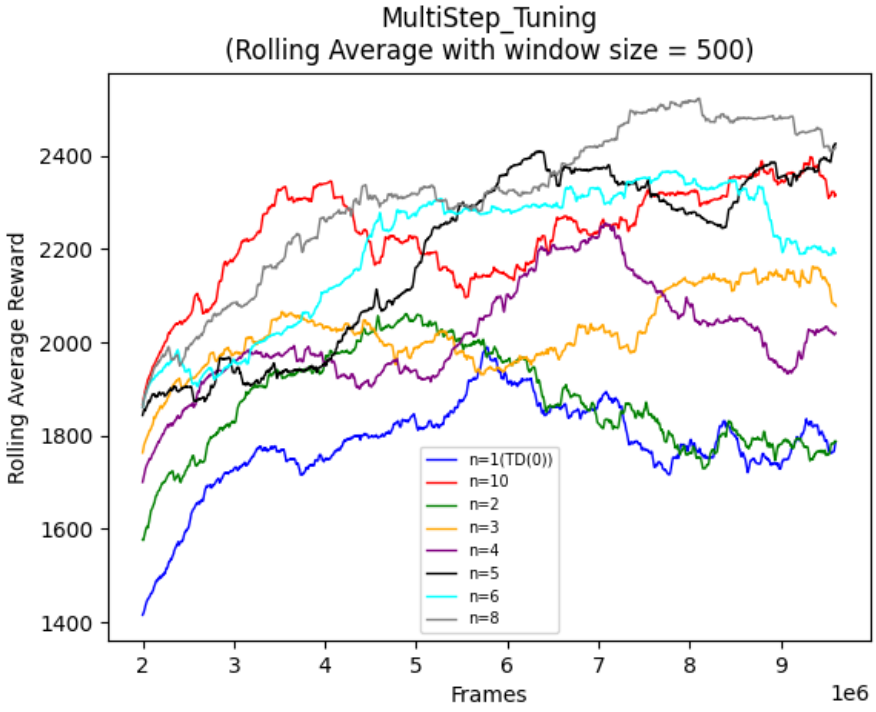


Figure 13: MultiStep Tuning (n) on standard **DQN** - *Rolling Average 500*

A.5.3 Categorical (C51) DQN Tuning (Across 600,000 steps)

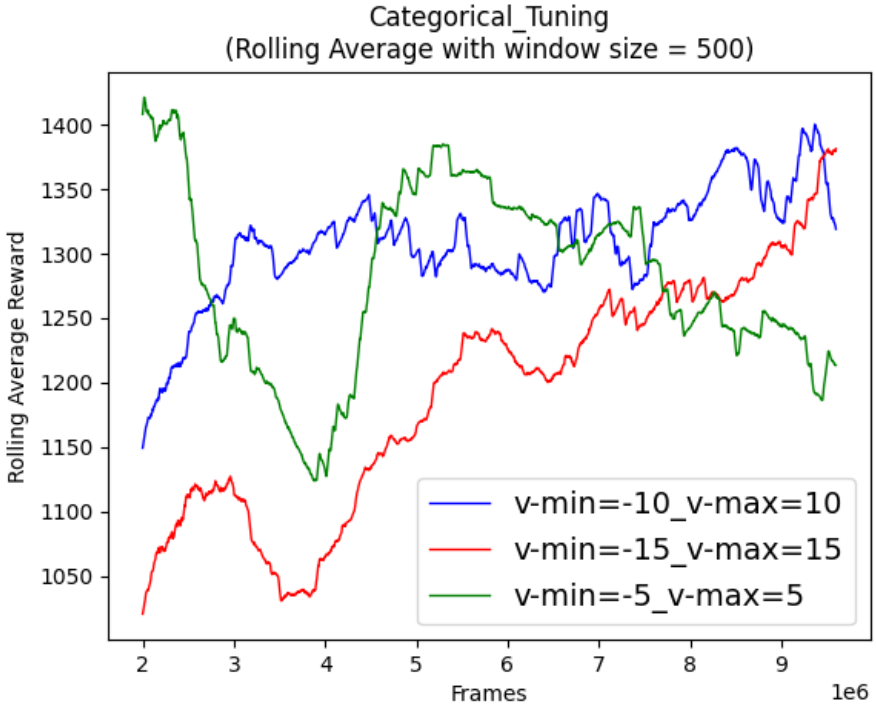


Figure 14: Categorical DQN Tuning (v_min, v_max, num_atoms = 51 (C51)) - *Rolling Average 500*

A.5.4 Noisy Network Tuning (Across 600,000 steps)

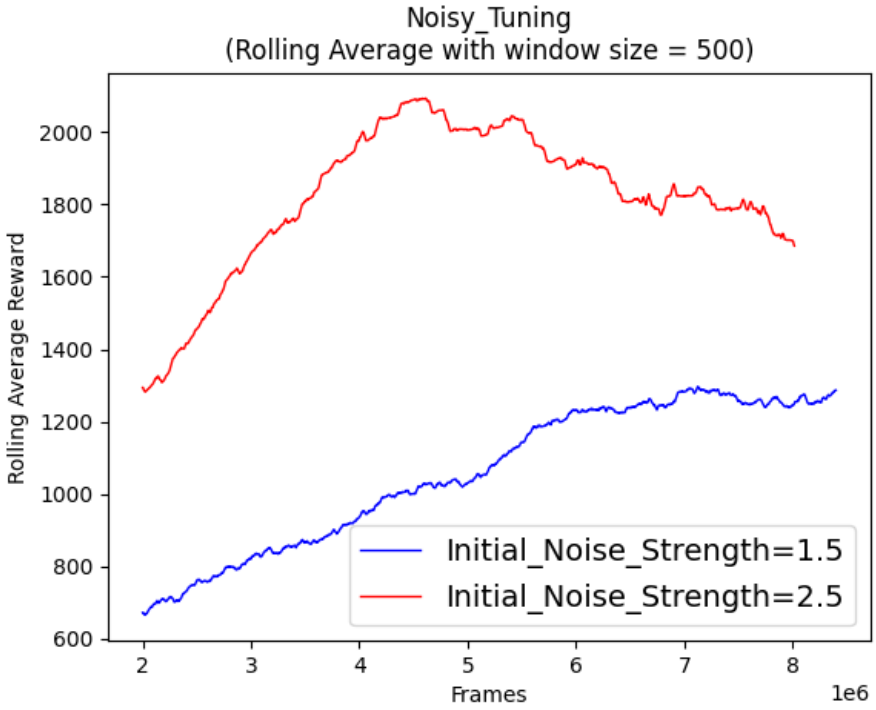


Figure 15: Noisy Network Tuning (initial noise strength) - *Rolling Average 500*

A.6 Final Parameters Used

Parameter	Value
Copy Network Frequency	10,000 steps
Experience Buffer Size	10,000
Experience Buffer Sample Batch Size	32
Discount Factor	0.9
Initial/Maximum ϵ	1.0
Ending/Minimum ϵ	0.01
ϵ Decay	0.9999
Learning Rate	0.00025
Adam ϵ	0.00015
PER α	0.6
PER β	0.4
PER ϵ	0.1
N-Step n	5
Categorical NUM-ATOMS	51
Categorical V-MIN	-10
Categorical V-MAX	10
Noisy Initial Strength	2.5
RND Initialisation Steps	200
RND Intrinsic Discount Factor	0.999

Table 3: Hyperparameters Used

A.7 RainbowDQN

A.7.1 Stacking Components and Comparing (Across 600,000 steps)

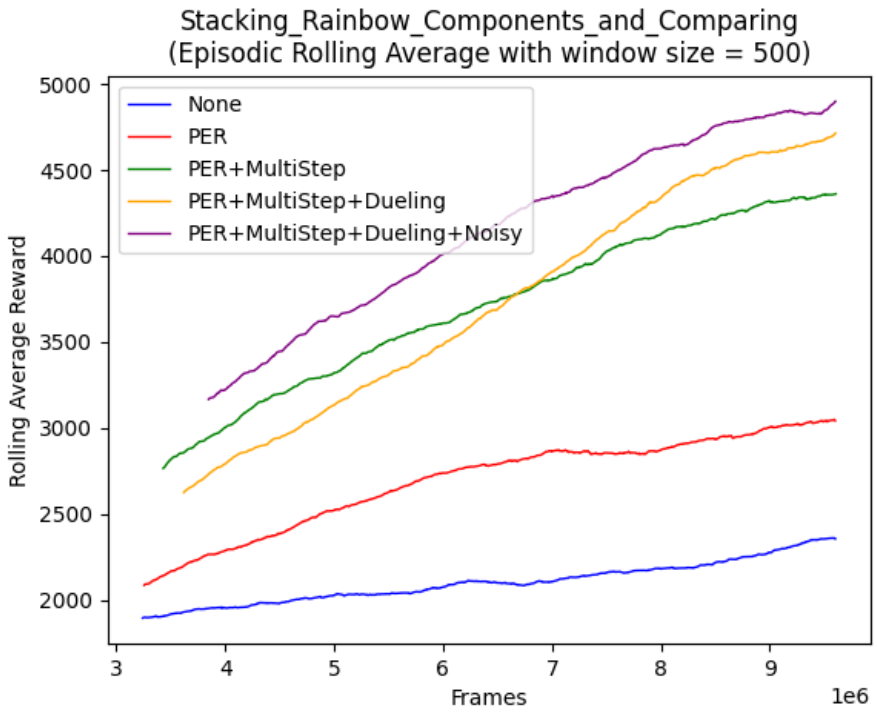


Figure 16: Stacking RainbowDQN components one at a time - *Episodic Rolling Average 500*

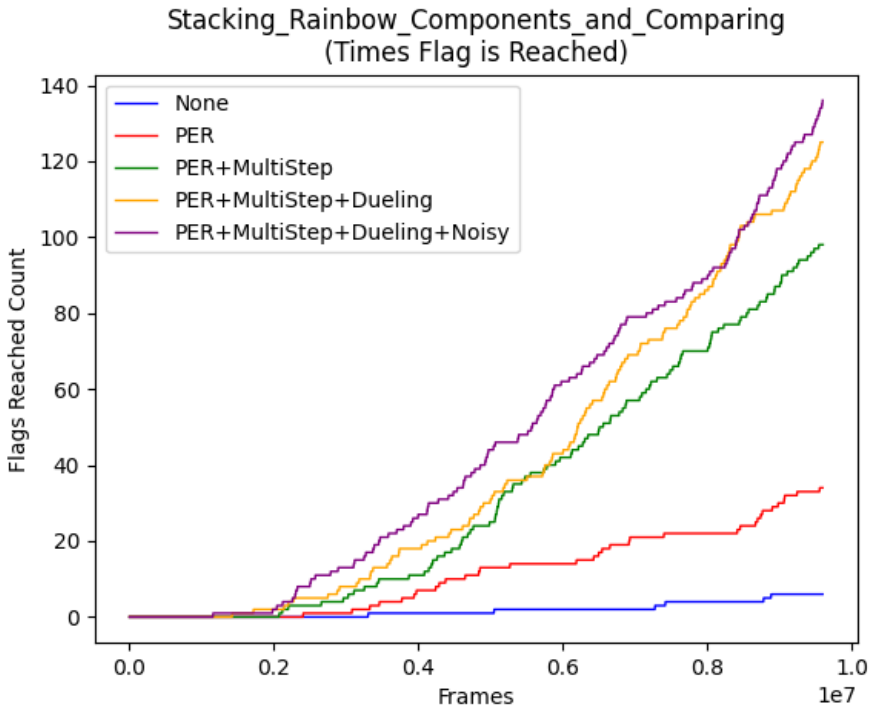


Figure 17: Stacking RainbowDQN components one at a time - *Flags Reached (a level is completed)*

A.8 Policy Transfer Learning

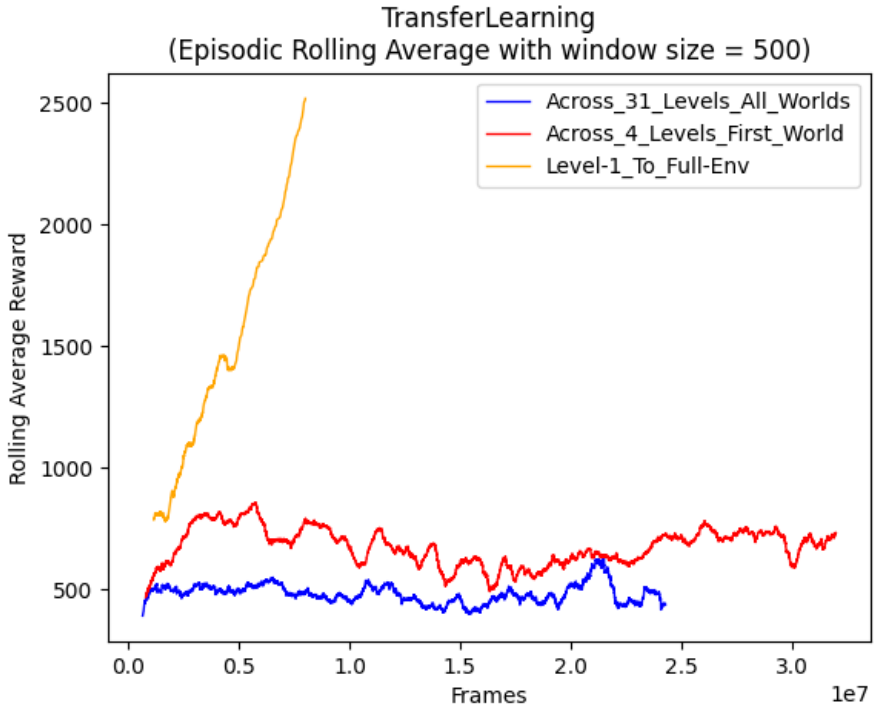


Figure 18: Policy Transferring Across Levels and Different Life Counts - *Episodic Rolling Average 500*

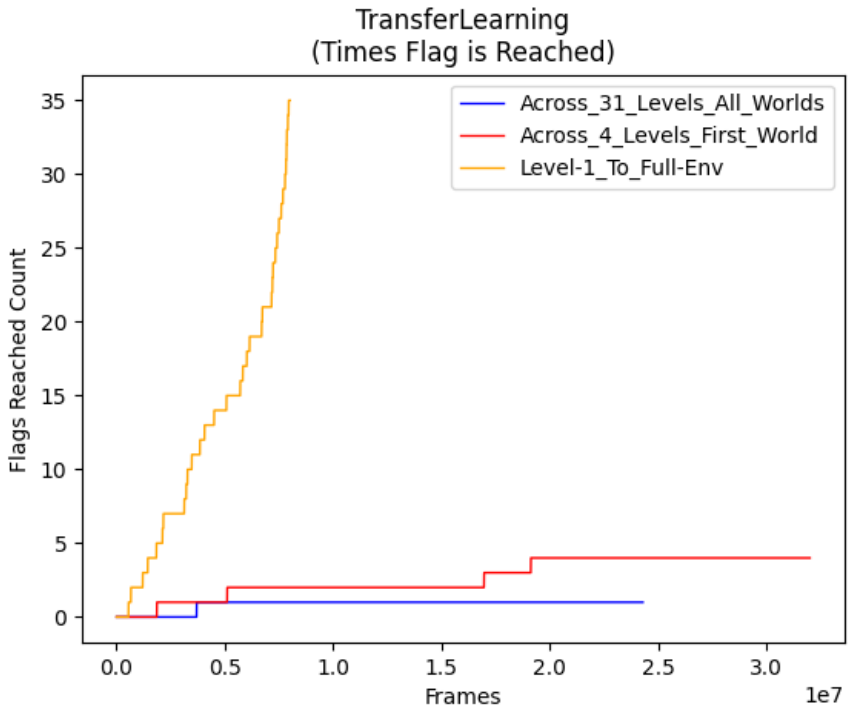


Figure 19: Policy Transferring Across Levels and Life Counts - *Flags Reached (a level is completed)*

A.9 Final Agent Results

A.9.1 Final Agent Using All of RainbowDQN, bar Categorical

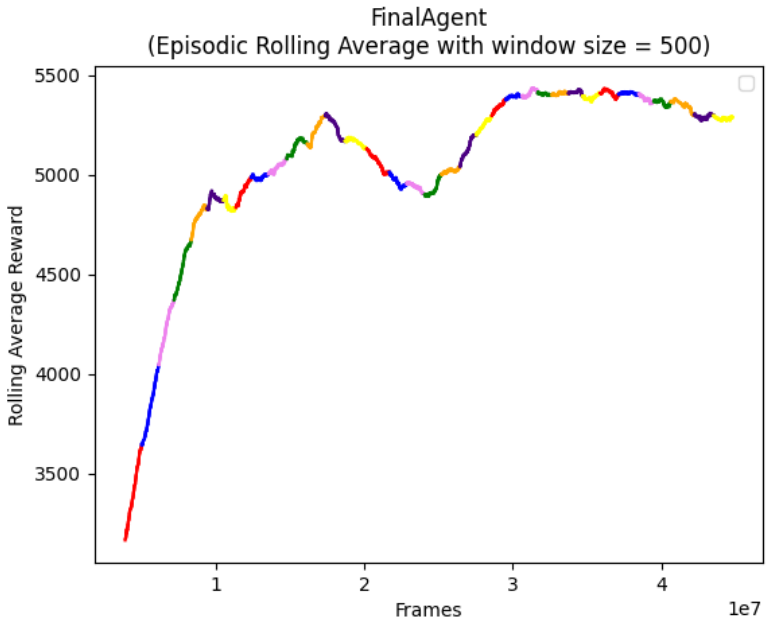


Figure 20: Final Agent Results - *Rolling Average 500*

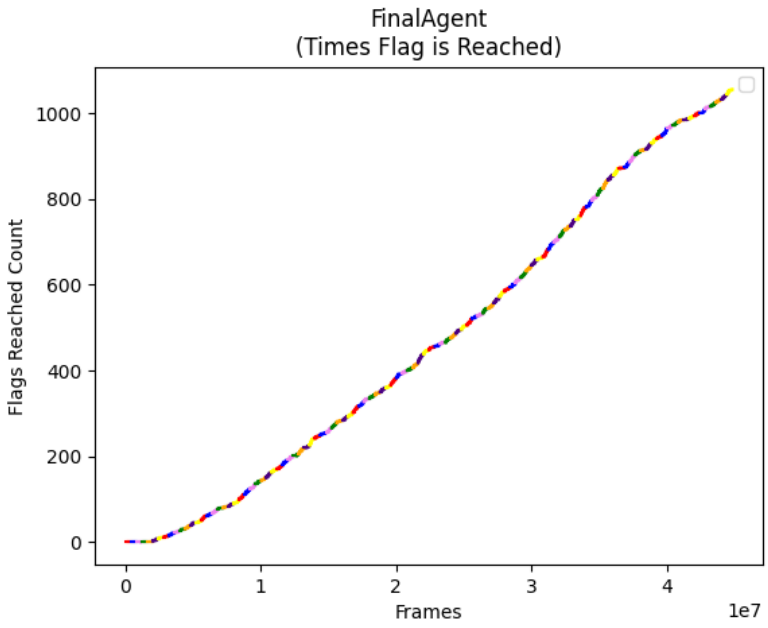


Figure 21: Final Agent Results - *Flags Reached (a level is completed)*

A.10 Other Tests

A.10.1 Final Agent on the Easiest vs Hardest Level

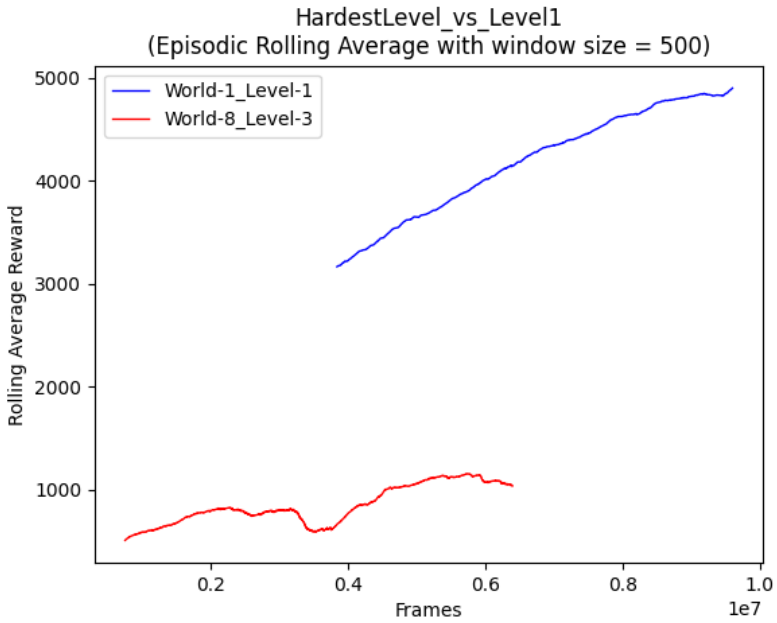


Figure 22: Easiest vs Hardest Level (note longer episodes means lines start later) - *Rolling Average 500*

A.10.2 CNN vs DNN on an early DQN model

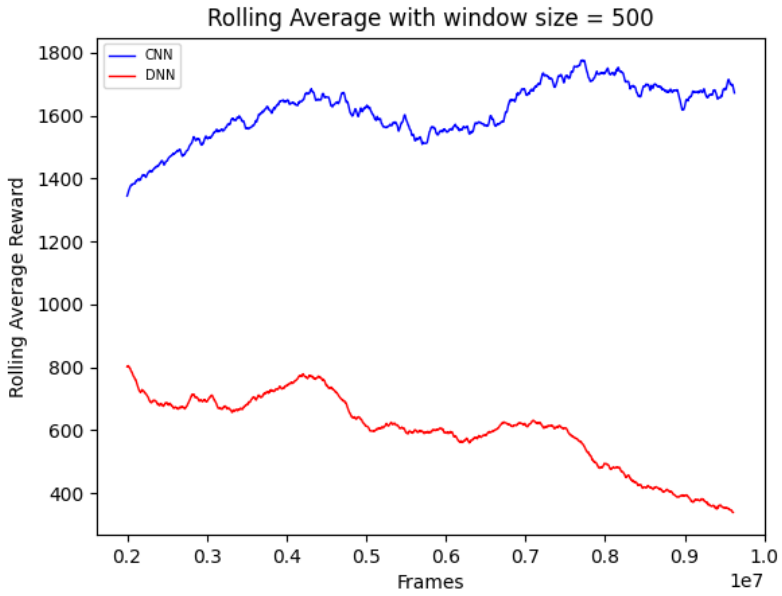


Figure 23: Early DQN Model using CNN vs DNN - *Rolling Average 500*