Learning to Play Super Mario Bros Through Deep Reinforcement Learning Methods

Student Name: Tom Potter

Supervisor Name: Dr Lawrence Mitchell

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

***Abstract* —**

Context/background: Deep reinforcement learning (DRL) methods have proved capable of generating super-human performance across a spectrum of computer games, extending agents to learn over even the most complex environments. However, this research commonly focusses on a small collection of game environments, such as the *Atari 2600* collection; significantly less work has been produced in less typical domains, such as NES games.

Aims: This project aims at investigating whether contemporary DRL methods typically applied to the Atari collection can be modified to create an agent which can efficiently learn to play in the NES game *Super Mario Bros*. This play should as time and data efficient as possible, and produce generalisable behaviour which isn't overtuned to the specific appearance of the training environment.

Method: A complete system for the employment of an agent to play Super Mario Bros is developed. Initially, a simplistic deep Q-learning agent is presented; then, multiple extensions to this method (typical of Atari agents) are implemented to improve performance and data efficiency. This begins with the progression to double-deep Q-learning, then explores a spectrum of implementations of neural network architecture, experience replay, state and action space manipulation, and an intrinsic reward system.

Results: The presented agent is shown to be consistently effective at crossing 71.6% of the training level (as a maximal average), beating the few existing agents developed for Super Mario Bros. This proves that the devised network, memory and intrinsic reward systems effectively incentivise an agent, in a time and data efficient manner.

Conclusions: The experimentation proved that many of the DRL methods proposed for learning Atari gameplay are effective in the domain of Super Mario Bros. This included Double Deep Q-Learning, experience replay, and intrinsic rewards, which were successfully employed to learn generalised behaviour, which performed well on both the training level and those of a similar structure.

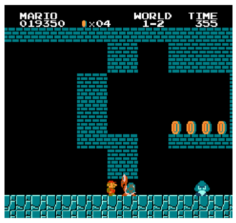
***Keywords* —** Reinforcement Learning, Double Deep Q-Learning, Deep Learning, Convolutional Neural Networks, Prioritised Experience Replay, Intrinsic Rewards, Super Mario Bros.

I INTRODUCTION

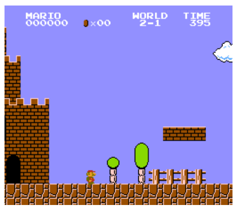
The development of artificially intelligent players for games has long been a focus of research. Some of the earliest machine learning methods operated as game players. Many of these early systems even outperforming their leading human counterparts, such as Tesauro's *TD-Gammon* (Tesauro 1995)—a *temporal difference learning* based backgammon player. Training machines to play board games is still a field of much progression; notably, DeepMind's AlphaZero learning superhuman play in chess, shogi, and Go (Silver et al. 2017a). However, recent research has focussed upon training players for computer games. A major factor in this popularity is the availability of a vast spectrum of varied games to learn over. The intrinsically controllable nature of these worlds, the minimal consequences for erroneous behaviour (when compared to learning in the physical world), and the fast speed at which experiences can be generated, further establish computer games as attractive machine learning environments (Shao et al. 2019).

Typically, these systems employ reinforcement learning, where an unsupervised *agent* is tasked with optimising their sequential decisions based upon a dynamic interaction with their *environment*. For most computer games, the agent is the player controlling the game character, and the world within which this character moves is the environment. The rise of deep learning has introduced new techniques to scale these methods to previously intractable environments (namely larger, more complex games) through employing *deep neural networks* (Arulkumaran et al. 2017), defining the field of *deep reinforcement learning* (DRL).

(a) World 1 Stage 1



(b) World 1 Stage 2



(c) World 2 Stage 1

Figure 1: A collection of game stages

The Nintendo Entertainment System game *Super Mario Bros* is regarded as one of the most commercially successful computer games in history—the original 1985 release selling over 50 million copies (IGN 2012). Despite this popularity, recent DRL success has largely overlooked this domain, instead focussing upon other games, such as the *Atari 2600* collection (Arulkumaran et al. 2017). Atari games have become an industry standard domain for evaluating DRL methods due to the variety of constituent games, allowing agents to learn behaviour over environments with diversity in both appearance and interaction style. The ease of accessibility of these games (for example through *OpenAI Gym*) further solidifies this prominence. Recent research has also focussed upon more complex *open-world* games, such as *ViZDoom* (a platform for the game *Doom*), and *StarCraft* (Arulkumaran et al. 2017). However, the existence of Super Mario Bros between these two classes—not having as fully-fledged an ecosystem of tools, and not being as unique as open-world games—is a possible reason for its oversight. Notwithstanding, this environment has many novel challenges and interactions which are fascinating to explore through DRL.

A Background

The aim of DRL is to produce an effective player through learning optimal valuations of each possible *action* that could be taken in each of the environment's *states*. For each time step t in the game, the action a_t is a possible move the game player can make (from the total *action space* A), and the state is s_t (part of the *state space* S) represents the current features of the environment: for example, a matrix of pixel values from a single frame of Mario gameplay.

Given action a_t is taken in state s_t, we denote the subsequent transition of experience as the tuple $(s_t, a_t, r_{t+1}, s_{t+1})$, where r_{t+1} signifies the *reward* (indicating the success of the previous step) and s_{t+1} the *successor state*. The sequence of transitions from an initial state s_0 to terminal state s_T is known as an *episode*. The success of each state can be deduced from the *return* R_t, calculated as the sum of rewards from some time t until termination:

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_{t+1}, \tag{1}$$

where γ is the *discount factor* weighting rewards between those awarded immediately and in the and long-term (Mnih et al. 2013).

Q-learning calculates the worth of states and actions through a *Q-function*. In simple environments this can be achieved through some *Q-table*, enumerating all possible states and actions. This function is exploited to select actions within the environment through an internal behaviour *policy* π_Q, mapping each state to an action: $\pi_Q \colon S \to A$.

In environments with large state and action spaces, this is computationally infeasible. Instead, we can conduct *deep Q-learning* (DQL), where the approximate Q-function $\tilde{Q} = Q$ is followed, allowing learning over high dimensionality spaces (Barreto et al. 2020). This shifts the goal of the agent from learning specific valuations to finding an optimised parameterisation of an approximator $\tilde{Q}$, commonly implemented as a *neural network*. Typically, the function is trained such that its outputs converge upon the expected return R_t of taking a_t in s_t:

$$\tilde{Q}^{\pi}(s_t, a_t) = \mathbb{E}\left[R_t | s_t, a_t\right]. \tag{2}$$

A *neural network* approximates the mapping between inputs and outputs through sequential passes through a set of layers. The network's *input layer* receives a set of data elements and passes them through a series of *hidden layers*, eventually producing a set of outputs across the *output layer*. Intrinsic to each layer is a set of *neurons*, each connected to the preceding layer via *weighted channels*. The associated *activation* value $a_L^n \in [0, 1]$ for each neuron n in layer L is computed via an *activation function* on input of the sum of activations over the vector of input channels, parameterised by a given weight matrix and *bias*. These parameters are tuned during training to find a minimum point in the loss between a set of output values and their optimal values. The gradient of the loss function is calculated via *backpropagation*, and network parameters adjusted in the direction of the negative gradient (known as *gradient descent*).

B Environment

Super Mario Bros is a *side-scrolling platform game*, where the player controls the game's protagonist *Mario*, navigating through his world from the start screen (leftmost point) to the goal state (rightmost point). This world contains coins, power boost items, and enemies. The collection of coins and elimination of enemies increases the game score. The primary aim of a player is to navigate from the start to the goal state, whilst avoiding enemies and maximising the score. The game consists of eight worlds, each containing of four *stages*. Each stage has a differing appearance, introducing novel challenges to the player—exemplified in Figure 1.

To navigate through the environment, the player manipulates Mario's movement through a controller consisting of two input types: a control pad, and a set of buttons. The control pad informs the direction of Mario's movement: *left* or *right*. The available buttons are as follows: *start*, *select*, *A*, and *B*. The former two are menu controls, and are out of the scope of this project. The *A* button allows Mario to jump, and the *B* button allows Mario to run in a specified direction.

There are two types of power boost item: the *magic mushroom*, and *fire flower*. Collecting these items apply time-limited power-ups; a magic mushroom transforms Mario into *Super*

Mario, and a fire flower unlocks *Fiery Mario* (Nintendo Entertainment System 1985). Both provide temporary invincibility against enemies, whilst Fiery Mario allows the player to throw flaming projectiles which eliminate enemies and consequently increase the game score. These benefits will prove useful to consider when ingraining an agent with a reward system that incentivises the optimisation of its behaviour and maximisation of the game score.

C Objectives

The research question posed by this paper is: *Can deep reinforcement learning methods devised for the Atari game collection be manipulated to learn generalised play in the NES game Super Mario Bros?* To address this, a stepping-stone approach will be employed to converge on the most effective generalisable agent; namely, one which can be trained upon one stage of the environment, and effectively applied to others. Four main aspects of the learning process will be addressed: the specific DRL method, state and action space representation, experience memory, and the neural network design. The minimum objective is a superficial evaluation of each field, devising a solution for each. The intermediate objective progresses each aspect to rely upon state-of-the-art techniques popularised through their utilisation in other domains. Finally, I will evaluate the effectiveness of these methods in resolving the presented research question. To do this, the solution will be trained on the initial stage, and tested on further stages to observe the success of each additional method upon the agent's generalisability and training efficiency.

II RELATED WORK

Reinforcement learning typically offers two main challenges: efficiently abstracting useful information from the state and action spaces (known as *feature extraction*), and learning a near-optimal policy for our agent. Whilst feature extraction can be performed on non-visual states, such as the analysis of RAM values, this project focusses on utilising pixel matrices of each frame of gameplay. A core reason behind this is the large variety of established manipulation methods for images, progressing in complexity from naive observations to complex systems of adaptation and downscaling. The process of learning a policy is equally as varied. Approaches vary from explicit action-value functions such as the *temporal-difference learning* of Tesauro's TD-Gammon (Tesauro 1995), to the employment of deep learning approximations, such as DeepMind's Agent57 (Puigdomènech Badia et al. 2020).

A Feature Selection and Sparse Extrinsic Rewards

When learning in complex environments, effort is required to reduce the dimensionality of state and action data. This is achieved through an encoding function, maintaining high-level features whilst reducing the overall data size. A core component of the *curiosity-driven* agent proposed by Pathak et al. (2017) for ViZDoom and Super Mario Bros, a feature vector can be inferred for each state, only representing the most relevant stimuli to the agent. This discarding of redundant information is useful in the abstraction of both the state and action spaces, and will be relied upon for subsequently presented dimensionality reductions.

A notable issue faced by many deep reinforcement learning approaches is attempting to train an agent in the presence of a *sparse extrinsic reward* signal r_t^e. In these environments, only a small proportion of states return a positive reward to the agent, such as games which only generate a reward in the goal state. A prime example of this is Super Mario Bros, where the vast majority of transitions induce zero reward; whilst in-game rewards are present (Section I B), the primary goal of rightwards motion is only rewarded if the end of the level is reached. The problem of *curiosity traps* is of similar origin: the agent can become stuck forever exploring unlearnable areas of the environment (such as complex dynamics, e.g. the effects of wind), as there is no extrinsic signal informing it that exploration here is not helpful. Similar issues are exhibited in Mario's world, such as obstacles which limit Mario's motion and require complex interactions to evade. If the agent gets stuck in one of these areas, experience will be dominated by monotonous transitions, which do not contain much useful information about the wider environment and thus are not beneficial to the general learning process.

Pathak et al. (2017) address these issues through the implementation of an *intrinsic reward system* based upon an environment dynamics model. The reward signal $r^i(s_t) = r_t^i$ is constructed limiting the incentive to learn upon states where the difference between the agent's predicted consequences of its actions and the observed consequences is not improved through training. A behaviour policy is then followed which selects actions that maximise both the extrinsic and intrinsic rewards. This model-based methodology is similarly exhibited in the *generative adversarial network* inspired approach of Engelsvoll et al. (2020). Namely, two agents are pitted against each other: a level creation network, learning to generate challenging instances of Super Mario Bros stages, and a player learning a behaviour policy from these. This eventually produces an accurate modelling network, and player for these stages.

The presented solution aims to tackle these problems without relying on the complex modelling procedures of Pathak et al. (2017) and Engelsvoll et al. (2020). A focus is placed upon a model-free system to provide additional rewards for an agent, guiding its learned behaviour, in a computationally intensive manner than the observed model-based methods.

B Double Deep Q-Learning

The values of the Q-function can be learned through the minimisation of the difference, for a set of function outputs x on state s_t, between x and the *Q-target* y—where y is the value upon which we want x to converge. Ideally, we want y to be the optimal value for x; however, these optimums are rarely available. Instead, we calculate outputs x_i for iteration i of training using the current Q-function $x_i = Q\left(s_t, a_t\right)$; the targets y_i then take the expected value of the sum of the immediate reward and maximal action-value in the successor state s_{t+1}:

$$y_i = \mathbb{E}_{s_{t+1}}\left[r_{t+1} + \gamma \max_{a' \in A}\left(Q\left(s_t, a'\right)\right) | s_t, a_t\right]. \tag{3}$$

However, using the maximal successor action as our learning target can cause a systematic overestimation of action values (Hasselt et al. 2015). This introduces a bias when we maximise over Q-function outputs, and thus an instability in training, leading to learning a poor quality policy.

To negate this, we can instead engage in *off-policy* learning (Mnih et al. 2013): we learn about the target policy μ whilst acting based upon a separate behaviour policy π. Specifically, given the current state s_t, an action is sampled from π; an alternate action is also sampled from μ. The value of this alternate action is then used as the target upon which we update the values of the behaviour policy. In *double Q-learning* (Hasselt 2010), two Q-functions Q^A and Q^B are used to inform our policies, and we probabilistically swap between which is used for the behaviour policy π, and the target μ. With probability $\frac{1}{2}$, Q^B is used to inform μ and calculate the targets y_i, and Q^A calculates the outputs x_i and is subsequently updated; otherwise, the roles of Q^A and Q^B are reversed. Through the combination of two independent functions, we reduce the estimation bias exhibited in the learned Q-values, producing a more accurate and reliable policy.

Hasselt et al. (2015) extend this to *double deep Q-learning* (DDQL). The updated method similarly employs two Q-functions, but each maintains a fixed purpose. State-action valuations x_i are based upon the function Q^π, which we update and follow for the behaviour policy:

$$x_i = Q^\pi \left(s_t, a_t\right), \tag{4}$$

$$y_i = \mathbb{E}_{s_{t+1}} \left[r_{t+1} + \gamma \max_{a' \in A} Q^\mu \left(s_t, a'\right) \right]. \tag{5}$$

Each Q-function is approximated via a neural network, where the parameters of Q^μ are fixed such that we have a stable point upon which the parameters of policy network Q^π can converge. After a given number of training steps, Q^μ is updated to the current values of Q^π, giving a new, more accurate point within the parameter space to converge upon. This iteratively improves the parameter set in an efficient and unbiased manner, improving the behaviour of our agent within its environment, until an optimised behaviour policy π^* is reached (Hasselt et al. 2015).

C Rainbow

In isolation, the aforementioned elements have been repeatedly shown to generate significant performance benefits. However, a recent focus has been on the combination and complementation of these methods, such as *Dueling DDQN* (Wang et al. 2015) which extends the DDQL method to exploit both a state-value function and a state-dependent action-value function. One of the most notable attempts at building one of these composite systems was presented by Hessel et al. (2017), combining extensions across the DRL landscape. Based atop a DQL agent, a total of six extensions addressed its major known limitations. This included the utilisation of DDQL—solving the overestimation problem—and a *prioritised replay* mechanism for sampling transitions of experience, allowing repeated use during training. The produced results showed that these additions significantly boosted performance on all counts, except for the presence of a duelling architecture (Wang et al. 2015) whose inclusion did not show any noticeable benefit.

This proved that composite systems are a viable area of research, generating state-of-the-art results in data-efficiency and performance across the 57 games in the Atari 2600 collection (Hessel et al. 2017). Thus, the hypothesis of constructing a similar composite system for data-efficient learning of effective play in the Super Mario Bros environment holds merit. The conclusions drawn by Hessel et al. (2017) surrounding which methods improve the learning process, and which do not, will also prove helpful in selecting elements for our agent. Specifically, the successes of DDQL and experience prioritisation affirm these as major focuses of this solution.

III SOLUTION

When designing a DRL agent, there are many implementation choices to consider—from neural network architectures to feature encodings. These provide a vast scope for adaptation and refinement of each agent to best suit the challenges of its domain. In this solution, the various design choices made distinctly follow the avenues and themes discussed in the related literature—refining these methods for successful performance in Super Mario Bros.

The structure of the composite agent proposed by Hessel et al. (2017) is followed, combining a collection of DRL methods. Specifically, the agent learns through DDQL, implementing two neural networks for the behaviour and target policies. The architecture of these networks will be discussed and evaluated subsequently. Furthermore, multiple extensions will be introduced, beginning with a basic and prioritised memory buffer, then moving to explore manipulations of the state and action spaces, including the construction of an intrinsic reward system.

The entirety of this solution was developed in Python, utilising *PyTorch* (Paszke et al. 2019) to implement all deep learning elements. This development choice was made as PyTorch is widely regarded as the industry standard for deep learning research, and provides an intuitive framework that is both easily comprehensible and rich in fast and efficient architectures and data structures (Paszke et al. 2019). The game environment will be emulated through a 3rd-party OpenAI Gym extension for Super Mario Bros *gym-super-mario-bros* (Kauten 2018).

A DDQL Agent

As detailed in Section II, DDQL is an off-policy DRL method, employing two networks to learn the best behaviour policy for an agent. DDQL is *model-free*, learning directly from the environment without building its own representation of it. The advantage of model-free learning is predominantly one of computational complexity and efficiency. Due to a model-based agent only being able to learn a policy as good as its model, significant time and memory must be invested for effective modelling of environment dynamics. Thus, to avoid this bottleneck, the presented solution was structured around a model-free approach. When compared to model-based learning, this improves the speed at which we can evaluate the utility of new elements of the system. DDQL was specifically chosen due to its balance between simplicity and stability. Namely, it combined the efficiency of DQL over the state and action spaces, with a reduction in the overestimation of state-action values, improving both the speed and stability of training.

B ε-Greedy Exploration

The DDQL systems selected actions in an ε-greedy manner. Greedy exploration with respect to the function Q simply involves selecting the maximal actions over the action-value outputs corresponding to the current state s_t. ε-greedy exploration surpasses this by introducing an element of randomness in the action selection process, increasing exploration and decreasing the likelihood of the approximator's parameters residing in a local minimum of the loss function. With probability ε a random action is sampled from the action space, and the greedy policy π_Q is followed otherwise.

The policy network employs this iteratively, selecting each action ε-greedily on input of the current state. The action is then employed in the environment, receiving a subsequent reward and successor state, which are immediately exploited during training, updating the network parameters through DDQL. If the state is non-terminal, the agent progresses, repeating this process.

C Training Process

To train this agent, at each iteration i we optimise the parameters θ_i^P of our policy network P by minimising the loss function $L_i\left(\theta_i\right)$ (Equation 6) between its outputs x_i (Equation 4) and the Q-target y_i (Equation 5). The value of y_i is calculated by the target network T.

$$L_i\left(\theta_i^P\right) = \mathbb{E}_{s_t,a_t}\left[loss\left(Y_i^T - x_I^P\right)\right] \tag{6}$$

In the presented solution, the loss function is implemented as the smooth L1 loss (Paszke et al. 2019) between each x_i and y_i. To compute the loss between two inputs x and y, the function L_β relies upon the absolute L1 loss between x and y if the computed value falls above a given threshold β, and the quadratic L2 loss otherwise (where $\beta = 1$):

$$L_\beta(x,y) = \begin{cases} |x-y| - \frac{\beta}{2} & \text{if } |x-y| \geq \beta \\ \frac{(x-y)^2}{2\beta} & \text{otherwise} \end{cases}. \tag{7}$$

This loss function was chosen as it combines the benefits of L2 loss—namely that the gradient of the loss function decreases as the parameters approach a local minimum—and the robustness of L1 loss against outliers. The parameters of the policy network are optimised at every training step—i.e. after the observation of each new transition—whilst the target network parameters are frozen and periodically updated to those of the policy network (every 5000 steps).

D Neural Network Architecture

As with any DRL solution, the design of the constituent neural network is paramount. A tradeoff must be made in the depth and complexity of the network architecture. Larger networks can become more accurate due to their increased number of parameters. However, this is at the expense of training speed, as passes through the internal layers take longer. In our case, due to the goal of learning over frames of gameplay, and the priority of rapid evaluations, this compromise was resolved in favour of faster training times via a small convolutional neural network.

D.1 Convolutional Neural Networks

The large input space associated with employing neural networks on images (i.e all $w \times h \times c$ pixel values) typically requires a large parameter space, making activation computations expensive. To improve efficiency, we can use a *convolutional neural network* (CNN) (LeCun et al. 1998), which constricts the parameter space through a series of *convolutional layers*. For activations in layer L_c^i, a convolution operation is performed by sliding a filter across the set of values in the preceding layer L_c^{i-1}, and computing the dot product between this filter and the values at each point in L_c^{i-1}. This produces an activation map over the neurons in L_c^i of the response of the filter to each point in the image. Through training, the activation function is tuned such that values correspond to the recognition of specific features within the image. This is why CNNs are typically employed in computer vision tasks, such as object detection and visual odometry.

Each convolutional layer is followed by applying a nonlinear activation function, typically the *rectified linear unit* (ReLU) (Paszke et al. 2019)—computed as the maximisation $y = \max(0, x)$. This clips activation values to the range $[0, \infty]$, increasing model sparsity. This results in a concise model—one with an increased likelihood of neurons having zero activation—meaning the remaining positively activated neurons correspond to more identifiable features of the image. Concise models can be trained faster due to the reduction in computations, and reduce overfitting, improving the predictive power of a CNN. Finally, *fully connected layers* are employed to restructure the output to the desired size and shape.

D.2 Final Architecture

The presented solution followed the four-layered CNN architecture proposed by Mnih et al. (2013) for learning over game frames. Input values ($w \times h \times c$ pixels) are propagated through two convolutional layers, the first employing an 8×8 filter, and the second a 4×4 filter. The network concluded with two fully connected layers, reshaping the output to a 1d array over the action space. All layers (except the final) were succeeded by applying the ReLU function.

E Replay Memory

A major issue introduced by the function approximation of DQL is an instability in the training process. The sequential nature through which consecutive frames of gameplay are collected means that the information they contain is highly correlated. This prevents the dataset from being *independent and identically distributed* (i.i.d.)—a necessary condition for stable training. Hence, as immediately training on each transition after they're experienced, consecutive training steps don't use i.i.d. data. Thus, even over many training steps the network's parameters may not converge upon an optimum, instead oscillating or diverging.

E.1 Experience Replay Training

We can encourage our training data to be i.i.d. through employing *experience replay training* (Zhang & Sutton 2017). Instead of immediately learning from each sequential experience e_t, we instead store these transitions into a *replay memory buffer* D (Mnih et al. 2013):

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}), \tag{8}$$

$$D = \{e_0, e_1, e_2, \ldots\}. \tag{9}$$

At each training iteration, n transitions are sampled uniformly at random from this buffer (where n is the *batch size*). The loss function is evaluated by employing the policy network upon the sampled set of states S, and the target network upon the successor states S'. With a large enough buffer, this reduces the impact of correlated data, as consecutive transitions are unlikely to be sampled within the same batch. It also allows the agent to learn from each transition multiple times, utilising experience in a more informed manner. The presented solution implements an experience replay buffer of capacity *30,000* and a batch size of *64*.

E.2 Prioritised Experience Replay

Recent implementations extend this to *prioritised experience replay* (PER) (Schaul et al. 2015), improving the learning process by selecting transitions proportionally to their learning utility. This process uses the error δ_j (Equation 10) associated with each transition e to prioritise experiences according to their utility in optimising the network's parameters θ_π. Each time we optimise θ_π, the transitions in the sampled batch B are utilised to calculate the set of Q-values $\{Q_\pi(s_t, a_t) : \forall s_t, a_t \in e \in B\}$ and targets $\{y_e \colon \forall s_{t+1} \in e \in B\}$. During PER, these values are additionally utilised to calculate the set of absolute errors:

$$\Delta = \{\delta_e \colon \delta_e = |y_e - Q_\pi(s_t, a_t)|, \forall e \in B\}. \tag{10}$$

The priority p_e of each element $e \in B$ is then set proportionally to δ_e. When selecting a batch of transitions, we sample probabilistically from the distribution P, calculated for each of the k elements of B. Each priority is scaled with respect to a prioritisation level α (where $\alpha = 0.6$):

$$P\left(e\right) = \frac{p_e^{\alpha}}{\sum_{k \in B} p_k^{\alpha}}. \tag{11}$$

Due to this non-uniform sampling, the distribution of elements in the sampled batch is not directly representative of the underlying dataset of experience—high priority experiences are over-sampled. To decrease the impact of high-priority samples, the loss function is weighted at each iterative update i as $w_j \cdot L_i\left(\theta_i^{\pi}\right)$, where w_j is inversely proportional to the p_e such that as $P\left(j\right) \to 1, w_j \to 0$. As training progresses, and the parameters of our network approach an optimum, this instability decreases. Thus, weightings are annealed through parameter $\tau = \tau_0 \to 1$:

$$w_j = \left(\frac{1}{N} \cdot \frac{1}{P\left(j\right)}\right)^{\tau}. \tag{12}$$

After first employing basic experience replay, the solution was adapted to utilise a prioritised replay buffer with a capacity of *100,000*. The batch size remained at *64*, and at each iteration of training elements of this batch were collected through the outlined probabilistic sampling method.

F Environment Representation

The representation of the environment's state and action spaces is a crucial factor for consideration in the design of a DRL agent, and one dependent entirely on the structure of the domain. State representation is a simple process in Markovian environments with a fixed state structure: e.g. representing a state in a game of chess as the 8×8 grid containing each player's pieces (Silver et al. 2017a). This becomes more complex in non-Markovian environments, such as computer games, where multiple prior changes to the environment influence the current state. Hence, a major focus of this solution is the manipulation of state representations.

F.1 State and Action Spaces

To best discuss possible manipulations, we must first consider the original construction of the state and action spaces output by the environment. In our case, observations come in the form of $256 \times 240 \times 3$ pixel images. The action space consists of 256 elements, each corresponding to a possible logical combination of controller inputs. The proposed modifications to the state space follow two themes: dimensionality reduction, and data supplementation. Dimensionality reduction aims to abstract away unnecessary data from the state and action spaces to decrease the complexity of a domain, improving the efficiency of an agent. Data supplementation is used to tackle problematic aspects of the environment which impede the learning process.

F.2 Action Dimensionality Reduction

Due to the nature of our action space (i.e. controller inputs), it is clear to see which action combinations are redundant, and constrict the legal combinations played by the agent, reducing the magnitude of the action space. Firstly, this solution banned the combinations of actions that are direct inverses: for example, the instruction of the agent to both move left and move right in a single step. This meant that whenever multiple actions were played in each step, the combination correlated to tangible moves within the environment. The Super Mario Bros manual (Nintendo Entertainment System 1985) outlines the effect of each action on the environment (such as the combination *right* + *A* + *B* performing a *running jump*). From this, the action space can be additionally constricted to only contain combinations useful for our agent to perform.

We can use our prior knowledge of the intended learning aims of the agent to inform further reductions. We know the overarching goal is for the player to progress as far right in the environment as possible. Thus, the subset of actions permitted for movement in the leftwards direction is constricted to a greater extent than the corresponding rightwards set. This allows the agent increased freedom in rightwards movement (compared to leftwards), encouraging the learning of a policy that prioritises progression in that direction, only moving left when absolutely necessary.

F.3 Frame Manipulation

The state space manipulation proposed by this solution follows two major steps, loosely emulating that utilised by Mnih et al. (2013) for the Atari landscape. Initially, we eliminate consecutive environment frames through *frame-skipping*, and secondly, employ a pipeline of downscaling and manipulation upon each selected frame. Thus, a compromise is maintained, retaining the core high-level features of the input whilst reducing the overall data size.

Prior to any manipulation, the process through which the agent receives frames of gameplay is trivial: actions are selected on every frame, providing a fine resolution over temporal events. However, this becomes computationally intensive when mapping high-resolution states to a large action space. Thus, frame-skipping is employed, where the agent only evaluates every k-th frame ($k = 4$ in this solution), generating a single action a_k. This action is repeated over the subsequent k skipped frames, increasing the speed at which episodes are experienced. It is proven that frame-skipping does not impede the asymptotic consistency of the policy being evaluated, and can actually benefit the learned more effective, generalised behaviour (Kalyanakrishnan et al. 2021).

Once a frame is selected, a downscaling pipeline is employed to decrease the magnitude of the input domain of the agent's neural network. Initially, we reduce the colour space from 3-channel $w \times h \times c$ images, to single-channel $w \times h$ images. 30% of the *R* channel is combined with 59% of the *G* channel and 11% of the *B* channel, following the ratio $R\colon G\colon B = 3\colon 6\colon 1$ typically used for approximating the human eye's colour weighting. The resulting frame is downsampled and cropped to 140×104 pixels, decreasing the resolution and removing redundant regions which do not impact the agent. Finally, pixel values are normalised to the range $[0, 1]$ to avoid the impediment to the neural network caused by large integer inputs.

F.4 Intrinsic Reward System

As discussed, two major challenges to learning in Super Mario Bros is the sparse extrinsic rewards and environment traps. The presented solution addresses these hurdles through ingraining the agent with an intrinsic reward system. The system is loosely based upon that of Pathak et al. (2017), however in order to remain model-free, incentives are generated through previously unutilized but informative aspects of the environment. It considers three components: the agent's position within the environment, the status of the agent (with respect to power boost items, described in Section I B), and a bias that heavily favours the goal state.

The position reward r_t^P is motivates the agent to progress as far rightwards from the start state as possible in each step. This is implemented through a position buffer, maintaining the horizontal distance of the agent from the *origin* (leftmost point of the start state). Through comparing the difference in position of consecutive states, the gradient of change of position of the agent over the last n transitions is determined. The agent is given a supplementary reward proportional to the magnitude of this gradient; a positive gradient indicates a rightwards trend, and so is positively rewarded. If the gradient is zero, this indicates the agent has failed to move over the last n steps, either out of choice or due to being constricted by some trap the environment. To minimise the likelihood of the agent choosing to stay stationary, and reduce the valuation of trap states, if consecutive zero gradients are observed the reward signal is negatively decayed exponentially below zero until it returns to being positive—indicating the agent has resumed movement. Thus, the intrinsic signal is designed to promote the maximisation of the position gradient, ensuring the main goal of the agent in every state is to move as far rightwards as possible.

For example, say the buffer follows the n values $[500, 525, \ldots, 655]$ (where elements are appended to the right). If we compare these, we can see that each increases from the last, and thus they exhibit a consistent trend of rightwards motion, inciting a positive reward. However, consider instead if all n of these values were the same. Here we observe a zero gradient, indicating the agent has not moved over the last n time steps. This would generate zero, or negative, reward—depending on how many consecutive zero-gradients have been observed.

As outlined in Section I, the agent can obtain multiple different *statuses*—Super Mario or Fiery Mario—which grant an advantage in the game. Thus, in the presented intrinsic reward model the agent receives a status reward $r_t^S = 10$ for every state within which Mario is either Super or Fiery. This encourages a high valuation of states within which the player's performance is aided by an advantageous status. The final aspect of the intrinsic reward is a bias r_t^G (set as the maximum reward) promoting the valuation of the goal state, solidifying this as the end goal of the agent. These components are consolidated into a single intrinsic reward signal. This signal is then combined with the sparse extrinsic reward from the environment, and the sum normalised to the range $[-15, 15]$ to provide consistency across the state space.

IV RESULTS

To evaluate this solution, the worth of each attribute of the system is analysed. Due to its composite nature, we focus testing on the three core configurable elements: the neural network architecture, experience replay memory, and environment manipulations. All subsequent experiments were undertaken on the University of Durham NVIDIA CUDA Centre GPU cluster.

A Agent Configurations

To analyse the merit of the agent's constituent neural network, an additional CNN was implemented utilising an adapted architecture, highlighting such that the time efficiency and accuracy of results of the original network. Namely, an additional convolutional layer utilising a 2×2 filter was implemented subsequent to the initial pair. The width of concluding fully connected layers was also reduced to illustrate the importance of these concluding operations. Thus the configurable options in terms of network architecture are *original* and *adapted*.

The memory implementation has three configurations, beginning with *basic*—utilising no memory buffer, and immediately training upon single sequential transitions. This baseline is extended to an experience replay system, denoted *replay*. *Prioritised* then extends this to implement prioritised experience replay.

		Network	Memory	Environment
Test 1 (network)	A	0: Original	2: Prioritised	2: Intrinsic Reward
	B	1: Adapted	2: Prioritised	2: Intrinsic Reward
Test 2 (memory)	A	0: Original	0: Basic	2: Intrinsic Reward
	B	0: Original	1: Replay	2: Intrinsic Reward
	C	0: Original	2: Prioritised	2: Intrinsic Reward
Test 3 (environment)	A	0: Original	2: Prioritised	2: Intrinsic Reward
	B	0: Original	2: Prioritised	2: Intrinsic Reward
	C	0: Original	2: Prioritised	2: Intrinsic Reward

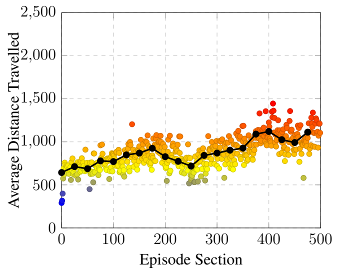
Table 1: Outline of tests to be conducted

The level of environment manipulation similarly has three levels. The *basic* method naively relays each full-resolution $256 \times 240 \times 3$ frame to the agent to select an action from the range $[0, 255]$. The second configuration, termed *reduction*, focuses upon state and action space reductions, downscaling frames to 140×104, and actions to $[0, 8]$. Lastly, *intrinsic reward* implements the proposed intrinsic reward system.

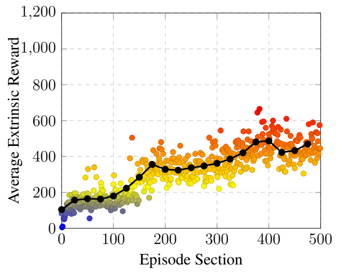
B Evaluation Metrics

For each experiment, three performance metrics are utilised. We first consider the extrinsic reward generated per episode, evaluated through the game score. One may propose additionally evaluating the intrinsic reward; however, this incurs a variety of issues. Firstly, this metric could never be used in tests not relying upon the intrinsic reward system. Secondly, when playing longer episodes, there is an increased opportunity for the agent to become trapped. Thus, generally successful episodes can be overpowered to a negative reward through minor incorrect behaviour.

The second metric used is the distance travelled by Mario over each episode. This is measured as the distance of Mario from the start state at each episodes termination (where the goal state has a distance of 3161). Analysis of the progression of this metric provides another stable representation of the agent's competency, as maximising Mario's final position is a major goal of the player. The final metric is the average training time of the agent. Through comparing the time efficiency of different configurations, we can demonstrate the effectiveness of our efforts towards dimensionality reduction of the state and action spaces.



(a) Test 0B distance (adapted architecture)



(b) Test 0B reward (adapted architecture)

Figure 2: Reward and distance results for adapting the network architecture

To conclude the evaluations, the agent's generalisability will be tested. After being been trained on World 1 Stage 1, the best performing agent configuration will be employed upon subsequent stages, evaluating the distance covered. This will test whether our agent has successfully learned a general policy, which is effective in multiple domains.

C Outcome of Experiments

The experimentation conducted is given in Table 1. Each test focuses on a specific attribute of the system, fixing all other parameters. This commences with the network architecture, moving then to the memory system, and concluding with the environment manipulation. To maintain consistency, all experiments conduct training over $10,000$ episodes, plotting the resulting performance, and the 20-point rolling average through each. For clarity of data, each plotted point corresponds to the average value (of the metric) over a section of 20 training episodes.

C.1 Network Architecture

Figure 2 demonstrates the subtle performance differences between the adapted and original CNN architectures (shown in Figures 3c and 4c). Both networks show a stable increase in reward over time, with a major performance jump seen over the initial 200 range and shallowing of the rate of increase over the final 100 sections. However, the adapted network is superior in both the initial rate of learning, and the final performance. When comparing Figures 2b and 3c, we can see that the adapted network learned to maximise the reward a lot faster over the initial 0–100 range, consistently generating a score of 200 in only 20 episode sections, compared to the initially poor performance of the original network (needing 100 episode sections to do the same).

Moreover, the final performance of the agent utilising the adapted network (with respect to the reward and distance) was marginally increased. The increases to the minimum, maximum, and mean (μ) values of the adapted setup over the last 50 episode sections exemplify this: $\min = 305, \max = 590, \mu = 452$, compared to the $\min = 290, \max = 550, \mu = 422$ of the original network. This trend is also exhibited in the improvement to distance travelled by the agent, increasing the mean to 1237.7 from 1052.3. Thus, it is shown larger CNNs generate better performance in general when learning over frames of gameplay.

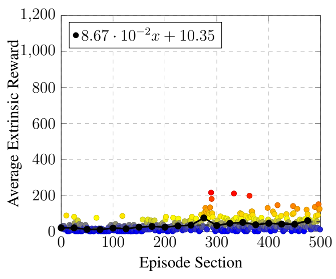
C.2 Memory System

		Distance				Reward			
		min	max	μ	σ	min	max	μ	σ
Test 2A	Initial	295.8	459.4	353.6	37.0	0.0	87.5	18.8	18.7
	Final	313.7	649.4	460.4	87.3	0.0	150.0	48.8	37.6
Test 2B	Initial	333.3	886.1	649.5	172.0	0.0	440.0	110.5	76.1
	Final	1246.0	2263.9	1761.3	213.4	320.0	890.0	585.1	116.9
Test 2C	Initial	313.9	518.7	378.7	37.8	0.0	45.0	19.0	12.6
	Final	849.2	1600.3	1237.7	144.7	290.0	550.0	422.2	68.8

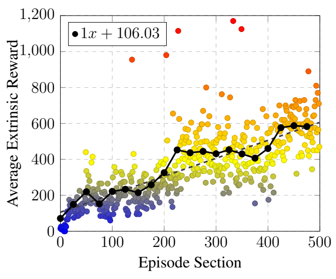
Table 2: Minimum, maximum, mean (μ), and stand deviation (σ) for all memory tests between the initial 50 episode sections and final 50

Test 2 evaluates the memory system, the results demonstrated in Figures 4 and 3. When using no memory buffer, the agent only marginally improves its performance over training, with the majority of episodes exhibiting a distance below 750. The minimum, maximum, and mean values (Table 2) demonstrate that performance is improving slightly over training. However, this trend is very shallow, as the mean distance travelled per episode only increased by 30.2%.

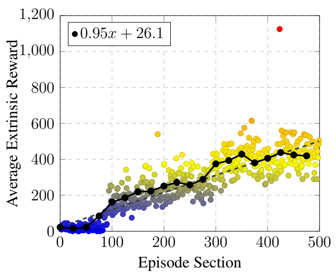
Upon introducing experience replay, a significant increase in performance was demonstrated to both the reward and distance (Figures 3b and 4b), rapidly improving performance over the initial part of training. In both cases, as training progresses the rate of improvement decreases.



(a) Test 1A (no memory)



(b) Test 1B (basic memory)



(c) Test 1C (prioritised memory)

Figure 3: Reward results of memory configurations

However, the trend of the agent consistently improving distance travelled and reward generated is maintained, exemplified by the increase in minimum, maximum, and mean values (Table 2). Thus, the significant performance benefit of experience replay is demonstrated, allowing the agent to successfully learn to maximise the extrinsic reward and distance travelled per episode.

The prioritised experience replay based agent also demonstrates this improving performance (Figures 3c and 4c). However, the extent of this success, in comparison to its preceding replay method, is surprising. Upon inspection of the trend of both the reward and distance, the rate of increase is less than that of the uniform replay method. This is exhibited in both the initial rate of learning, where a significant performance increase is only demonstrated after 100 episode sections, and the trend which succeeds it, exhibiting a noticeably shallower gradient.

These results show that the most effective memory implementation for an agent learning how to play Super Mario Bros is a non-prioritised experience replay buffer, with uniform sampling. These results were surprising, and demonstrate the novelty of our learning goal.

C.3 Environment Manipulation

Through experimentation, the benefit provided by the introduction of space and action space reductions, in comparison to learning over every full-resolution frame, is clear. Whilst Test 3B was able to complete the 10,000-episode training process in 1760 minutes, the baseline Test 3A was unable to complete training—taking 3150 minutes to evaluate only 24 episodes. Through extrapolating these results, we can deduce that to complete the allocated training period the unmodified agent would have needed over 1.3 million minutes, the equivalent of 911 days. This demonstrates the inefficiency of training over the raw state and action spaces, and the importance of our state and action space reductions, which successfully reduced the necessary training time to a manageable level. Furthermore, it is clear the reduced fidelity of the state space did not impede the agent's ability to learn behaviour, as the results associated with the reduction-based agent demonstrate that it is able to improve its performance in terms of both reward and distance.

However, Test 3B also demonstrates the challenges of the sparsely rewarding environment. The agent takes a large amount of experience to begin learning, and once this learning begins it quickly plateaus: 200 episode sections are needed until an increase to either the reward or distance is apparent. This is followed by a period of rapid increase over the range 220–250, improving the agent's performance. However, in both cases the rolling average indicates performance plateaus after this point, concluding training with a mean distance of 705 and reward of 159, indicating that the agent is struggling to learn beyond this point. This suggests the period within which the agent is learning to maximise its position and reward is very narrow over the training process when the extrinsic reward signal is solely used to inform the agent's behaviour.

The performance advantages of incentivising the agent through an intrinsic reward are demonstrated in the preceding Figures 3c and 3c. The agent's initial rate of learning is significantly improved, showing a period of rapid increase in both score and distance comparable to that of the 220–250 region of the extrinsically motivated agent significantly earlier, within the first 100 episode sections of training. Furthermore, the final values of both the reward and distance are consistently higher: the mean distance over the final 50 episode sections is increased by 267% (from 159 to 585) and reward 149% (from 705 to 1761). This proves that the intrinsic reward signal did motivate the agent to both cross further through the level, and generate more reward, preventing the bottleneck which occurs when solely learning from an extrinsic signal.

V EVALUATION

In this section, an evaluation is made of the strengths and limitations of the experimentation, quantifying the worth of each element of the composite system. Thus, the most effective combination of methods over our domain can be devised through the correlation between the successful results, and an evaluation of the system as a whole can be conducted.

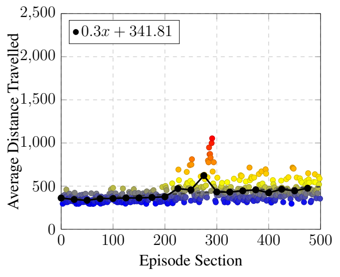
A Network Evaluation

Figure 2 demonstrates how the additional convolutional layers of the adapted network generated an increased performance with respect to both extrinsic reward and distance travelled, exhibited through the initial rate of learning and final results. Although this performance gap may partly be due to the stochasticity of the environment, it is likely the increased reduction of the pixel space facilitated through the additional convolutional layer aided the adapted network. This suggests the additional parameters increased the accuracy of feature recognition within the scene, tuning behaviour with respect to these features more rapidly. However, the stagnation of results between the episode sections 200–300 may suggest that these parameters became overfit to the challenges early in the environment. This is likely due to the increased parameter count, as this early stagnation is not observed in the smaller, original architecture.

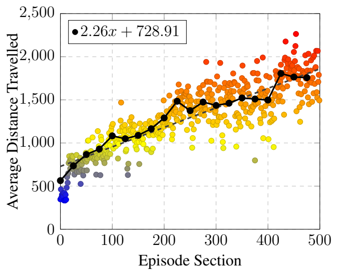
The core advantage of the original architecture is it's temporal efficiently, completing the $10{,}000$ training episodes in 866 minutes, compared to the 1856 minutes of the adapted network. Putting the 7.11% increase to mean score into the context of this 2.14 times longer training time, it is clear the original network trains considerably more efficiently. As the goal of experimentation is to compose an agent which not only generates effective performance, but does this in a data and time efficient manner, a compromise must be maintained in configuring the best agent. Thus, the adapted architecture was discarded, and the 7.11% performance decrease accepted, in favour of the increased time efficiency of the original network.

B Memory Evaluation

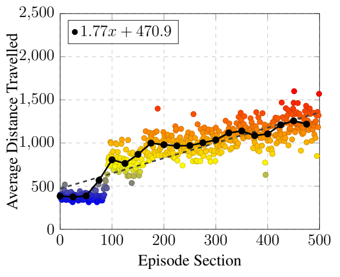
The performance increase introduced by using experience replay is exemplified through a comparison of the gradient of the linear regression lines plotted through the distance results of Figure 4. The agent utilising no memory buffer generates a very shallow trend, with a gradient of 0.3, whereas the replay method demonstrates a significantly steeper increase in average distance per episode section, with a gradient of 2.26. Furthermore, the mean extrinsic reward generated



(a) Test 1A (no memory)



(b) Test 1B (basic memory)



(c) Test 1C (prioritised memory)

Figure 4: Distance results of memory configurations

over over the final 50 episode sections was 1098.9% higher, and mean distance increased by 282.5%. These results demonstrate the significant advantage provided by improving upon the detrimentally naive method of directly learning from transitions. The poor performance of this technique is likely due to the non-i.i.d. data, meaning an unbiased representation of the true gradient of the loss function cannot be generated during backpropagation.

The PER-based agent also demonstrated this increased performance over the baseline method; the mean reward increased by 765.1%, and mean distance by 168.8%. However, in comparison to the basic replay method, these values are unexpectedly low. Namely, the mean reward produced using PER was only 72.1% of that of the uniform method, and the distance only 70.2%. This reduced performance is further exhibited in the gradients of the regression lines plotted through Figures 3 and 4. For the reward, these gradients are relatively similar (0.95 for PER, compared to 1 for basic replay). However, when evaluating the distance plot, the uniform memory system noticeably outperforms the PER-based agent—exhibiting a gradient of 2.26 compared to 1.77.

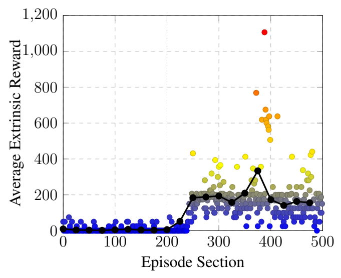
When proposing PER, Schaul et al. (2015) discussed the environment-dependent nature of its benefit. Over the Atari collection, they found it provided no performance benefit in 10 of the 49 games tested. A further 8 games exhibited reduced performance, including Tutankham. Tutankham demonstrates a similar goal to that of our agent—navigating a maze in order to find keys and shoot down enemies—possibly explaining the underwhelming performance of PER in Super Mario Bros. It is likely that the challenge of learning complex prolonged action sequences (typical of both games) is is not one suited to PER. Furthermore, the initially poor performance of the agent implies the PER system may be prioritising certain experiences too much, and overfitting the network to the monotonous, naive transitions which fill the buffer early on during training, reducing the effectiveness in later challenging parts of the environment.

Figures 3 and 4 demonstrate one notable advantage of the PER-based agent. Both reward and distance results in the latter stages of training are less varied, in comparison to the basic replay method. Over the final 50 episode sections, the standard deviation of the distance travelled by the PER agent is only 58.8% of its uniform counterpart, and the extrinsic reward only 67.8%. This suggests that whilst the overall behaviour produced by the experience replay method is superior, the PER agent generates more consistent, precise performance.

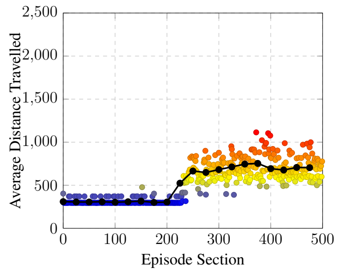
Evaluating the time efficiency of each memory method unsurprising demonstrates that the addition of complexity to the memory system increases training time. Whilst the agent utilising no memory buffer took 378 minutes to complete training, the additional computations associated with experience replay increased this to 686 minutes. However, the significantly increased performance—utilising 63 more transitions per training iteration to generate 1098.9% greater mean reward—justifies this 81.4% increased training time. When using PER, this training time increased to 866 minutes, due to the increased computational complexity associated with prioritisation. However, due to the efficient search and update operations provided by storing priorities within a binary Sum Tree, this impact is only a marginal 26.2% increase over uniform replay. Thus, the temporal efficiency of the overall system is preserved. However, due to the performance benefits of the basic experience replay buffer, the conclusion is drawn that this is the most effective memory method (of those explored) for the chosen environment.

C Environment Evaluation

The performance increases provided by the downscaling operations and intrinsic motivator are evident. By representing states through only 7.89% of the original number of pixel values (reducing frame resolution from $240 \times 256 \times 3$ to 104×140), and constricting the action space to only 3.51% of its original size (from 256 to 9), the computational challenge of each Q-function evaluation was significantly reduced. This improved the time efficiency of the agent, allowing training over significantly more episodes, taking only 0.133% of the time needed by the baseline method for evaluating the initial 24 episodes.



(a) Test 3B reward (space reductions)



(b) Test 3B distance (space reductions)

Figure 5: Reward and distance results of environment dimensionality reduction

However, an analysis of Figure 5 demonstrates how the agent failed to learn successful behaviour over the first 220 episode sections. By comparing this to the PER and intrinsic reward based agent shown in Figures 3c and 4c, it suggests this poor initial performance may partly be due to the PER memory, as both systems exhibit an initial lack of learning. However, learning solely from extrinsic rewards has drastically heightened the impact of this plateau, causing the agent to take 2.44 times longer to overcome it and begin learning from experience. Furthermore, the variation of reward results generated through purely extrinsic reward-based learning is higher than when the presented intrinsic reward it utilised. Comparing Figures 3c and 4c to Figure 5 shows how learning solely through extrinsic rewards demonstrates a higher standard deviation across training, peaking at 181.3 between the range 350–450, compared to only 116.9 for the intrinsic reward based agent over the same range—a 55.1% increase. This suggests that when the agent is additionally motivated by the presented intrinsic reward signal, the final performance of the agent is not only higher (increasing the maximum distance by 126.7% and reward 101.9%), but more consistent, meaning the training process is more stable and effective.

D Overall Solution and Approach

Through the preceding experimentation, the agent which generates the most successful behaviour, in the most time and data efficient manner, can be deduced. Figures 4b and 3b confirm this to be the DDQL agent using experience replay with uniform sampling, state and action space reductions, and the intrinsic reward system. This agent generated a maximal average distance travelled within the environment of 2264, and mean average distance of 1761 (over the 20-episode sections). Given the distance between the start and goal states is 3161, this corresponds to the agent crossing 71.6% and 55.7% of the stage respectively. Additionally, when observing the fully trained agent, successful games within which the agent reached the goal state were repeatedly witnessed (a success which is lost in the averages plotted).

To put this success in context, the few existing Super Mario Bros based agents should be considered. The presented agent outperforms its most notable competitor in this field: the curiosity-driven agent of Pathak et al. (2017). On World 1 Stage 1 this method learned to cross 30% of the level (with no extrinsic reward), compared to the 71.6% and 55.7% figures demonstrated by the presented agent. Furthermore, the presented system undercuts the computational complexity of the curiosity-driven method, which relied upon two dynamics models—a forwards predictor of actions, and an inverse model for learning the feature space—to inform its intrinsic reward system. It also allowed our agent to train over less episodes of gameplay, requiring only 10,000 (on average corresponding to 1 million training iterations) in comparison to the 10–20 million training iterations necessary for the model-based approach. Similarly, the complexity associated with the generative adversarial network inspired approach of Engelsvoll et al. (2020)—which employed a neural network to learn a dynamics model for constructing game stages—was avoided.

One of the major goals of this project was learning *generalised* play. To test this, the maximally performing trained agent was deployed into a variety of different game stages, recording the resulting distance travelled by Mario over 100 test episodes. The agent was found to perform best on the initial stages of World 2, generating a maximum distance of 1015 on Stage 1, 1440 on 2, and 1325 on 3 (corresponding to 32.1%, 45.5%, and 41.9% coverage). These stages had a common theme; they were all visually analogous to the training stage (Figure 1c), maintaining a similar appearance and structure. However, in stages that significantly visually differ, the agent did not show such impressive performance: a maximum distance of 675 was achieved in World 1 Stage 2 (21.3% coverage). This suggests that the main limiting factor of generalised play is the extraction of features from each frame. However, given the impressive performance on the aforementioned stages of World 2, a competent level of generalisation has been achieved.

VI CONCLUSIONS

In this project, a DRL agent—utilising Double Deep Q-Learning, experience replay, state and action space manipulations, and an intrinsic reward signal—has been devised that can successfully learn to play the game Super Mario Bros. Through demonstrating a peak performance of crossing 71.6% of the training level (over a 20-episode average), this paper proves that it is possible to successfully apply DRL methods devised for the Atari game collection to this environment. This improves upon the performance of the main intrinsic-reward based method devised for learning Super Mario Bros gameplay, presented by Pathak et al. (2017), which managed to cross 30% of World 1 Stage 1 (without extrinsic rewards). This has been achieved completely model-free, reducing the computational complexity associated with other DRL agents of this domain. Moreover, the method generalised well to other stages of the environment with visually similar features to those of the training stage, managing to cross 45.5% of World 2 Stage 2.

Further work should surround increasing the performance of this generalisation. The agent's neural network architecture should be adapted to improve the recognition of general features. The utilisation of *Noisy Nets* may aid in these efforts. These networks introduce a noisy linear layer combining a deterministic stream with a noisy input channel, improving their ability to learn complex state-conditional action sequences. This would aid the agent in simulating the complex strategies necessary to evade environment traps in Super Mario Bros, and facilitate the learning of more objective action values. Furthermore, the efforts towards time and data efficiency could be extended through utilising *divide-and-conquer decomposition* (Barreto et al. 2020). To handle the vast amounts of data inherent in DRL problems, the learning process is decomposed into a set of smaller *tasks*, upon which we produce a feature vector with respect to a corresponding reward function (composed of reward functions of other solved tasks). Analogous to *dynamic programming*, this optimises learning over large state and action spaces by breaking the learning problem into a sequence of easily solvable steps—improving efficiency.

References

Arulkumaran, K., Deisenroth, M. P., Brundage, M. & Bharath, A. A. (2017), 'Deep Reinforcement Learning: A Brief Survey', *IEEE Signal Processing Magazine* **34**(6), 26–38.

Barreto, A., Hou, S., Borsa, D., Silver, D. & Precup, D. (2020), 'Fast reinforcement learning with generalized policy updates', *Proceedings of the National Academy of Sciences* **117**(48), 30079–30087.

Engelsvoll, R. N., Gammelsrød, A. & Thoresen, B. I. S. (2020), Generating Levels and Playing Super Mario Bros. with Deep Reinforcement Learning Using various techniques for level generation and Deep Q-Networks for playing (Master's thesis), Master's thesis, University of Agder, University of Agder, Grimstad.

Hasselt, H. v. (2010), Double q-learning, *in* 'Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2', NIPS'10, Curran Associates Inc., Red Hook, NY, USA, pp. 2613–2621.

Hasselt, H. v., Guez, A. & Silver, D. (2015), 'Deep Reinforcement Learning with Double Q-learning', *arXiv e-prints* p. arXiv:1509.06461.

Hessel, M., Modayil, J., Hasselt, H. v., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. & Silver, D. (2017), 'Rainbow: Combining Improvements in Deep Reinforcement Learning', *arXiv e-prints* p. arXiv:1710.02298.

IGN (2012), 'The History of Mario: A look in Mario's roots may help gamers see Nintendo's famous mascot within a bigger framework'.
URL: *https://www.ign.com/articles/1996/10/01/the-history-of-mario*

Kalyanakrishnan, S., Aravindan, S., Bagdawat, V., Bhatt, V., Goka, H., Gupta, A., Krishna, K. & Piratla, V. (2021), 'An Analysis of Frame-skipping in Reinforcement Learning', *arXiv e-prints* p. arXiv:2102.03718.

Kauten, C. (2018), 'Super Mario Bros for OpenAI Gym', GitHub.
URL: *https://github.com/Kautenja/gym-super-mario-bros*

LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998), 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE* **86**(11), 2278–2324.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013), 'Playing Atari with Deep Reinforcement Learning', *arXiv e-prints* p. arXiv:1312.5602.

Nintendo Entertainment System (1985), 'Super Mario Bros: Instruction Booklet'.
URL: *https://www.nintendo.co.jp/clv/manuals/en/pdf/CLV-P-NAAAE.pdf*

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019), PyTorch: An Imperative Style, High-Performance Deep Learning Library, *in* H. Wallach and H. Larochelle and A. Beygelzimer and F. d'Alché-Buc and E. Fox and R. Garnett, ed., 'Advances in Neural Information Processing Systems 32', Curran Associates, Inc., pp. 8024–8035.

Pathak, D., Agrawal, P., Efros, A. A. & Darrell, T. (2017), 'Curiosity-driven Exploration by Self-supervised Prediction', *arXiv e-prints* p. arXiv:1705.05363.

Puigdomènech Badia, A., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D. & Blundell, C. (2020), 'Agent57: Outperforming the Atari Human Benchmark', *arXiv e-prints* p. arXiv:2003.13350.

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015), 'Prioritized Experience Replay', *arXiv e-prints* p. arXiv:1511.05952.

Shao, K., Tang, Z., Zhu, Y., Li, N. & Zhao, D. (2019), 'A Survey of Deep Reinforcement Learning in Video Games', *arXiv e-prints* p. arXiv:1912.10944.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. & Hassabis, D. (2017a), 'Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm', *arXiv e-prints* p. arXiv:1712.01815.

Tesauro, G. (1995), 'Temporal Difference Learning and TD-Gammon', *Commun. ACM* **38**(3), 58–68.
URL: *https://doi.org/10.1145/203330.203343*

Wang, Z., Schaul, T., Hessel, M., Hasselt, H. v., Lanctot, M. & de Freitas, N. (2015), 'Dueling Network Architectures for Deep Reinforcement Learning', *arXiv e-prints* p. arXiv:1511.06581.

Zhang, S. & Sutton, R. S. (2017), 'A Deeper Look at Experience Replay', *arXiv e-prints* p. arXiv:1712.01275.