Evaluating MuZero's performance using Super Mario Bros

Sreeharsha Udayashankar
s2udayas@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Abstract

An important application area of artificial intelligence lies in the development of intelligent agents, capable of playing games at skill levels comparable to humans. The first intelligent agents to perform such feats were model-based and designed to play fixed-rule games such as BackGammon, Go and Chess. These agents have long been specialized and limited to the specific game they were designed to target. For example, a model capable of playing chess would not fare very well if tasked with playing another game such as Go.

Recent model-based approaches such as AlphaZero [24] have evolved past this limitation and demonstrate impressive results across a variety of games. However, these approaches were still limited as they could only handle fixed-rule games such as Chess and Go. The newest development in this area is MuZero [23], which could not only handle fixed-rule games just like AlphaZero [24], but could handle visually rich environments with seemingly no rule-sets as well. Model-free algorithms such as Deep-Q-Learning [18] were the only approaches thought to be able to handle such environments till date.

The goal of this project is to evaluate MuZero on the visually rich game Super Mario Bros [15], to see if it can hold up its impressive record when compared to state-of-the-art model-free algorithms.

Introduction

Playing games has been a major application area for research in both machine learning and artificial intelligence. Algorithms have been devised for a variety of games including BackGammon [28], Chess [5, 1], Go [7, 30] and Bridge. Traditional techniques for this include Alpha-Beta Search [20, 16], Minimax Search [22, 21] and Monte-Carlo Tree Search [6]. However, in recent times, reinforcement learning has been applied to play numerous games including Chess [23, 24, 29], Go [25, 24, 23], BackGammon [28] and various games from the Atari 2600 Gaming Library [17, 23].

Reinforcement learning can be broadly divided into 2 subcategories: model-based and model-free learning approaches. Some examples of model-based approaches include DYNA [27], Prioritized Sweep [19] and ARTDP [2]. Model-free approaches to reinforcement learning such as Deep-Q-Networks [17] and R2D2 [14] have also been gaining steadily in popularity and usage.

Google's DeepMind project announced AlphaZero [24], a model-based reinforcement learning approach (Ref), in late 2017 which was able to master a variety of games such as Chess, Go and Shogi with a single reinforcement-learning based approach. They evaluated it by pitting against the world-champion programs and humans in each respective game. For example, AlphaZero played 1000 games against Stockfish 8 [1], winning 155 of them and losing only 6 [24]. It had similar impressive results against Stockfish 9 as well. In Go, it was pitted against Crazy Stone [7] and Zen [30] as well as the human European Champion, Fan Hui. It scored impressively against each of these opponents as well [24]. AlphaZero was one of the first programs to show that a single generic approach could work in different environments with different rules, a huge step towards general AI at the time.

Subsequently, Google has now developed MuZero [23], which can extend the planning approaches of AlphaZero to accommodate new scenarios such as playing Atari Games, which were previously restricted to model-free reinforcement learning approaches. MuZero has been evaluated on Chess, Go, Shogi and the Atari 2600 benchmark [23, 3]. MuZero outperforms or matches AlphaZero in each of the three games AlphaZero was previously evaluated on [23] and goes ahead to massively outperform model-free approaches such as DQN and R2D2 in the Atari Gaming Benchmark [23].

The goal of my research project is to determine whether MuZero can hold up the same impressive performance record against DQN on the popular game Super Mario Bros. Previously, agents trained to play Super Mario Bros have utilized Deep-Q Networks [17] and Double Deep-Q Networks [11] to play through individual levels and have outperformed other model-based approaches which do so.

Project Summary

1. **Research Problem:** Evaluation of MuZero on Super Mario Bros and comparison against DQN with Double-Q-Learning

2. **Main Algorithms Implemented / Modified:** MuZero and Deep-Q-Networks with Double-Q-Learning
3. **Evaluation:** Compare in-game scores on an overworld training level.

This project has been carried out using the public MuZero implementation on Github [10] as well as the super-mario-bros environment [15] for OpenAI Gym [4]. A custom DQN [17] implementation with Double-Q-Learning [11] has been used as the baseline during evaluation. Both algorithms have been trained on the same overworld level.

Evaluation Details

1. **Performance Metric:** The performance metric used to evaluate these algorithms is the in-game score at the end of the episode. This score is a function of various factors such as the time taken to complete the level (if completed), the number of special blocks hit as well as the number of monsters killed while traversing the level.
2. **Environment Features:** The models have been trained and evaluated using the OpenAI Gym [4] environment. In order to simulate Super Mario Bros, a previously designed plugin for this environment, the super-mario-bros package [15] built on top of the nes-py emulator using Python 3.7, have been used.

Main Results

The key takeaway from this paper is that MuZero [23] appears to provide performance comparable to that of model-free algorithms on visually rich environments such as Super Mario Bros. This falls in line with it's previous impressive performance record over model-free algorithms on the Atari ALE [23, 3].

However, in terms of computational performance, MuZero [23] is far more computationally intensive than Double Deep-Q-Networks [11], as seen in the Results section. However, these results might just be a function of the MuZero implementation [10], and might improve with targeted performance optimization measures. These were beyond the scope of this project.

Contributions

The contributions of this paper are:

- Evaluation and Comparison of MuZero [23] and Double Deep-Q-Networks [11] on Super Mario Bros
- Modification of the MuZero [10] repository to handle working with Super Mario Bros
- The implementation of a custom Deep-Q-Network with Double-Q learning [11] to play Super Mario Bros
- Hyperparameter optimization for MuZero [23] and Double-Deep-Q-Networks [18, 11] to target the Super Mario Bros [15] environment

Related Work

Training agents to play games is a well-explored research area. One of the earliest examples in this regard was TDGammon [28] , an artificial neural network developed by IBM to play BackGammon. It was trained using temporal difference learning [28] and achieved a level of play which was slightly below those of top human players at the time. Future efforts to extend this approach to other board games [13] had limited success.

When playing games such as Chess, early reinforcement learning approaches such as NeuroChess [29] have been considered inferior to search-based approaches. For instance, NeuroChess was a neural network trained using temporal difference learning and it only won 13% of its games against GNUChess [9]. Deep Blue [5], which was the first engine to beat human world champion Garry Kasparov in 1997 used a massively parallel search-based approach to evaluate positions and determine the optimal choice of moves. Up until recently, many engines such as StockFish [1] and Hydra [12] utilized similar search based approaches with improvements such as Alpha-Beta Searching [20, 16]. Similarly, Crazy Stone [7] utilized a Monte-Carlo search based approach to play the popular game Go.

However, search-based approaches only work in cases where the rules of the system are known or an accurate simulator can be interacted with. Model-based reinforcement learning algorithms attempt to first estimate the dynamics of the environment and then use this to plan ahead. An example of this is PILCO [8]. However, in visually rich environments like those used in Atari 2600 games [3], model-based approaches fall behind model-free approaches such as Deep-Q-Networks [17] and Recurrent Replay Distributed Deep-Q-Networks [14]. These approaches estimate the optimal policy and value functions directly by interacting with the environment. On the flip side, model-free approaches tend to perform poorly in scenarios such as Chess and Go.

In late 2017, Google's DeepMind released AlphaGo [25], a program which utilized Monte-Carlo tree searches as well as deep neural networks. AlphaGo uses the following components:

1. A fast rollout policy p_π
2. A policy network SL trained using supervised learning based on expert human moves in a variety of Go positions (p_σ)
3. A policy network RL, initially set to SL and later improved using reinforcement learning and playing multiple games against itself (p_ρ)
4. A value network v_θ

The policy network takes as input a representation of the board. It then passes this through multiple convolutional layers which use p_σ and p_ρ as weight parameters and outputs a probability distribution for all possible legal moves in the given position. The value network similarly takes the board representation as input and estimates the probability of winning the game from a given position. AlphaGo combines these policy and value networks using Monte-Carlo tree searches. The tree is traversed completely without backing up and leaf node expansion is carried out by a combination of p_σ, the value network v_θ and the fast rollout policy p_π and combined into an evaluation using the mixing parameter λ.

AlphaGo was able to beat the state-of-the-art search-based program Crazy Stone [7] as well as the European Champion Fan Hui in a series of games. Google later extended this approach to AlphaZero [24], a program capable of playing Go, Chess and Shogi using the same combined approach. AlphaZero had a couple of differences when compared to AlphaGo:

- It estimated the outcome of a given position (win / loss / draw) vs simply calculating the probability of a win like AlphaGo.
- The rules of Go were symmetric and this property was used to boost the training data set by rotating and reflecting positions. The rules of chess are asymmetric, for instance pawns cannot move backwards. AlphaZero therefore does not assume symmetry.
- AlphaZero maintains a single neural network that is updated continually and used for self play games vs maintaining copies of multiple previous iterations to form a self-play pool like AlphaGo did.

AlphaZero had significant success and could beat AlphaGo at Go, Stockfish 8 and 9 at Chess and Elmo at Shogi [24]. Google further built upon the AlphaZero protocol to develop MuZero [23]. MuZero utilizes the planning based approach of AlphaGo and AlphaZero but does not need access to an accurate simulator or need to know some rules of the game. It instead models just the aspects which are important to an agent's decision making process. Several of AlphaZero's components which relied on the accurate simulator / rulebook have been replaced with a single neural network. After evaluation on the Atari 2600 benchmark [3] using the Atari Arcade Learning Environment (ALE) [3], MuZero outperformed model-free algorithms such as DQN [17] and R2D2 [14] and was the new state-of-the-art on the Atari benchmark.

Methodology

This section outlines the approach adhered to by this paper. The goal is to compare MuZero [23] and DQN [17] using the game Super Mario Bros [15]. The following paragraphs describe each algorithm in further detail.

MuZero

MuZero [23] was developed by Google's DeepMind in late 2020. It builds on the approach taken by AlphaZero [24] and AlphaGo [25] and extends it to visually rich environments such as Atari 2600 games, run via the Atari Arcade Learning Environment [3]. A key improvement over AlphaZero was that MuZero could master environments with unknown dynamics and did not need to know the rules beforehand, which was previously only done using model-free learning approaches. The architecture of MuZero is shown in Figure 1.

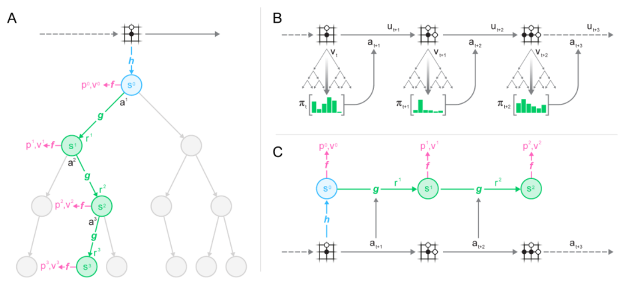


Figure 1: Muzero. This figure has been taken from the MuZero [23] paper and is owned by the original authors.

MuZero is a planning based algorithm which uses Monte-Carlo Tree Searches and deep neural networks. As shown in Figure 1A, MuZero consists of 3 connected components: dynamics, prediction and representation functions. Given a hidden state s_{k-1} and a potential action a_k, the dynamics function generates an immediate reward r_k and a new state s_k. The prediction function then calculates the policy p_k and value function v_k from the new hidden state s_k. The representation function is used to generate the initial hidden state s0 by taking the board state or Atari game state as input.

At each timestep t, the algorithm makes predictions for each of the K steps ranging from $1....K$. These predictions are made by a model u_θ where θ are parameters conditioned on past observations $o_1, o_2....o_t$ and future actions $a_t, a_{t+1}....a_{t+k}$. The model predicts the following quantities:

- The policy $p_t^k \approx \pi(a_{t+k+1}|o_1, ..., o_t, a_{t+1}, ..., a_{t+k})$
- The value function $v_t^k \approx E[u_{t+k+1} + \gamma u_{t+k+2} + ...|o_1, ..., o_t, a_{t+1}, ..., a_{t+k}]$
- The immediate reward $r_t^k \approx u_{t+k}$

where u is the true observed reward, π is the policy used to select real actions, and γ is the discount function of the environment.

The internal representation at each timestep consists of the dynamics, prediction and representation functions as described above. Given this model, it is possible to search over hypothetical future trajectories $a_1, ..., a_k$ given past observations $o_1, ..., o_t$. At each timestep t in the environment, the algorithm executes a Monte-Carlo Tree Search as shown in Figure 1B. An action a_{t+1} is sampled from the policy π_t, sent to the environment and observation o_{t+1} and reward u_{t+1} are generated. At the end of the episode, this trajectory is stored into the replay buffer.

As shown in Figure 1C, when training the model, a trajectory is sampled from the replay buffer, the observations $o_1, o_2...o_t$ are passed to the representation function h. The model is unrolled for K steps and each step receives the action a_{t+k} and hidden state s^{k-1}. The parameters of the dynamics, prediction and representation functions are subsequently jointly trained via backpropagation.

The overall loss function with L2 regularization is given by:

$$l_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, p_t^k) +$$

$c||\theta||^2$
where l^r, l^v, and l^p are loss functions for reward, value and policy respectively.

Deep-Q-Networks and Double-Q-Learning

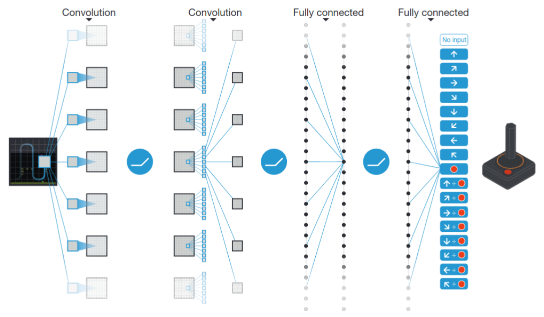


Figure 2: Deep-Q-Network Structure for Atari 2600. This figure has been taken from the DQN paper [17] and is owned by the original authors.

Deep-Q-Networks [17] are a model-free approach designed to provide human-like control in visually rich environments. Figure 2 shows the structure of a sample Deep-Q-Network used to play Atari 2600 games. This approach uses a deep convolutional network to approximate the optimal value function:

$$Q^*(s,a) = \max_{\{\pi\}}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} ... | s_t = s, a_t = a, \pi|]$$

which is the maximum sum of rewards r_t discounted by γ at each time-step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation s and taking an action a.

The paper uses experience replay i.e. selecting a random set of experiences during Q-updates, which removes correlations in the observation sequence and smooths over the data distribution. They also use an iterative update which only adjusts Q towards target values which are periodically updated. These methods are used to reduce instabilities which can arise in Q-updates due to correlations in observational data.

They parameterize a Q function $Q(s, a; \theta i$ using the convolutional network where θ are the weights of the Q-network at iteration i. The agent's experiences at each time step t are stored in a replay buffer in the format $e_t = (s_t, a_t, r_t, s_{t+1})$. During training, Q-learning is applied on a minibatch of samples chosen uniformly at random from all the experiences in the stored pool. The Q-learning at iteration i uses the following loss function:

$$L_i(\theta_i) = E(s,a,r,s) \approx U(D)[(r + \gamma \max_{a'} Q(s', a'; \theta_i^- - Q(s, a; \theta_i)^2]$$

in which γ is the discount factor determining the agent's horizon, θ_i are the parameters of the Q-network at iteration θ_i^- are the network parameters used to compute the target at iteration i.

They evaluated the Deep-Q-Network on the Atari 2600 benchmark and it achieved more than 75% of a professional human tester's score across more than half of the games. It was also better than a linear approximator on a majority of the games [17].

However, Hasselt et al [11] demonstrated that this algorithm can be overly optimistic, overestimating action values under certain conditions. They evaluated this behavior on the Atari 2600 [3] benchmark and showed that it can result in suboptimal performance. They then proposed a few specific changes to the DQN [17] algorithm and proposed the Double-Q-Learning [11] approach, which performed significantly better in comparison to the DQN [17] approach on the Atari [3] benchmark.

Super Mario Bros

Super Mario Bros [26] is a 1985 arcade platform game developed and published by Nintendo. Players assume the role of Mario and the aim is to traverse multiple levels consisting of various obstacles, traps and monsters. These obstacles can range from static walls to secret pipes which can tunnel players to other levels. Levels can be of two kinds: Overworld levels and Underwater levels. Underwater levels have different movement mechanics and are considered to be out of the scope of this project.

Each level can consist of multiple hazards such as pits and moving enemies, as well as power-ups such as the Mushroom and the Fire Flower which can significantly aid players in level completion. The collective set of actions taken by a player throughout the level, including destroying enemy monsters and picking up power-ups, contribute to their overall score for the level, displayed on the top right of the screen by convention.

Evaluation Environment

Evaluation Framework In order to train and evaluate the algorithms, the OpenAI Gym framework [4] has been used. Super Mario Bros has been ported to this framework in the form of the super-mario-bros [15] package which has been used as well. MuZero [23, 10] and DQN [17] implementations have been modified / rewritten in order to work with this interface. DQN with Double-Q-Learning [11] has been selected as the primary baseline to evaluate MuZero's performance as DQNs have provided solid erformance in visually rich environments such as on the Atari 2600 benchmark [17, 11].

The algorithms' performances have been compared using the training score (a.k.a the reward) attained on a selected overworld level after a fixed number of training episodes. They have both been trained on the same overworld level.

Hardware and Software Setup Both algorithms have been trained and evaluated on the same machine with the following characteristics:

- **CPU:** AMD Ryzen 5800X
- **GPU:** Nvidia RTX 3060
- **Memory:** 16 Gigabytes

- **Nvidia Driver and CUDA Versions:** nvidia-driver-495 and CUDA 11.5
- **Operating System:** Ubuntu 20.04.3 LTS (Focal Fossa)
- **Python:** Python 3.8.10
- **PyTorch:** PyTorch v1.10.0
- **TensorFlow:** Tensorflow v2.7.0

Results

Deep-Q-Networks with Double-Q-Learning

A custom DQN with Double-Q-Learning has been implemented using PyTorch to play Super Mario Bros. The class diagram for our implementation has been shown below and a few key functions have been described in detail.

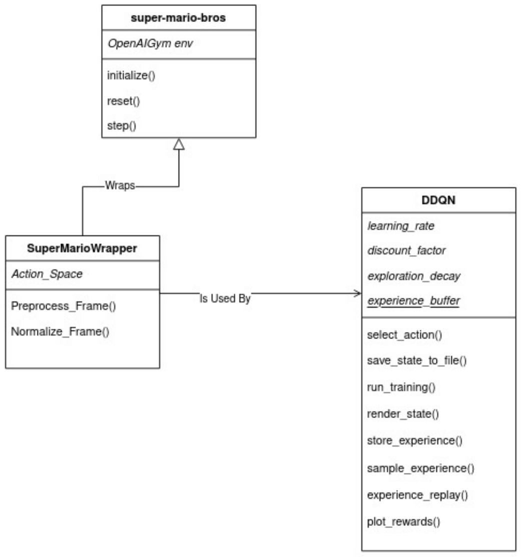


Figure 3: Double Deep-Q-Learning for Super Mario - Class Diagram

The code structure is described below:

Class - super-mario-bros This is the super-mario-bros base package [15] which provides access to the SuperMario Bros [26] environment via an OpenAI Gym [4] interface. It contains the underlying environment parameters as well as functions such as reset(), step() and initialize() which perform functions as outlined in the Gym [4] documentation.

Class - SuperMarioWrapper This is a class which wraps around the base super-mario-bros class and contains the following parameters:

- *Action Space:* The total action space accessible by the agent. For the scope of this project, we have limited the action space to RIGHT_ONLY and disabled more complex movement patterns. This action space consists of 5 actions including Jump and Move Right.
- *Preprocess_Frame():* This converts the incoming frame into a 96x96 greyscale frame to be used as part of the input state.
- *Normalize_Frame():* Normalizes the pixel values within a frame to fall within the range 0 to 255.

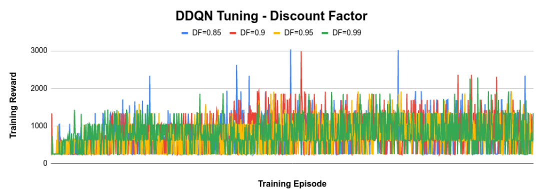
Class - DDQN This is the core class that holds the hyper-parameters for the Double Deep-Q-Network as well as the experience buffer. It also contains all functions relevant to DQN operations, described below:

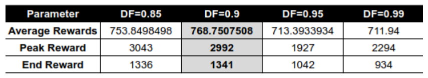
- *Learning Rate, Discount Factor and Exploration Decay:* Hyperparameters used to train Double Deep-Q-Networks as mentioned in previous literature [11, 18].
- *Experience Buffer:* Buffer used to hold experiences from which a random batch of experiences is sampled during learning. Each experience is of the form *(state_1, action, reward, state_2, terminate* where $state\_1$ and $state\_2$ represent the initial and final states, $action$ represents the action taken. $reward$ represents the reward obtained and $terminate$ indicates whether the episode was terminated after this experience.
- *select_action():* Function to select the next action based on the greedy policy, as outlined in [18, 11].
- *save_state_to_file():* Saves the network as a pickled file, in order to reload it later for testing if necessary. This function also dumps the rewards obtained per episode during training into a JSON file for further analysis.
- *run_training():* Trains the model for the specified number of training episodes.
- *render_state():* This is to render the environment state into a human readable form during testing.
- *store_experience():* Saves an experience in the *Experience Buffer* for sampling at a later time.
- *sample_experience():* Randomly sample a batch of experiences from the *Experience Buffer*.
- *experience_replay():* 'Replays' a set of sample experiences, updating network parameters and performing Q-Learning.
- *plot_rewards():* Visualizes the training rewards obtained per episode.

Deep-Q-Networks with Double-Q-Learning: Hyper-parameter Tuning

We have examined three distinct hyper-parameters for Double Deep-Q-Networks and tuned their values. Due to computational power limitations, 1000 training episodes were run for each hyper-parameter value and the results have been outlined below. For detailed descriptions of each of the hyper-parameters, please refer to the original papers [18, 11].

Discount Factor We have examined three distinct values for the discount factor γ. The graph of training rewards vs training episodes has been shown below along with a tabulation of key statistics from these episodes.



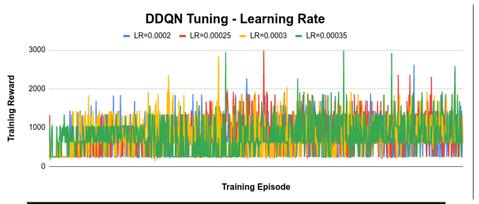


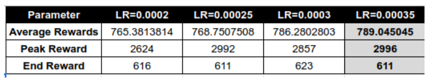
Parameter	DF=0.85	DF=0.9	DF=0.95	DF=0.99
Average Rewards	753.8498498	**768.7507508**	713.3933934	711.94
Peak Reward	3043	**2992**	1927	2294
End Reward	1336	**1341**	1042	934

Figure 4: Deep-Q-Networks - Discount Factor Tuning

As we can see from the results of Figure 4, the optimal value for the discount factor $\gamma = 0.9$. While $\gamma = 0.85$ produces similar results, this value produces a slightly higher average training reward. Higher values of γ such as 0.95 and 0.99 make the performance significantly worse.

Learning Rate We have examined four distinct values for the learning rate α. The graph of training rewards vs training episodes has been shown below along with a tabulation of key statistics from these episodes.



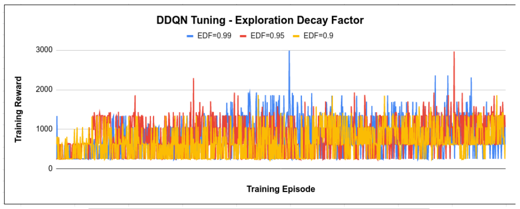


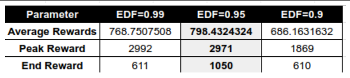
Parameter	LR=0.0002	LR=0.00025	LR=0.0003	LR=0.00035
Average Rewards	765.3813814	768.7507508	786.2802803	**789.045045**
Peak Reward	2624	2992	2857	**2996**
End Reward	616	611	623	**611**

Figure 5: Deep-Q-Networks - Learning Rate Tuning

As we can see from the results of Figure 5, the optimal value for the learning rate $\alpha = 0.00035$. While other values also produce similar results, this value produces a slightly higher average training reward.

Exploration Decay We have examined three distinct values for the exploration decay factor. The graph of training rewards vs training episodes has been shown below along with a tabulation of key statistics from these episodes.





Parameter	EDF=0.99	EDF=0.95	EDF=0.9
Average Rewards	768.7507508	**798.4324324**	686.1631632
Peak Reward	2992	**2971**	1869
End Reward	611	**1050**	610

Figure 6: Deep-Q-Networks - Exponential Decay Factor Tuning

As we can see from the results of Figure 6, the optimal value for the exponential decay factor is 0.95. While other values also produce similar results, this value produces a slightly higher average training reward.

MuZero

The MuZero implementation on Github [10] uses TensorFlow. It can handle simple OpenAI Gym [4] based games such as the Cartpole [4] game. This implementation has been modified heavily to enable it to work with Super Mario Bros. The code structure for the modified implementation has been shown below:

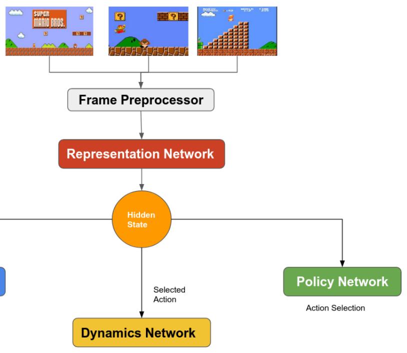


Figure 7: MuZero for SuperMarioBros - Components

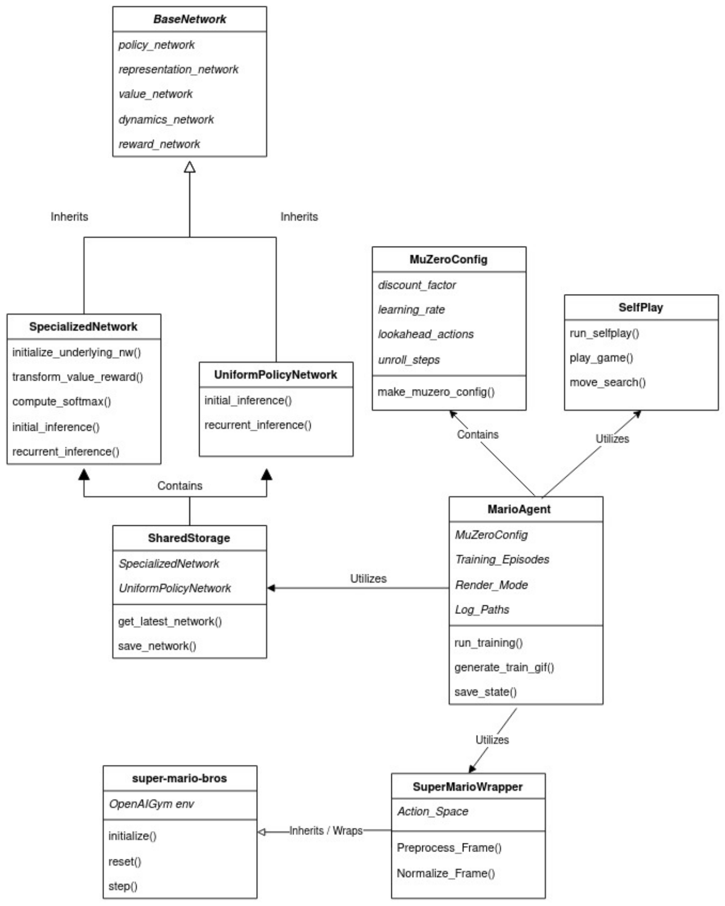


Figure 8: MuZero for SuperMarioBros - Class Diagram

Class - super-mario-bros This is the super-mario-bros base package [15] which provides access to the SuperMario Bros [26] environment via an OpenAI Gym [4] interface. It contains the underlying environment parameters as well as functions such as reset(), step() and initialize() which perform functions as outlined in the Gym [4] documentation.

Class - SuperMarioWrapper This is a class which wraps around the base super-mario-bros class and contains the following parameters:

- *Action Space:* The total action space accessible by the agent. For the scope of this project, we have limited the action space to RIGHT_ONLY and disabled more complex movement patterns. This action space consists of 5 actions including Jump and Move Right.
- *Preprocess_Frame():* This converts the incoming frame into a 96x96 greyscale frame to be used as part of the input state.
- *Normalize_Frame():* Normalizes the pixel values within a frame to fall within the range 0 to 255.

Class - BaseNetwork This is the BaseNetwork class from the MuZero Github repository [10]. It contains the underlying policy, reward, value, dynamics and representation networks.

Class - SpecializedNetwork This type of network is tailored to the SuperMarioBros environment and has been custom implemented. It has the following functions:

- *initialize_underlying_nw():* This function creates and initializes the policy, value, reward, dynamics and representation networks from the *BaseNetwork* class.
- *transform_value_reward():* Transformation function for rewards and values.
- *compute_softmax():* Function to compute softmax values for actions efficiently.
- *initial_inference() and recurrent_inference():* Functions consistent with the expected network representation within the Muzero repository [10].

Class - UniformPolicyNetwork This type of network assigns uniform values to all possible actions i.e. represents a uniform policy. This network is used to fill the experience buffer for MuZero with a few episodes at the start of the training session. In our experiments, we have used this *UniformPolicyNetwork* to save up to 100 episodes into the replay buffer at the start of training sessions. The same experiences have been used across all hyper-parameter value tuning training runs as well.

Class - SharedStorage This class is a modified version of the *SharedStorage* class from the public MuZero repository [10]. It stores multiple copies of each kind of network it holds. It has been modified to hold the new *SpecializedNetwork* along with the *UniformPolicyNetwork*. Its functions are as follows:

- *get_latest_network():* Returns the latest updated network from storage.
- *save_network():* Saves the network from the specified time step into a pickled file for retrieval and usage later.

Class - MuZeroConfig Holds the hyperparameters and configuration info for the underlying MuZero algorithm. This is a modified version of the *MuZeroConfig* class within the public repository [10].

Class - SelfPlay Contains functions to run training episodes, populate the experience replay buffer, update the underlying networks as well as search for the optimal move using the specified networks' policy.

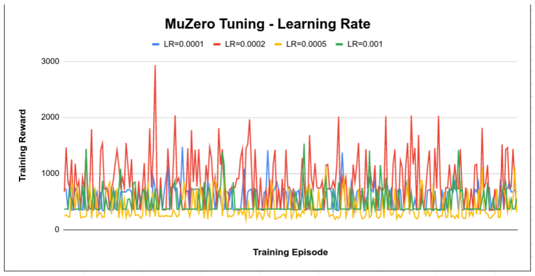
Class - MarioAgent This is the main driver class for the entire program. It holds an instance of the *MuZeroConfig* class as well as other parameters such as the number of training episodes to run. It interacts with various modules such as the *SuperMarioWrapper*, *SharedStorage* and the *SelfPlay* modules. Its functions are as follows:

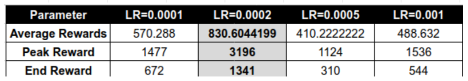
- *run_training():* Runs the specified number of training episodes using the *SelfPlay* and *SharedStorage* modules.
- *generate_train_gif():* Runs the specified number of training episodes using the *SelfPlay* and *SharedStorage* modules and generates human readable GIFs showing the actions taken in each episode.
- *save_state()*: Saves states to the specified log directory paths utilizing save functions within other modules.

MuZero: Hyper-parameter Tuning

We have examined and tuned four hyper-parameters for MuZero with SuperMarioBros. The following subsections outline our efforts on each hyper-parameter in detail. For detailed descriptions of the hyper-parameters, please refer to the original MuZero paper [23]. As MuZero was computationally intensive, 500 training episodes have been run for each hyper-parameter value.

Learning Rate We have examined four distinct values for the learning rate α. The graph of training rewards vs training episodes has been shown below along with a tabulation of key statistics from these episodes.



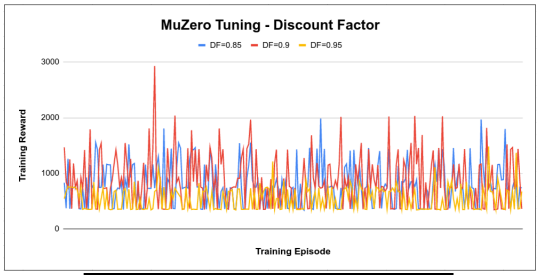


Parameter	LR=0.0001	LR=0.0002	LR=0.0005	LR=0.001
Average Rewards	570.288	**830.6044199**	410.2222222	488.632
Peak Reward	1477	**3196**	1124	1536
End Reward	672	**1341**	310	544

Figure 9: MuZero - Learning Rate Tuning

As we can see from the results of Figure 9, the optimal value for the learning rate $\alpha = 0.0002$. This value produces the best average and peak training rewards.

Discount Factor We have examined three distinct values for the discount factor γ. The graph of training rewards vs training episodes has been shown below along with a tabulation of key statistics from these episodes.

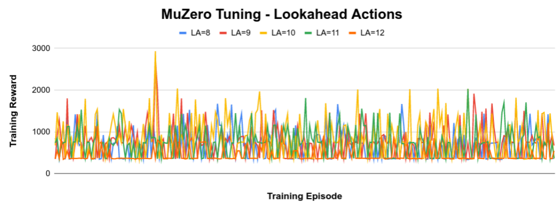


Parameter	DF=0.85	DF=0.9	DF=0.95
Average Rewards	791.8634538	**830.7787611**	538.6506024
Peak Reward	1986	**3196**	1483
End Reward	1341	**1179**	672

Figure 10: MuZero - Discount Factor Tuning

As we can see from the results of Figure 12, the optimal value for the learning rate $\gamma = 0.9$. This value produces the best average and peak training rewards.

Number of Look-ahead Actions We have examined five distinct values for the number of look-ahead actions. The graph of training rewards vs training episodes has been shown below along with a tabulation of key statistics from these episodes.

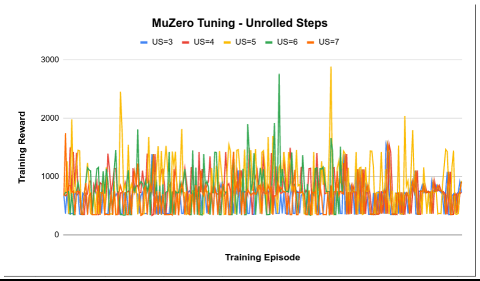


Parameter	LA=8	LA=9	LA=10	LA=11	LA=12
Average Rewards	667.6385542	784.4457831	**830.7787611**	736.9558233	451.9598394
Peak Reward	1672	2725	**3196**	2028	1210
End Reward	361	672	**865**	369	544

Figure 11: MuZero - Lookahead Actions Tuning

As we can see from the results of Figure 12, the optimal value for the number of look-ahead actions is 10. This value produces the best average and peak training rewards.

Number of Unrolled Steps We have examined five distinct values for the number of unrolled steps. The graph of training rewards vs training episodes has been shown below along with a tabulation of key statistics from these episodes.



Parameter	US=3	US=4	US=5	US=6	US=7
Average Rewards	607.2371542	779.7215686	**828.0547689**	755.511811	648.3937008
Peak Reward	1547	1547	**2897**	2763	1744
End Reward	370	672	**1444**	705	705

Figure 12: MuZero - Unrolled Steps Tuning

As we can see from the results of Figure 12, the optimal value for the number of unrolled steps is 5. This value produces the best average and peak training rewards.

MuZero vs Deep-Q-Networks with Double-Q Learning

Computational Performance Comparison

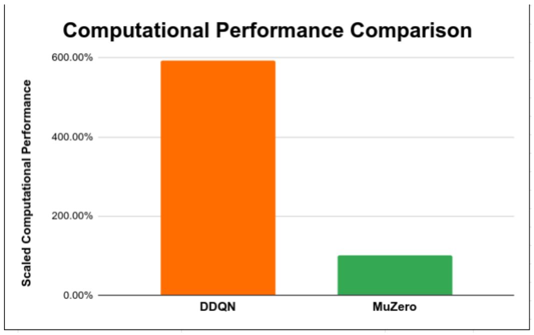


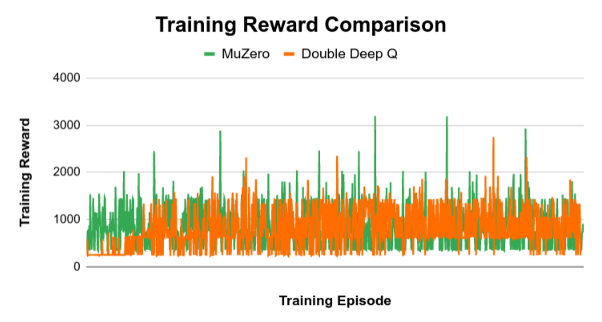
Figure 13: MuZero vs DDQN - Computational Performance

We analyzed the computational performance of both algorithms by recording the amount of time taken to train 2500 episodes of each algorithm. We have presented our results in Figure 13, which contains both algorithm's performances scaled to that of MuZero as a baseline (100%). From the graph, we can see that Double Deep Q Learning is much more computationally efficient than MuZero and it takes nearly 6 times the amount of time for MuZero to go through the same number of episodes as Double Deep Q Networks. This had a significant impact on our experiment as we were restricted to a lower total number of training episodes on MuZero due to computational restrictions.

This performance difference could be attributed to multiple factors such as:

- The computational nature of the algorithms themselves
- Double Deep Q Networks were implemented using PyTorch whereas the public MuZero implementation [10] uses TensorFlow
- The public MuZero implementation [10] may need to be optimized for performance. While we have put some effort into this, a full scale performance optimization of this repository was beyond the scope of this project.

Evaluation and Comparison on Super Mario Bros



Parameter	MuZero	DDQN
Average Rewards	830.808	812.001
Peak Reward	3196	2751
End Reward	352	606

Figure 14: MuZero vs DDQN

We have trained the modified MuZero implementation as well as the Double Deep Q Networks implementation using Super Mario Bros for 3000 episodes each and presented our results in Figure 14. We can make the following observations from the graphs:

- MuZero and Double Deep Q Networks achieve similar training performance across the 3000 episodes they were run for. MuZero performs very slightly better in terms of average reward per training episode and peak rewards.
- Both algorithms seem to have similar increases in average reward over time

However, it must be noted that training MuZero took nearly 6 times as long as training Double Deep Q Networks, as mentioned above.

Discussion

Result Analysis The goal of this project was to examine if MuZero [23] held up its impressive performance record against DQNs on the Atari ALE benchmark [3, 23] on other visually rich environments as well. Our examination reveals that this indeed does appear to be the case, as we can see that MuZero performs similar to if not slightly better than Deep-Q-Networks with Double-Q-Learning [11]. Over the 3000 training episodes that were run on each algorithm, the average and peak training rewards achieved by MuZero seem to be similar to those of the Double Deep-Q-Network. This falls in line with previous literature [23] where it initially demonstrated its performance record on the Atari ALE benchmark [3].

However, it is worth mentioning that the computational performance of these algorithms has not been analyzed in previous literature. Our analysis shows that Deep-Q-Networks with Double-Q-Learning are significantly more computationally efficient than MuZero. Further performance analysis is necessary to confirm if the performance gap observed is merely a result of an unoptimized implementation or a function of the algorithms themselves, as noted in our results.

Limitations There were a few limitations in our analysis which have been outlined below:

- Due to limited available computational resources, we were unable to evaluate all the possible hyper-parameters of MuZero and Double Deep-Q-Networks. We acknowledge that this might not be the best obtainable performance for these algorithms.
- Due to the limited available computational resources and the duration of this project, we were unable to run our evaluation on multiple overworld levels and were restricted to looking at a single level during training. Algorithmic performance might differ on other levels, especially those with different mechanics such as underwater levels.

Conclusion

In this project, we have attempted to train and evaluate the model-based MuZero algorithm's [23] performance on the popular game Super Mario Bros [26]. We have also compared its performance to Deep-Q-Networks with Double-Q-Learning, the state of the art model-free algorithm traditionally used with visually rich environments. We have implemented a custom Deep-Q network for this task using PyTorch and modified the public MuZero implementation [10] in order to do so.

The key takeaway from this project is that MuZero does indeed achieve comparable performance to that of Deep-Q-Networks on Super Mario Bros. However, the computational performance of each algorithm is vastly different and further examination is required into the cause of this gap. The following research directions are possible for this project:

- Further examination of hyper-parameters for MuZero and DDQN
- Evaluation using multiple overworld training and testing levels
- Performance comparison on underwater levels with different mechanics
- Investigation into the large computational cost differences between MuZero and Double-Deep-Q-Networks

References

[1] *About - Stockfish - Open Source Chess Engine*. URL: https://stockfishchess.org/about/ (visited on 10/07/2021).

[2] Andrew G. Barto. "Adaptive Real-Time Dynamic Programming". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 19–22. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_10. URL: https://doi.org/10.1007/978-0-387-30164-8_10 (visited on 10/07/2021).

[3] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[4] Greg Brockman et al. *OpenAI Gym*. _eprint: arXiv:1606.01540. 2016.

[5] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291.

[6] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI". In: *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AIIDE'08. event-place: Stanford, California. AAAI Press, 2008, pp. 216–217.

[7] *Crazy Stone*. URL: https://www.remi-coulom.fr/CrazyStone/ (visited on 10/07/2021).

[8] Marc Peter Deisenroth and Carl Edward Rasmussen. "PILCO: A Model-Based and Data-Efficient Approach to Policy Search". In: *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*. Ed. by Lise Getoor and Tobias Scheffer. Omnipress, 2011, pp. 465–472. URL: https://icml.cc/2011/papers/323%5C_icmlpaper.pdf (visited on 10/07/2021).

[9] *GNU Chess - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/software/chess/ (visited on 10/07/2021).

[10] Johan Gras. *MuZero*. original-date: 2019-12-08T22:23:25Z. Dec. 2021. URL: https://github.com/johan-gras/MuZero (visited on 12/13/2021).

[11] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *arXiv:1509.06461 [cs]* (Dec. 8, 2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461 (visited on 10/07/2021).

[12] *Hydra (chess)*. In: *Wikipedia*. Page Version ID: 952013554. Apr. 20, 2020. URL: https://en.wikipedia.org/w/index.php?title=Hydra_(chess)&oldid=952013554 (visited on 10/07/2021).

[13] Ghory Imran. "Reinforcement Learning in Board Games". In: *Reinforcement Learning in Board Games* (2004). Publisher: University of Bristol.

[14] Steven Kapturowski et al. "Recurrent Experience Replay in Distributed Reinforcement Learning". In: International Conference on Learning Representations. Sept. 27, 2018. URL: https://openreview.net/forum?id=r1lyTjAqYX (visited on 10/07/2021).

[15] Christian Kauten. *gym-super-mario-bros: Super Mario Bros. for OpenAI Gym*. Version 7.3.2. URL: https://github.com/Kautenja/gym-super-mario-bros (visited on 10/07/2021).

[16] Donald E. Knuth and Ronald W. Moore. "An analysis of alpha-beta pruning". In: *Artificial Intelligence* 6.4 (Dec. 1, 1975), pp. 293–326. ISSN: 0004-3702. DOI: 10.1016/0004-3702(75)90019-3. URL: https://www.sciencedirect.com/science/article/pii/0004370275900193 (visited on 10/07/2021).

[17] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015). Bandiera_abtest: a Cg_type: Nature Research Journals Number: 7540 Primary_atype: Research Publisher: Nature Publishing Group Subject_term: Computer science Subject_term_id: computer-science, pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236 (visited on 10/07/2021).

[18] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv:1312.5602 [cs]* (Dec. 2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602 (visited on 12/13/2021).

[19] Andrew W. Moore and Christopher G. Atkeson. "Prioritized sweeping: Reinforcement learning with less data and less time". In: *Machine Learning* 13.1 (Oct. 1, 1993), pp. 103–130. ISSN: 1573-0565. DOI: 10.1007/BF00993104. URL: https://doi.org/10.1007/BF00993104 (visited on 10/07/2021).

[20] M. M. Newborn. "The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores". In: *Artificial Intelligence* 8.2 (Apr. 1, 1977), pp. 137–153. ISSN: 0004-3702. DOI: 10.1016/0004-3702(77)90017-0. URL: https://www.sciencedirect.com/science/article/pii/0004370277900170 (visited on 10/07/2021).

[21] Judea Pearl. “Asymptotic Properties of Minimax Trees and Game-Searching Procedures”. In: *Artif. Intell.* 14.2 (1980), pp. 113–138. DOI: 10.1016/0004-3702(80)90037-5.

[22] Aske Plaat et al. “A New Paradigm for Minimax Search”. In: *arXiv:1404.1515 [cs]* (Apr. 5, 2014). arXiv: 1404.1515. URL: http://arxiv.org/abs/1404.1515 (visited on 10/07/2021).

[23] Julian Schrittwieser et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”. In: *Nature* 588.7839 (Dec. 24, 2020), pp. 604–609. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-03051-4. arXiv: 1911.08265. URL: http://arxiv.org/abs/1911.08265 (visited on 10/07/2021).

[24] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *arXiv:1712.01815 [cs]* (Dec. 5, 2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815 (visited on 10/07/2021).

[25] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676 (Oct. 2017). Bandiera_abtest: a Cg_type: Nature Research Journals Number: 7676 Primary_atype: Research Publisher: Nature Publishing Group Subject_term: Computational science;Computer science;Reward Subject_term_id: computational-science;computer-science;reward, pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://www.nature.com/articles/nature24270 (visited on 10/07/2021).

[26] *Super Mario Bros.* en. Page Version ID: 1060081112. Dec. 2021. URL: https://en.wikipedia.org/w/index.php?title=Super_Mario_Bros.&oldid=1060081112 (visited on 12/13/2021).

[27] Richard S. Sutton. “Dyna, an integrated architecture for learning, planning, and reacting”. In: *ACM SIGART Bulletin* 2.4 (July 1991), pp. 160–163. ISSN: 0163-5719. DOI: 10.1145/122344.122377. URL: https://dl.acm.org/doi/10.1145/122344.122377 (visited on 10/07/2021).

[28] Gerald Tesauro. “Temporal Difference Learning and TD-Gammon”. In: *Commun. ACM* 38.3 (Mar. 1995). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: https://doi.org/10.1145/203330.203343.

[29] Sebastian Thrun. “Learning to Play the Game of Chess”. In: *Proceedings of the 7th International Conference on Neural Information Processing Systems*. NIPS’94. event-place: Denver, Colorado. Cambridge, MA, USA: MIT Press, 1994, pp. 1069–1076.

[30] *Zen (go program) at Sensei’s Library*. URL: https://senseis.xmp.net/?ZenGoProgram (visited on 10/07/2021).