COMP 596 Report

Training an AI Agent that plays Super Mario Bros using Reinforcement Learning Techniques

***Abstract*— Traditionally, playing and testing games has always been a labor-intensive task. In video games, with just a couple simple rules, a video game can become extremely complex to most simple computers. It was difficult to automate computers to play most video games. Thanks to the recent advancement of Reinforcement Learning algorithms and development of OpenAI Gym, it is possible to automate machines to play complex video games. In this project, we aimed to explore various Reinforcement Learning techniques to train agents to play the 1985 Nintendo game Super Mario Bros. Our agents are convolutional neural network based models that have been trained with raw frames captured from the game and the basic info such as scoring scheme and possible actions for the player. The agent learns the control policies from raw pixel data using deep reinforcement learning techniques.**

Keywords: **Super Mario Bros, Artificial Neural Networks, Reinforcement Learning, Algorithms, Q-Learning, Game Playing, Machine Learning, Deep, Convolutional Neural Network, Proximal Policy Optimization, Advantage Actor Critic.**

I. INTRODUCTION

Many Machine Learning (ML) enthusiasts and gamers have used interesting Artificial Intelligence based techniques to train computers to play different games. However, the implementation of such tasks was not always an easy task because of the complexity of the game. Also, the lack of big datasets in a video game makes it impossible to train a model using traditional supervised learning techniques.

To the researchers, Deep Reinforcement Learning (Deep-RL) [1] is one of many powerful tools available to researchers in the machine learning field, which is the most appropriate approach to play video games automatically. Deep Reinforcement Learning is capable of training agents with very little user generated data. This is because for many domains all that is required from the user is an adequately specified reward function and not a huge dataset of gameplay videos. This generality makes Deep-RL appropriate for tackling domains with highly variable environments that present state spaces as large as usually found in video games. For example, in an average Super Mario Bros. [2] game level, the player may be at any of 3000 x-positions at any time within a few thousand timesteps, during which they may take any of 16 actions (e.g. move left/right, jump, fire etc.) that can change its state. These environments are analogous to real world problems because of these complexities.

As a big fan of the Super Mario Bros. game and having an interest in learning more about Reinforcement Learning, we aim to create a project with one goal in mind: applying reinforcement learning algorithm to teach computer to play Super Mario Bros. game without any human supervision [3]. This report is structured as follows. First, we explain the background of the chosen techniques, followed by the approaches taken to implement the project. Then, the results are shown at the end for analysis and comparisons. Finally, we discuss the challenges and the future work necessary to elevate this project for further success.

II. RELATED WORKS

Reinforcement learning (RL) and deep learning approaches have become standard for generating game-playing agents and have been utilized to play a wide range of games [4], which includes not only arcade games but also first-person shooter games.
Each genre in games industry has its own difficulties; certain tactics are more effective in some contexts than others. Therefore, it is vital to investigate which playtesting methods have been used in different genres of games that are comparable. Deep Q Learning (DQN) [5] and actor-critic methods such as Proximal Policy Optimization (PPO) [6] are cutting-edge techniques for Atari games that use memory-features or pixel data.

For platform games, researchers have also applied some other techniques. For example, the original DQN algorithm suffers from overestimation problem in function approximation. A modification of DQN algorithm was proposed in [7], and termed it as Deep RL with Double Q-learning (DDQN), a technique that would double check the Q learning, which was able to solve the overestimation problem for function approximation at large scale. Robert Clark in [8] applied the DQN and DDQN combined with Convolutional Neural Networks (CNN) [9] techniques to develop a working model that can play Super Mario Bros. Students from Stanford University [10] had also done a project on using Q learning on abstract Mario state vector of a game environment.

III. Technical Background

A. *Game Mechanics*

Super Mario Bros. has been a very popular video game during the late 1980s. It was developed and published by the Nintendo company [2]. In the Super Mario Bros. game, the player controls a character Mario and must put him through different challenges that is finally intended to save the Princess of Mushroom Kingdom in the end. The goal of the game is to just move the character to the right and get to the end of each stage. The player can move to the left up to a certain point, but once they start to move towards left, the camera won't follow. When the player moves to the right, the camera follows the character so that it stays in the center. If the player goes all the way to the left and then back to the right, the camera won't stay in the middle of the player until it reaches the center point again.

On each level, there are many different obstacles, some of which don't seem dangerous int the first place, like blocks that stick out of the ground or are placed above the player. However, depending on the location of obstacles, they can stop the player from moving forward. Since each stage has a time limit, this could threaten Mario to lose when time run out.

When the player comes across an obstacle like this for the first time, it usually needs to jump over it to avoid it. The most common thing to do in the game, other than running and walking, is to jump. When you jump, you can get past obstacles, kill enemies, and get power-ups. When jumping, players can change their position while in the air so they can land in the spot they want.

The main goal of the game is for Mario to get to each stage's ending point (a flag post) without losing all of his life chances. The second goal is to get the highest possible score, which you can do by killing rivals, collecting coins, and finishing a level as quickly as possible. Figure 1 shows a typical game screenshot (level 1) for the Super Mario Bros. game.

Fig. 1. An example of Mario jumping over an obstacle

B. *Reinforcement Learning Background*

Machine learning models are taught through reinforcement learning to make a series of decisions. The result of that learning process is an agent. The agent learns how to reach a goal in an environment that is uncertain and could be hard to learn. For each next time step in training, the agent gets a delayed reward or penalty that indicates how did its last action turn out. In reinforcement learning, an AI is put in a situation like a game, which is a great way to teach it how to play a video game. The computer tries different things until it finds the right answer. The agent would not receive any hints or rules during the learning process. The only resource it has is the defined reward function. Its goal is to get as much reward as possible during experience.

What makes Reinforcement learning different than supervised learning and unsupervised learning is RL does not necessarily require training data. Instead, RL generates its own data by applying different actions in the environments, multiple time. It's up to the agent to figure out how to do the task to get the most reward, starting with completely random trials and ending with complex strategies and skills that most people are unable to do.. Reinforcement learning is currently the best way to help a machine be creative. It does this by using the power of search and many trials. Unlike humans, artificial intelligence can learn from playing thousands of games at the same time, if a reinforcement learning algorithm is run on a computer system with enough power.

Reinforcement learning has many different algorithms, but they mainly fall under two categories: model-based and model-free. The model-free algorithms consist of two types: value based and policy-based algorithms. An example of value-based algorithm is Deep Q Network (DQN). A common policy-based algorithm is Proximal Policy Optimization (PPO). An Actor-Critic algorithm has the characteristics of both policy-based and value-based combined.

Even though there are a lot of RL algorithms, it doesn't look like there exists a good way to compare all of them. It made it hard for researchers and engineers to choose which algorithms to use for a certain task. The goal of this project is to solve this problem by talking briefly about RL using the same environment set up and code implementation, and giving an overview of some of the popular techniques applicable in the Super Mario Bros. game.

C. *Deep Q Network (DQN)*

DeepMind [11] came up with the advent of DQN algorithm in 2015. By combining RL with deep neural networks (DNNs) at a large scale, it was able to solve a wide range of Atari games (some of them even to the level of a superhuman). The algorithm was created by adding DNNs and a method called "experience replay" [12]to a classic RL algorithm called "Q-Learning."

Q learning is a value-based algorithm, which updates a value based on current state and action. That value, called Q value, is the memory of the action's consequence. If the consequence is positive, it is a reward. If the consequence is negative, it is a penalty. Each Q value is mapped to a state action combination.

When Q learning involves techniques of Neural Network to assist in learning, we have Deep Q Learning or Deep Q Network. Deep Q Learning often utilize a technique called experience replay during training. Experience Replay is a technique of RL that makes use of a replay memory. It stores the agent's experiences at each time step and adds them up over many episodes to make a replay memory. Together, DQN and experience replay are used to store the steps of an episode in memory for off-policy learning, where random samples are taken from the replay memory. Also, the Q-Network is usually optimized towards a fixed target network that is updated with the latest weights on a regular basis. This makes it more stable because short-term frequent switching of values caused by a moving target are stopped.

D. *Proximal Policy Optimization (PPO)*

The Proximal Policy Optimization (PPO) [6] is a policy-gradient type of method. The algorithm directly optimizes the expected reward by estimating the gradient of the policy from the trajectories taken by the agent. This is usually an on-line method, where only the most recent transitions taken by the agent is used to update the policy. PPO is easy to use during development because no hyperparameter tuning is needed.

Recent advances in the use of DNNs for control management, like in video games, are based on policy gradient methods. But it's hard to get satisfactory results with the methods based on policy gradient because of their sensitivity to the size of the steps. If the steps are too small, progress is too slow to be useful, and if they are too big, the signal gets lost in the noise or there could be huge drops in performance. In addition to this, they usually suffer from the problems of low efficiency, because they may take as many as million, or even billions, of timesteps to learn even very simple tasks, if the step size is too small.

E. *Advantage Actor Critic (A2C)*

The Advantage Actor Critic (A2C) algorithm [13] of RL domain integrates the Policy Based algorithms with the Value Based RL algorithms. Policy-based agents aim for directly learning a policy (i.e. a set of actions that are likely to happen) by mapping the states, they are currently in, to the actions they should do. Value-based algorithms learn to choose actions based on what they think the input state or action will be worth.

It doesn't use a replay buffer because it has more than one worker. The actor critic algorithm consists of two networks—the actor network and the critic network—that collaborate for finding the solution to a given problem. At each time step, the actor network chooses an action, and each input state is judged by its quality, or Q-value, by the critic network.

The Advantage Function figures out, at top level, the agent's Prediction Error or the Temporal Difference (TD) Error. The advantage function is not a value; it is a function. The TD error can help us get close to the advantage function, but it's not the same thing.

As the critic network learns which states are better and which are worse, the actor employs this useful information to teach the agent to look for good states and avoid bad ones. That means that we can't learn the value function for one policy by following another policy. If we're using experience replay, for example, we'll use a different policy because when learning is based on too-old data, the information is used from a policy that's a little bit different from the current state.

IV. Proposed Methods

We are aiming to experiment with several algorithms of reinforcement learning with the goal to have RL models that can play Super Mario Bros. game successfully. For this purpose, first, we need to set up our emulator to communicate with the RL package. Once the environment is set up, the agent learns through a series of trials and errors with the allowed actions. Lastly, after agents of each algorithm finish their learning, they will be tested and compared against each other for performance metrics analysis.

The following sections will explain in detail about every method being employed in this project.

A. *Emulator*

OpenAI Gym [14] is an open-source toolkit for Python language to implement RL algorithms and to compare these algorithms with each other. It gives learning algorithms and environments a standard way to talk to each other, along with multiple sets

of environment that provide standard compatibility with the provided API. These multiple environments can be set up for a specific game. There is an environment developed for Super Mario Bros. game called gym_super_mario. Python programs can simply import the package and use them.

The environments in Gym API are modelled as simple env classes in Python. The environment constitutes of what the agent interacts with in a set of multiple observations, actions, and rewards. The agent, at each step, selects an action from a pre-determined set of actions, in which, the programming set up would be movements. The action is sent to the emulator, which controls the character and then changes the character's game score as well as internal state to reflect what happened. The agent can't see what's going on inside. The agent looks at an array of raw pixels values which shows the current frame on the screen. Another crucial aspect the agent keeps track of is the reward which is affected by the result of each action.

In the emulator, there are a few of available action set to apply to the game play. Those are: 'COMPLEX_MOVEMENT', 'RIGHT_ONLY', and 'SIMPLE_MOVEMENT'. I used Simple Movement for this project. The simple movement consists of: No action, move right, 'move right and press A' (jump), 'move right and press B' (attack when powered up, or move faster), 'move right and press both A and B', press A to jump, and move left. These simple movements are the main actions that the agent will execute during the entire training time.

The game will automatically close if Mario dies or shortly after the flagpole is touched. The game will only accept inputs after the timer has started to decrease. The total reward is determined by the built-in reward function.

B. Reward Function

The reward function in Super Mario Bros. game assumes that the goal of the game is to move the agent, without dying, as far to the right as possible and as quickly as possible. Thus, the reward for this game is made up of three different variables:

1) v: the difference in agent x values between states *[14]*
   - in this case this is instantaneous velocity for the given step
   - v = x1 - x0
     - x0 is the x position before the step
     - x1 is the x position after the step
   - moving right ⇔ v > 0
   - moving left ⇔ v < 0
   - not moving ⇔ v = 0
2) c: the difference in the game clock between frames *[14]*
   - the penalty prevents the agent from standing still
   - c = c0 - c1
     - c0 is the clock reading before the step
     - c1 is the clock reading after the step
   - no clock tick ⇔ c = 0
   - clock tick ⇔ c < 0
3) d: a death penalty that penalizes the agent for dying in a state *[14]*
   - this penalty encourages the agent to avoid death
   - alive ⇔ d = 0
   - dead ⇔ d = -15

❖ r = v + c + *d [14]*

The total reward is clipped so that it comes into the range (-15, 15). By default, the gym_super_mario_bros library determines the reward at each step by calculating Mario's velocity (positive points while moving right, negative points while moving left, zero while standing still), plus a penalty for every frame that passes to encourage movement, and a penalty if Mario dies for any reason. While this is a robust reward system, the levels could be played out more "normally" (i.e., the way most humans would play them) by rewarding Mario for increasing his in-game score by defeating enemies, grabbing coins, and collecting power-ups. This information is saved in an information dictionary, and the previous score can be compared based on the dictionary with the current game score to find a difference in the latest step and add it to the reward.

C. Use a Convolution Neural Networks

The images of Video frames of the game are directly used as input to the proposed model after performing dimensionality reduction to reduce computation costs. Each frame was sampled from the original 256 × 240 pixels to a more pixelated grayscale image of 56x56 pixels. The stable baseline 3 libraries from a common reinforcement learning Python package handle the convolutional neural network (CNN) design. Moreover, CNN is used in this model by applying the wrappers to the environment. Wrapper classes make learning easier and are commonly used for reinforcement in the learning process. OpenAI Gym makes it easy to create custom wrappers around an environment to alter various aspects of the game or virtual world, such as modifying image shapes and sizes, stacking multiple frames together, and changing reset conditions.

After applying the above wrappers to the environment, the final wrapped state consists of 4 grayscaled consecutive frames stacked together, as shown in Fig. 2 on the left. Each time Mario makes an action, the environment responds with a state of this structure. A 3-D array of this size represents the structure [4, 56, 56].

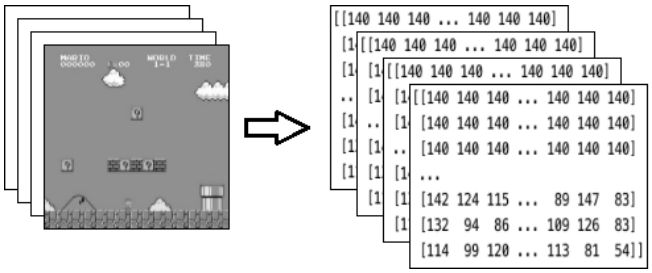

Fig. 2. Apply CNN so Machine can understand Super Mario graphic

D. *Training Models*

The training setup was consistent across all the three algorithms discussed in Section IV.B, enabling a simple comparison between different algorithms for this use case. A gym environment variable was initialized with Super Mario Bros at level 1 to start training reinforcement learning algorithms. The environment variable was then passed through several wrappers to customize (or preprocess) the environment, such as pixelating all of the frames and making them grayscale.

The environment was set up to train over 1,000,000 timesteps and 512 n_step. This means that we have one environment running during the call with about 20,000 policy updates. An episode is an interval between the agent's terminal state and the game's finish point. This assumes a total of 2000 training episodes for each model.

The time, it takes to finish the training, is variable for difference algorithms due to the complexity of each algorithm and the environmental setup, which was more optimized for specific algorithms as compared to others.

Training time comparison:

DQN	PPO	A2C
1 hour 29 minutes *	20 hours 41 minutes	11 hours 50 minutes

Tab. 1. Training time comparison table

DQN has a significantly shorter training time than the others, as shown in Tab. 1. This is not because it is a better-optimized method than the other two, but because the images were scaled very small to match the available memory in this particular training session.

Checking the code and the result, PPO takes the longest time for the environmental set up to train. A2C also takes quite a long time, although significantly less than PPO.

V. RESULTS

Although all three development environments for DQN, PPO, and A2C have the same configuration, however, the results varied among them. The following section entails each reinforcement learning technique's results. It explains some of the metrics and loss function graphs obtained through a standard tool for computing metrics in machine learning called TensorBoard.

A. *DQN Results*

After training over 1000000 timesteps and 512 n_step, the model we obtained is shown in Fig. 3. Using simple scpiy and matplotlib, this metric was obtained to describe the DQN result model.

Below is a plot of DQN's Loss function.

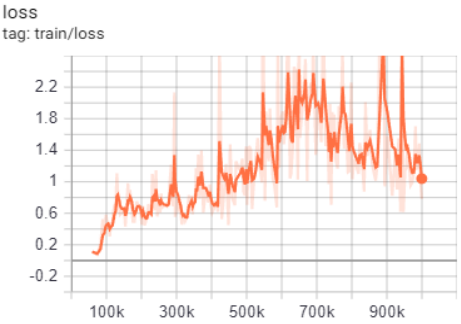

Fig. 3. Loss Function for Deep Q-learning

This can be observed that during training in Fig 3, starting at 650,000, the model stops getting improvement. The results suggest that as we keep training past the 650,000 timesteps mark, the model will not improve anymore unless we change something to optimize the training process further.

A test was conducted to verify whether this DQN model will be overfitted after training 600,000 timesteps. At three different training points, when timestep is at 450,000, 600,000, and 1,000,000, the DQN model is tested by playing the game for 400 episodes is shown in Table 2.

RESULT OF DQN AFTER TESTING WITH 400 EPISODE

Result of DQN			
Total Time-step	***At 450000***	***At 600000***	***At 1000000***
Average distance travel by Mario	582 steps	641 steps	627 steps

Tab. 2. Comparison of DQN models at the different training stages.

After the DQN model is running for 400 episodes three times, each time at a different stage of training. On average, over 400 episodes, Mario can only travel 582 steps when the model is trained at 450,000 timesteps or approximately 900 episodes. At 600,000 timesteps (approximately 1200 episodes), DQN performance was improved, but not much more improved beyond that. The model stops improving at the 1,000,000 timestep mark but performs worse on average. The results matched the loss function graph.

It is concluded that there are no more benefit of keep training the agent beyond 600,000 timestep, unless we modify the setup.

B. PPO Results

After training, the model was captured at every 100000 timestep mark. Using TensorBoard, the following figures were obtained to represent the loss functions of the model:

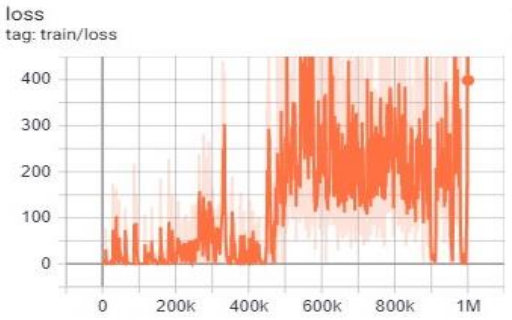

Fig. 4. Loss graph of PPO model

The loss function graph tells us that at around 300,000 timesteps, PPO model seems to peak its performance. As we get past 450,00 timestep, the loss function goes up so high that it suggests that the model would perform worse at the later stage of training.

The policy gradient loss graph also shows a similar graph. These metrics show that more training does not always mean a better model, and it was proven when we evaluated the PPO model at different training stages several times. Each time has a testing duration of 400 episodes.

RESULT OF PPO AFTER TESTING WITH 400 EPISODE

Result of PPO				
Total Time-step	***At 300000***	***At 450000***	***At 600000***	***At 1000000***
Average distance travel by Mario	864 steps	633 steps	679 steps	892 steps

Tab. 3. Compare PPO models at the different training stages.

We can see that at 300,000 timesteps, we have an outstanding performance from PPO. The Super Mario, on average, has traveled farther. At 450,000 and 600,000 timesteps, we see that the model performance is even worse than the 300,000 timesteps, even though it has more training experience. We only see the improvement at the 1M timestep mark, as shown in Tab 3. Based on our observation, this technique is very optimized at 300,000 timestep and the performance has decreased as the loss function increases beyond the 450,000 marks. It is not until towards the end that we see the performance gets good results; the performance at 1 millison timesteps is reflected at the drop in loss function in the graph.

C. A2C Results

After training the model 1000000 timestep mark, the loss graph is shown in Fig. 5. Using TensorBoard, the following figures were obtained to represent the entropy loss functions of the model:

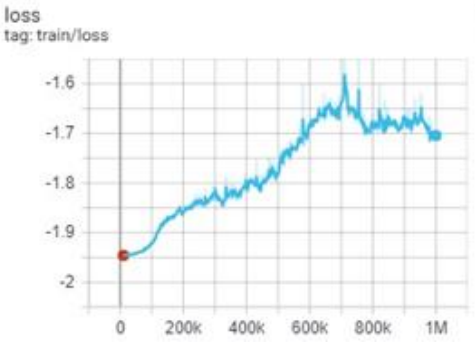

Fig. 5. Loss graph of A2C model.

A test was conducted to verify whether this A2C model will be overfitted beyond the 1,200 episodes. At three different training points, when timestep is at 450,000, 600,000, and 1,000,000, the A2C model is tested by playing the game for 400 episodes.

RESULT OF A2C AFTER TESTING WITH 400 EPISODE

Result of A2C			
Total Time-step	***At 450000***	***At 600000***	***At 1000000***
Average distance travel by Mario	504 steps	505 steps	502 steps

Tab. 4. Compare A2C models at the different training stages.

The result in Tab 4 shows that the A2C performed severely compared to the other models. Mario, on average, does not travel very far from the starting point. Even at a different stage of training, the model still performs poorly and shows no improvement. We concluded that in this technique, as we get pass 400,000 timesteps the agent stops improving. The overall results are not good. Unless we modified the code to set up our environment and parameters differently, it is not worth it to keep training with A2C technique.

D. Conclusion

In this work, 3 reinforcement learning models are trained that can play Super Mario Bros. game automatically. Each model was trained with a different reinforcement learning algorithm. Because the training was done only with level 1 of the game, the resulting models can only be played best with level 1.

In all three models, Mario neither reaches the end of the level. Based on the observations, Mario has

always died early in the game or got stuck somewhere, not knowing how to move forward. I tried to keep the observation similar among the models. Even though each model works differently, it was concluded that for the task of developing a policy based on many rules and a complex environment, PPO is the best option.

Of course, this conclusion is drawn without any regard for the potential fine-tuning works specifically for each algorithm. However, with more time and resources invested in fine-tuning and optimizing the data before training and developing a custom reward function, the models that we would obtain are expected to be more successful than our current implementation.

VI. Discussion

A. *Challenges*

The preparation of the simulation environment, which is highly dependent on the job to be completed, is the main problem in reinforcement learning. It is also not a simple task to transfer the model from the training environment to the real world.

Scaling and modifying the neural network that controls the bot is another issue. The only way to communicate with the network is to use the reward and punishment mechanism and no correct outputs are available as in supervised learning. This constraint may lead to catastrophic forgetting, in which new information causes some old information to be lost from the network.

Yet another challenge during this research work was the lack of hardware resources. Training using a Graphical Processing Unit (GPU) will decrease the waiting time for the algorithm to finish training. Moreover, another hardware limitation was small amount of RAM. The images were downsampled to 56x56 pixels because the stable_baselines3 library required more than 50 GB in memory to store many tensors during the training process. The available RAM was only limited to 32GB, so the downsized images of size 56x56 pixels were used for different experiments in this research work.

B. *Plans for Future Work*

As mentioned in the previous section, more time should be spent setting up a development environment, in which graphical hardware can be utilized to enhance the training speed in future work. Another aspect that needs to be improved is to fine-tune the model by cleaning and processing the data before training, in this case, the environment variable. The improved training data will give a much-improved model as a result.

Another aspect of this project that was not completed was the ability to play the different levels of the game. The model can only play level one of the game, which makes this model very weak against real-world use cases. Future efforts need to explore options of expanding the model to be more robust and handle multiple different levels of this game, perhaps using a different level to train.

Feature detection will be another aspect of this research work that was outside the scope of learning and experimenting with reinforcement learning algorithms used in this work. Finally, the future work should consider implementing object detection techniques to identify the enemy and hazardous objects before making Mario move and avoid those hazards.

VII. References

[1] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv Prepr. arXiv1811.12560*, 2018.

[2] N. Inc., "The Super Mario Bros. Game." https://supermariobros.io/ (accessed May 14, 2022).

[3] M. E. Taylor, "Teaching reinforcement learning with mario: An argument and case study," 2011.

[4] D. G. LeBlanc and G. Lee, "General Deep Reinforcement Learning in NES Games".

[5] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep Q-learning," in *Learning for Dynamics and Control*, 2020, pp. 486–489.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms." arXiv, 2017. doi: 10.48550/ARXIV.1707.06347.

[7] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, 2016, vol. 30, no. 1.

[8] Robert Clark, "Mario Gym Routine - A practical guide to writing a RL application which trains Mario to beat levels," *Towards Data Science*, 2020. https://towardsdatascience.com/marios-gym-routine-6f095889b207

[9] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*, 2017, pp. 1–6.

[10] Y. Liao, K. Yi, and Z. Yang, "CS229 Final Report Reinforcement Learning to Play Mario," 2012. [Online]. Available: http://www.stanford.edu/~yzliao/pub/CS229_report.pdf

[11] S. D. Holcomb, W. K. Porter, S. V Ault, G. Mao, and J. Wang, "Overview on deepmind and its alphago zero ai," in *Proceedings of the 2018 international conference on big data and education*, 2018, pp. 67–71.

[12] W. Fedus *et al.*, "Revisiting Fundamentals of Experience Replay," *CoRR*, vol. abs/2007.06700, 2020, [Online]. Available: https://arxiv.org/abs/2007.06700

[13] F. AlMahamid and K. Grolinger, "Reinforcement Learning Algorithms: An Overview and Classification," in *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2021, pp. 1–7.

[14] C. Kauten, "Super Mario Bros for OpenAI Gym," *Python Package Index*, 2020. https://pypi.org/project/gym-super-mario-bros/