Influence of Hyperparameters on Deep Reinforcement Learning Training Speed

Karena Qian

College of Natural Sciences

University of Texas at Austin

December 2, 2024

Abstract

Deep reinforcement learning (DRL) is one of the newest and most invested subdomains in the artificial intelligence (AI) circle. Despite their many strengths, DRL models has one weakness that has yet to be fully addressed: slow training speeds. We demonstrate four different common hyperparameters in deep reinforcement learning and their unique influences on the training speed of agents. It turned out that larger learning rates, smaller discount factors, and larger exploration decay factors may all help agents learn faster.

Contents

1. Introduction

Deep reinforcement learning (DRL) is one of the newest and most invested subdomains in the artificial intelligence (AI) circle. It powers many of today's ground-breaking innovations, including self-driving cars, humanoid robotics, emotional machines, music generation, and more (Li, 2018)! Despite their many strengths, DRL models has one weakness that has yet to be fully addressed: slow training speeds.

Many practical applications requires AI models that can learn their task quickly and correctly (Walther, 2021). Unfortunately, DRL models, more often then not, need a lot of training, which makes them really expensive investments (Scholten and Kuijt, 2023). In fact, a DRL model may require several weeks of uninterrupted game-play to learn Atari while regular humans only need 15 minutes to reach its level of expertise (Barreto et al., 2020). As of now, there are several solutions proposed that mitigate, if not solve, this problem, including parallelization (Ganeshkumar, n.d.), sample efficiency (Scholten and Kuijt, 2023), transfer and curriculum learning (Maluso, 2021), model-based reinforcement learning (Maluso, 2021), and more. However, there is one approach that might seem insignificant, but may play a huge role in improving training speed, namely hyperparameter optimization.

Unfortunately, optimizing hyperparameters is often a difficult task to accomplish, especially if the datasets and the DRL model complexity are large, which often leads to increased training time and resource consumption (Jomaa et al., 2019). It is also a very important task since, according to multiple sources, hyperparameters directly impact the training duration of DRL models (Ashraf et al., 2021; Eimer et al., 2023; Jomaa et al., 2019; Yang and Shami, 2020). Therefore, the goal of our experiment is to figure out how and how much hyperparameters influence DRL training speed. Then, hopefully, we can use the results to gather insight as to how hyperparameters can be adjusted to increase training speed.

2. Research Background

2.1. Deep Reinforcement Learning (DRL)

2.1.1 Brief History

Deep reinforcement learning (DRL)'s origins can be traced all the way back to the advent of reinforcement learning (RL), which itself originated from studies about reinforced behavior in animals and optimal reward problems during the 1900s (Agostinelli et al., 2018). Since then, RL experienced a rapid succession of advancements, including the introduction of genetic algorithms, TD Learning, and Q-Learning, all the way to the early 2010s (Agostinelli et al., 2018). Then, in 2013, DRL was introduced through Google DeepMind's Deep Q-Learning (DQL) algorithm that learned how to play Atari 2600 video games at a superhuman level (Alom et al., 2018; Arulkumaran et al., 2017; Shehu, n.d.). Just two years later, a super-program called AlphaGo defeated its first Go human opponent, and another two year later, it itself was defeated by its descendant called AlphaGo Zero (Shehu, n.d.). Since then, DRL has become a prominent area of AI and machine learning (ML), and has been applied in many areas including games, robotics, healthcare, science, arts, natural language processing, finance, motion analysis, and more (Li, 2018).

2.1.2 Overview

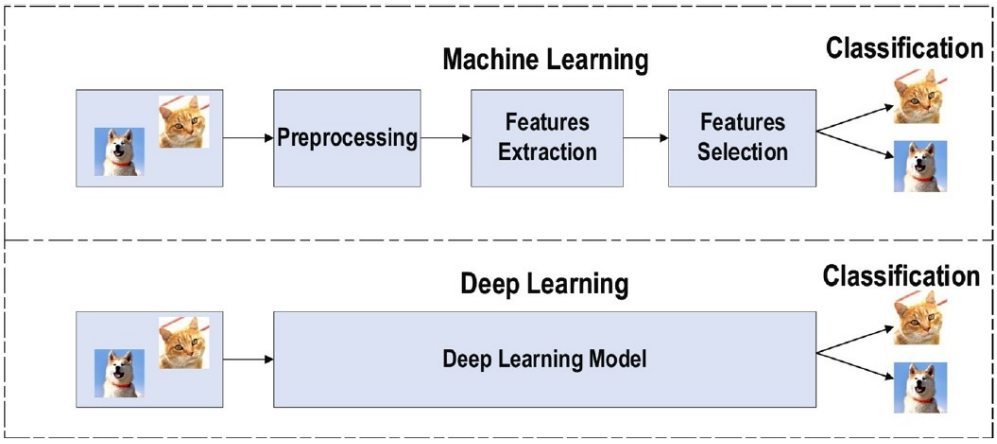


Figure 2.1: ML vs DL (image from Alzubaidi et al., 2021)

To understand why DRL is such a powerful invention, we first need to understand its two main components: deep learning and reinforcement learning. **Deep learning (DL)** is a subdomain of AI and ML that is centered around the development of an innovative ML scheme for state representation or function approximation: **deep neural networks (DNN)** (figure 2.2) (Li, 2018). Unlike most ML models, DNNs have one or more hidden layers (or mappings) between the input and output (figure 2.1) (Li, 2018). This allows them to detect "compositional hierarchies" in natural signals (Li, 2018), or in other words, break down high dimensional data (image, text, and audio) into low dimensional features (Arulkumaran et al., 2017). Common examples of DNN used in research and applications are the Convolutional Neural Network (CNN), the Recurrent Neural Network (RNN) and the Long Short Term Memory Network (LSTM) (Li, 2018).

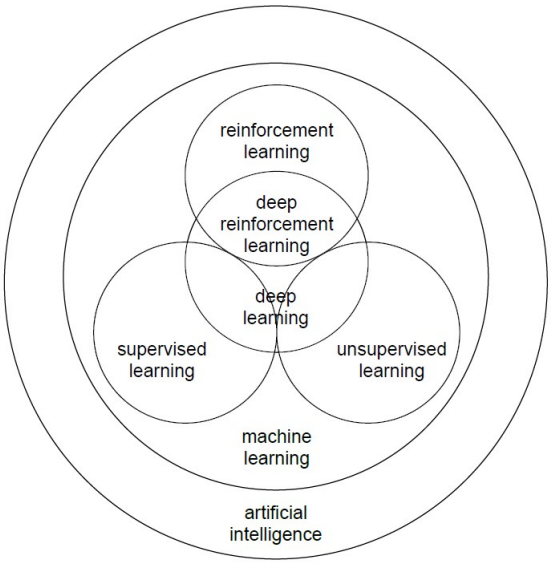


Figure 2.2: Relationship Between Deep Learning, Reinforcement Learning, and Deep Reinforcement Learning (and others) (image from (Li, 2018))

Just like DL, **reinforcement learning (RL)** is a very prominent and unique subdomain of both AI and ML (figure 2.2). It encompasses the development of learning algorithms that specialize in building and training ML models that make sequential decisions or actions (Zhang et al., 2021). Unlike other types of learning algorithms, including supervised and unsupervised learning, future and past predictions (called **actions** in RL) are closely correlated, the past ones affecting future ones (Zhang et al., 2021).

This correlation is the result of how ML models are trained in RL algorithms. During an iteration of a typical RL training loop (also called a **step**), an **agent** (a machine learning model (Aljadery and Sharma, n.d.)) interacts with its **environment** (the world its in (Achiam, 2018)), then adjusts its behavior based

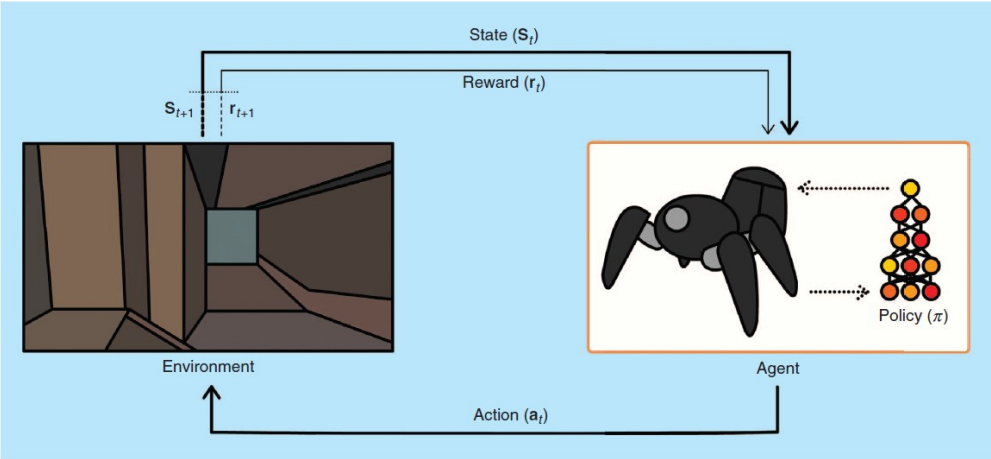


Figure 2.3: Training Loop in a Typical Reinforcement Learning Algorithm (image from Arulkumaran et al., 2017)

on gained **rewards** (signals from environment telling if the environment state is good or bad (Achiam, 2018)) given in consequence of its action (Majid et al., 2021). Training in RL can be broken down as follows:

1. The agent observes environment state s_t at step t.

2. The agent, using its policy π (a rule used to decide the action (Achiam, 2018)), selects an action a_t from its action space (set of all valid actions (Achiam, 2018)) and executes it in s_t.

3. The agent receives a reward r_{t+1} from environment and observes the next state s_{t+1} of the environment.

4. The agent takes this **transition** (information about s_t, a_t, s_{t+1}, and r_{t+1}) and uses it to improve its policy π.

5. Then steps 1-4 repeat until the environment gets into a terminal (ending) state. Afterwards, if the whole training session is episodic, the training session restarts (Li, 2018)

(Figure 2.3 gives a beautiful visual representation of this training loop.) The goal of this training is to teach the agent the optimal policy π that maximizes the total reward received from interactions with environment (Arulkumaran et al., 2017; Majid et al., 2021).

Now, there was a really significant problem normal RL algorithms faced: lack of scalability caused by dimensional limitations (Arulkumaran et al., 2017). Due to their inability to process high dimensional inputs like vision and speech, normal RL could only solve low dimensional problems (Mnih et al., 2013).

DRL algorithms are able to surpass this weakness by integrating the technologies of DL into the workings of RL to create one technological powerhouse (Li, 2022; Majid et al., 2021). These learning algorithms behave the same as regular RL algorithms, but with DNNs as the policies of the agents (Aljadery and Sharma, n.d.). This special combination is what makes DRL-trained models such powerful and widely applicable pieces of technology in today's world.

2.2. Hyperparameters

Even though DRL algorithms are very powerful and effective, they are not always easy to work with. Like all other ML algorithms and models, DRL algorithms and models consists of two types of parameters: the trainable model parameters, which are initialized then updated through training, and the preset **hyperparameters**, which are set before training begins (Ashraf et al., 2021; Yang and Shami, 2020). Even though they are often not the main focus of ML, hyperparameters still play an important role since they help either with the configuration of ML models or the specification of algorithms used to minimize the model loss (Yang and Shami, 2020). Because they adapted concepts from two subdomains of ML and AI, DRL algorithms and models have a wide portfolio of adjustable hyperparameters, which often include the following:

- **Learning Rate**, which determines the step size of the gradient descent algorithm, which then determines how fast the DRL algorithm converges to the optimal policy (Yang and Shami, 2020).

- **Discount Factor** (often denoted as γ), which represents the importance of future rewards in proportion to immediate rewards (Chadi and Mousannif, 2023; van Hasselt et al., 2015). Smaller values prompts the DRL agent to disregard future rewards more, while larger values encourages the agent to value future and immediate rewards more equally (FutureMachineLearning.org, n.d.).

- **Exploration Rate**, which is a probability that controls how often the agent "explore", or choose a random action (Sheryl, 2024).

- **Batch Size**, which determines how much the DRL algorithm samples from a special memory bank called a **replay buffer/memory** (Paszke and Towers, n.d.).

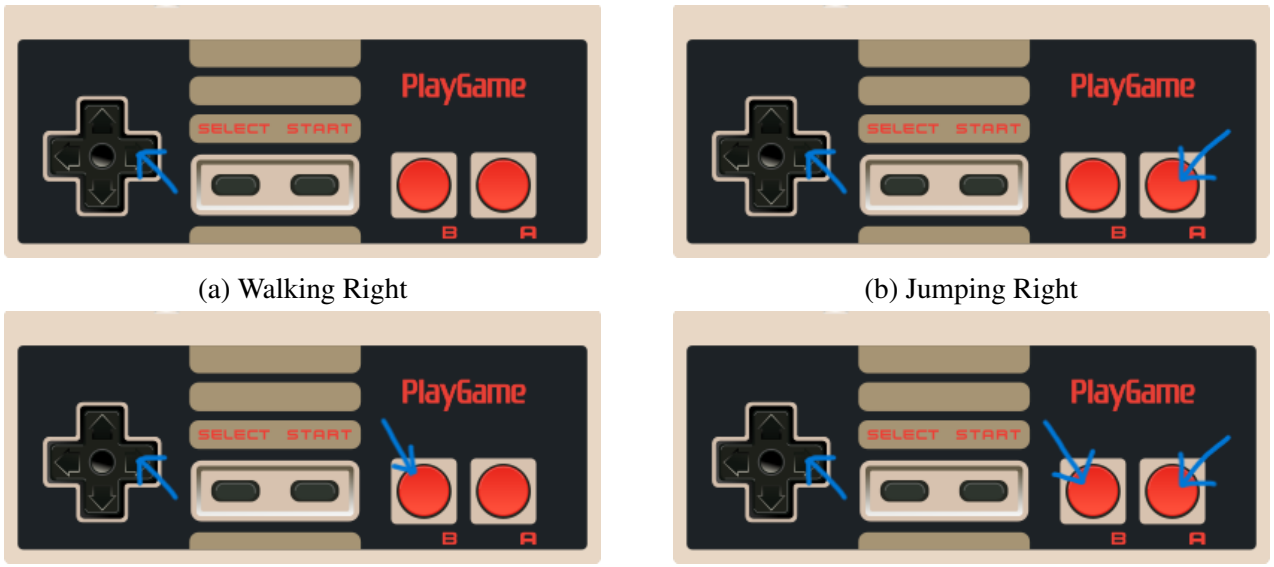
- **Replay Buffer Size**, which is the size of the memory bank that tracks the DRL agent's interactions with its environment (Nair, n.d.).

- **Target Network Update Rate** (often denoted as τ), which controls how frequently the parameters of a special **target network** are updated (Nair, n.d.).

With so many unique yet important hyperparameters to tune, it is no wonder that finding the optimal hyperparameters for DRL algorithms and models is a very daunting task.

3. Materials and Data Sources

Unlike many ML algorithms, in most deep reinforcement learning (DRL) algorithms, RL agents train using data they gather from the **environment** that they interact with. For our experiment, we decided to let our agents train on a customized Nintendo Entertainment System (NES) Super Mario Bros game environment provided by the "gym-super-mario-bros" OpenAI Gym environment (Kauten, 2018). This special environment allows our agents to control the main character, Mario, through a virtual NES game system, and it gives them all the data they need to learn about their surroundings, including observations of the surroundings, points awarded to the agents, whether the game terminated, and more.



(a) Walking Right

(b) Jumping Right

(c) Running Right

(d) Running Jumping Right

Figure 3.1: The Action Space's NES Controls (NES image from GDJ, 2023)

For simplicity, our agents are given a small action space with only five moves, of the 256 possible NES move combinations, to choose from: walk right (figure 3.1a), jump right (figure 3.1b), run right (figure 3.1c), and run jump right (figure 3.1d) (Nintendo, 2016). Our RL agents are also allowed only 1 life, or attempt, to pass through a single stage. This may result in our agents merely memorizing the layout of a certain stage, but at the same time it helps simplify and quicken the training process.

During the training process, each agent is given all the information they need to learn how to traverse their environment. First, they are given player's-point-of-view **observations** of the stage, which they use to decide and evaluate the actions they take. Originally, each observation consisted of an image showing

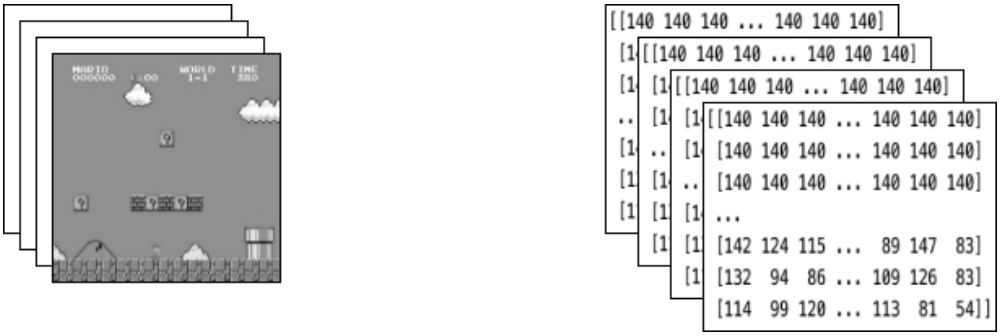


Figure 3.2: Image Preprocessing (image from Feng et al., 2023)

the state of the game environment; however, each image is quite large in size, and often contained extra unnecessary information. To address this, we convert each image into a gray-scale representation and resize that down to a perfect square. We also squash 4 consecutive gray-scale squared images together into one image, which is then sent to the agent.

After each taken action, our agents are given a **reward**, which is a signal from the environment, often awarded points, that informs them if the action they took yielded the desired outcome (Achiam, 2018). The reward that the Super Mario Bros environment gives is calculated using the following **reward function** (clipped into a range for uniformity):

$$reward = v + c + d \tag{3.1}$$

$$-15 < reward < 15$$

Here, v represents the instantaneous horizontal velocity of the agent-controlled Mario, which is calculated as $x_1 - x_0$, where x_1 is Mario's current horizontal position and x_0 is Mario's previous horizontal position. This variable encourages our agents to move right as quickly as possible, since they are rewarded if Mario is moving right, but penalized if Mario is moving left. The next term, c, represents a deduction that penalizes the agent if the game clock is still ticking. It is calculated by $c_0 - c_1$, where c_0 is the game clock reading before the agent's action, and c_1 is the game clock reading after the agent's action. This deduction prevents our agents from taking no course of action with Mario, or in other words, making him stand still at any given time. Finally, the last term, d, represents a heavy -15 penalty triggered only if Mario dies. This encourages our agents to avoid killing Mario as much as possible (Kauten, 2018).

Another important piece of data give to our agents is a special variable called **done**. It simply informs

our agents whether the game ended, which occurs if Mario dies or the agent runs out of time. Of course, this outcome is not desirable, and as described above, the reward function encourages our agents to avoid such situations as much as possible.

Other information provided by the environment includes the number of coins, how my lives Mario has, the game score, and more. However, a piece of data that is most useful to us is the "flag-get" status, which tells us whether the agent has successfully completed the stage or not. This data is not used by our agents directly, but is an important metric that we used in our analysis (Kauten, 2018).

4. Research and Methods

To implement and train our agents, we decided to use a special deep reinforcement learning algorithm called Double Deep Q-Networks (DDQN). Due to our lack of experience implementing reinforcement learning algorithms, this decision was largely made by a tutorial, written by Yuansong Feng, Suraj Subramanian, Howard Wang, and Steven Guo, that we decided to follow (Feng et al., 2023). The tutorial's particular implementation of the DDQN algorithm employs Convolutional Neural Networks (CNNs), which is perfect for our agents since they have to learn from image data. Thus, we used the tutorial to implement the DDQN algorithm, then adjusted and tweaked it so we can experiment on four of the various hyperparameters the algorithm utilizes.

4.1. Double Deep Q-Networks (DDQN) Algorithm

4.1.1 Q-Learning

To understand the DDQN algorithm we need to start from the widely used and simplistic **Q-Learning** algorithm. Q-Learning is a tabular TD-Learning algorithm founded by Christopher Watkins in 1989 (Chadi and Mousannif, 2023). It has the following properties:

- **Off-Policy**, which means the algorithm's agent uses one model for exploration and another for exploitation (Chadi and Mousannif, 2023), separating the acting from the learning (Jang et al., 2019).
- **Value-Based**, which just indicates that the model's policy depends on the action-value function Q function (Achiam, 2018; Chadi and Mousannif, 2023)
- **Model-Free**, which indicates that the algorithm cannot know the rules of the environment beforehand (e.g., the rules of chess) (Chadi and Mousannif, 2023).
- **TD (Temporal Difference)**, which means the model updates after each step (Chadi and Mousannif, 2023).
- **Tabular**, which indicates that the model of the agent is a table that stores some computed values (Chadi and Mousannif, 2023).

Each Q-Learning agent utilizes a special table called a **Q-table**, in which each entry representing the correctness of executing certain actions in certain environment states (Chadi and Mousannif, 2023).

To obtain such correctness values, Q-Learning uses a special **action-value function** called a **Q function** to evaluate the correctness of the actions the agents take, which is as follows:

$$Q_t(s,a) = E[Q_t|s_t = s, a_t = a] = E[\sum_{k=0}^{T} \gamma^k r_{t+k+1}|s_t = s, a_t = a] \tag{4.1}$$

Unlike some other action-value functions, function 4.1 relies on both the states s of the environment and the actions a of the agent (Chadi and Mousannif, 2023). It then returns a special value called a **Q-value** that represents the quality of the agent's actions with respect to both the cumulative reward $\sum_{k=0}^{T} \gamma^k r_{t+k+1}$ and the environment state.

With these tools, for each step, the agent uses the Q-table and the following policy to select the action that has the maximum Q-value (Achiam, 2018):

$$a_t(s) = \underset{a}{argmax}(Q(s_t,a))$$

Then, it updates Q-values in its Q-table based on the observed reward and the estimated future Q-value (Chadi and Mousannif, 2023; Jang et al., 2019)

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \cdot [r_{t+1} + \gamma \cdot \max_a(Q(s_{t+1},a)) - Q(s_t,a_t)] \tag{4.2}$$

And so, the goal of Q-Learning is to teach the agent the optimal Q-table that assigns the highest values to the most optimal state-action entries (Chadi and Mousannif, 2023).

Even though the Q-Learning is a very simple and useful algorithm, it suffers from some crippling limitations. First, its applications are very limited since it cannot train in environments with continuous high-dimensional states and actions (Chadi and Mousannif, 2023). This is because of its usage of finite-size tables as its agents' "brains," (Chadi and Mousannif, 2023) which can become costly both in usefulness and memory (Jang et al., 2019). Also, Q-Learning can overestimate significantly, especially in some stochastic environments. This is due to positive bias stemming from $\max_a(Q(s_{t+1},a))$ in equation 4.2.

4.1.2 Deep Q-Learning

In an attempt to address these issues, researchers from Google DeepMind introduced the **Deep Q-Learning (or Deep Q-Networks or DQN)** algorithm in 2013 (Chadi and Mousannif, 2023). This algorithm works the same as the Q-Learning algorithm, but with 3 major difference:

1. the integration of Neural Networks
2. the introduction of a replay buffer
3. the addition of a target network

Instead of using a Q-table to store Q-values, DQN makes use of a neural network (NN) (oftentimes a Convolutional Neural Network) to perform that same function. And so, instead of updating Q-values in a table, the algorithm updates the parameters θ of the NN instead (van Hasselt et al., 2015)

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta [r_{t+1} + \gamma \max_a (Q(s_{t+1}, a; \theta_t)) - Q(s_t, a_t; \theta_t)] \tag{4.3}$$

Because NNs can process continuous high-dimensional data, this makes DQN much more applicable than regular Q-Learning.

However, just integrating a NN won't suffice, since NNs often highly unstable. To address this, a **replay buffer** was added to the DQN algorithm. It's main function is to simply store transitions. Then from it, the agent can randomly sample transitions to learn from (Chadi and Mousannif, 2023).

Another addition that helped stabalize the NN model and learning process is the **target network** (Chadi and Mousannif, 2023). This network is a NN with weights θ^- whose sole purpose is to calculate "real" target Q-values that the predicting network, called the **Q-network**, learns from using this equation (Chadi and Mousannif, 2023; Jang et al., 2019):

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta [r_{t+1} + \gamma \max_a (Q(s_{t+1}, a; \theta_t^-)) - Q(s_t, a_t; \theta_t)] \tag{4.4}$$

Unlike the Q-network, this NN does not get updated after every step, but rather, it updates by copying the parameters of the Q-network after a number of steps or episodes (often referred to as τ) (Amber, 2019; Chadi and Mousannif, 2023; van Hasselt et al., 2015).

Even though the DQN algorithm greatly expanded Q-Learning's capabilities, it can still suffer from significant overestimation of its Q-values (van Hasselt et al., 2015). This is largely due to great training

instability, especially in large high-dimensional environments (Chadi and Mousannif, 2023). DQN added great improvements to the Q-Learning algorithm, but it is unable to solve the overestimation problem.

4.1.3 Double Q-Learning

To address this issue, Hado van Hasselt took Q-Learning and developed a new algorithm called **Double Q-Learning (DQL)** in 2010 (Jang et al., 2019). This algorithm works exactly the same as Q-Learning, but with one major difference. It divides the update equation into two separate ones, as shown below (van Hasselt, 2010):

$$\begin{aligned} Q^A(s_t,a_t) &\leftarrow Q^A(s_t,a_t)+\alpha\cdot[r_{t+1}+\gamma\max_a(Q^B(s_{t+1},a^*))-Q^A(s_t,a_t)] \\ a^* &= \max_a(Q^A(s_{t+1},a)) \\ Q^B(s_t,a_t) &\leftarrow Q^B(s_t,a_t)+\alpha\cdot[r_{t+1}+\gamma\max_a(Q^A(s_{t+1},b^*))-Q^B(s_t,a_t)] \\ b^* &= \max_a(Q^B(s_{t+1},a)) \end{aligned} \tag{4.5}$$

Each of these two equations learn from a random separate set of experiences or transitions, but both work together to help the agent choose the actions (van Hasselt, 2010). Later in another paper, van Hasslet et al. generalized the decoupling of the update to the following:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha\cdot[r_t+\gamma\cdot Q'(s_{t+1},\underset{a}{argmax}(Q(s_{t+1},a))-Q(s_t,a_t)] \tag{4.6}$$

It turns out that doing this splitting will prevent DQL from grossly overestimating the Q-values (van Hasselt, 2010)! However, there was one major problem with this algorithm, namely that it was introduced in a tabular setting (van Hasselt et al., 2015). Thus, it shares the same applicable limitations as regular Q-Learning!

4.1.4 Double Deep Q-Networks

Eventually, in 2015, Hado van Hasselt, Arthur Guez, and David Silver took both DQN and DQL and combined them into one algorithm that solved both of Q-Learning's major obstacles called **Double Deep Q-Networks (DDQN)**! This algorithm works exactly the same as DQN, but with a slightly different

update equation (van Hasselt et al., 2015):

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta [r_{t+1} + \gamma Q(s_{t+1}, \underset{a}{argmax}(Q(s_{t+1}, a; \theta_t)); \theta_t^-) - Q(s_t, a_t; \theta_t)] \tag{4.7}$$

Unlike the DQN update equation 4.4, this equation 4.7 adopts the decoupling concept shown in the generalized DQL update equation 4.6. It takes the original $\max_a(Q(s_{t+1}, a; \theta_t^-))$ operation, then transformed it into $Q(s_{t+1}, \underset{a}{argmax}(Q(s_{t+1}, a; \theta_t)); \theta_t^-)$ (van Hasselt et al., 2015). This not only addresses the overestimation issue in DQN, but also minimizes computational overhead (Jayakody, 2022). An overview of the DDQN algorithm can be found in algorithm 1.

Algorithm 1 Double Deep Q-Networks (Chadi and Mousannif, 2023; van Hasselt et al., 2015)

Initialize replay memory R to capacity N
Initialize predictor CNN Q with random weights θ
Initialize target CNN $\hat{Q}$ with same weights $\theta^- = \theta$
for each episode in $[1, M]$ **do**
  Get initial state s_0
  for each step t **do**
    With probability ε, choose a random action a_t ▷ exploration
    Otherwise $a_t = argmax[Q(s_t, a; \theta)]$ ▷ exploitation
    Execute a_t and get reward r_{t+1} and next state s_{t+1}
    Save transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in R
    If R has enough transitions, sample a small batch of random transitions from R
    if episode terminates at step $t+1$ **then**

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta [r_{t+1} - Q(s_t, a_t; \theta_t)]$$

    else

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta [r_{t+1} + \gamma Q(s_{t+1}, \underset{a}{argmax}(Q(s_{t+1}, a; \theta_t)); \theta_t^-) - Q(s_t, a_t; \theta_t)]$$

    end if
    For every Cth step, update target $\hat{Q} = Q$
  end for
end for

Note about algorithm 1: the 1st two steps in the innermost for loop is the ε-greedy method that encourages the RL agents to balance exploring their actions (select random actions) and exploiting the knowledge they gained (choose actions with the highest Q-values) (Chadi and Mousannif, 2023), (Sheryl, 2024). "This approach ensures that the agent continues to explore the environment, while also taking advantage of its current knowledge to make informed decisions. (Sheryl, 2024)"

4.2. Convolutional Neural Network

The basic structure of the Convolutional Neural Network (CNN) that the agents used is as follows (Feng et al., 2023):

```
Convolutional -> ReLU -> Convolutional -> ReLU -> Convolutional -> ReLU
-> Flatten -> Dense -> ReLU -> Output
```

The first part of this CNN is the **convolutional layers**. It uses 3 convolutional layers to extract relevant, higher-level, and more complex features from input images using a set of kernels (or filters) (O'Shea and Nash, 2015; Zhao et al., 2024). It then outputs a set of feature maps representing different features detected in the input images (Krichen, 2023).

The next part of this CNN are the ReLU (Rectified Linear Unit activation layers. These layers follow both the convolutional layers and the dense layer, basically apply this simple equation to each element of the output (Krichen, 2023):

$$f(x) = max(0,x) \tag{4.8}$$

Their purpose is to introduce non-linearity into the CNN so it can approximate nonlinear functions (Krichen, 2023; Zhao et al., 2024). They also prevent the output from getting too high or low, and assist in mitigating the gradient vanishing problem (Krichen, 2023; Zhao et al., 2024).

The rest of the CNN consists of the flatten layer, the dense layer, and the output layer. The flatten layer converts images into column vectors (Saha, 2018), the dense layer simply connects every neuron in the flatten layer to its own neurons, and the output layer produces a probability distribution over the action space of each agent (Krichen, 2023).

As in any typical deep neural network, our CNN updates through back-propagation. This process first propagates the error from the outer layer backwards. Then it computes the derivative of the loss function with respect to the weights and biases of each layer. Finally, it updates the weights and biases with the help of an optimization algorithm. This entire process ensures that the weights are adjusted to minimize the error between the predicted output and the actual values (Krichen, 2023).

4.3. Tested Hyperparameters

For this experiment, we decided to adopt an approach inspired by David Mack's experiment published on FreeCodeCamp.com (Mack, 2018). Basically, we used the DDQN algorithm to train the same CNN

on the first stage of the Super Mario Bros game, but every time, we varied only one hyperparameter. We decided to vary the following four hyperparameters with the following values:

- **Batch Size**: 16, 32, 48, and 64
- **Discount Factor**: 0, 0.1, 0.3, 0.5, 0.7, 0.9, and 1
- **Learning Rate**: 5 values ranging from 0.000001 to 1 with equal intervals
- **Exploration Rate Decay Factor**: 4 values ranging from 0.99995 to 0.9999874999999999 with equal intervals

The reason we are not varying the exploration rate is because our algorithm utilizes the exponential decay schedule, which decreases the exploration rate exponentially by a decay factor at each step of the training loop (Sheryl, 2024).

While one hyperparameter is experimented on, we held the other three hyperparameters constant. We decided to adopt most of the constant values shown in the tutorial, with only a slight change added to the discount factor constant. These are the constant values for each of the four tested hyperparameters (Feng et al., 2023):

- **Batch Size**: 0.99999975
- **Discount Factor**: 32
- **Learning Rate**: 0.99
- **Exploration Rate Decay Factor**: 0.00025

For each hyperparameter, we ran the training procedure for each value until the agent is able to successfully complete the stage it was placed in. We tracked the time it took each agent to learn to successfully navigate its environment, the length of each training episode, the time it took to complete the stage, and other metrics. In the end, we trained 20 agents separate agents, each with varying results.

5. Results

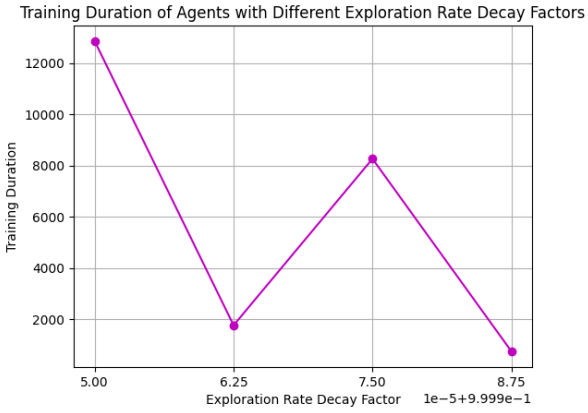
5.1. Hyperparameters vs Training Time

First, we will examine the training times of each of our 20 DRL agents using images 5.1 (which shows the training duration of different agents) and 5.2 (which shows the average mean episode length of different agents).

(a) Different Learning Rates

(b) Different Discount Factors

(c) Different Batch Sizes



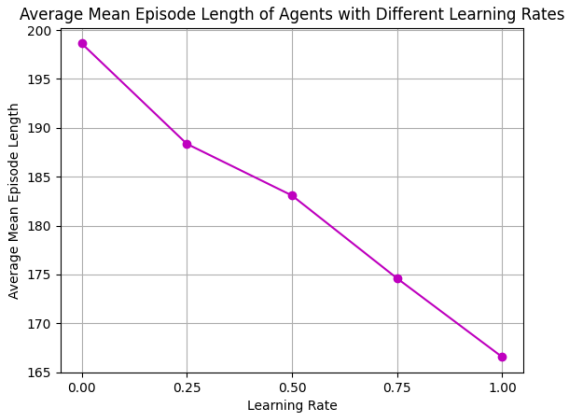
(d) Different Exploration Rate Decay Factors

Figure 5.1: Training Duration of RL Agents

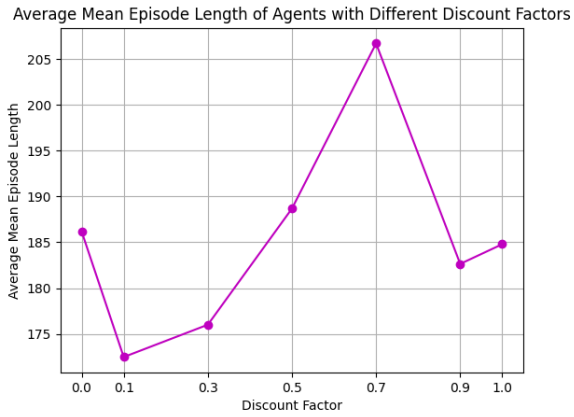
According to figures 5.1a and 5.1d, the learning rate and the exploration rate decay factor (ERDF) of the DDQN algorithm seem to have an overall negative linear relationship with training duration. Smaller values of these two hyperparameters seem to correspond with longer training sessions, while larger values seem to correspond with shorter training sessions. As shown in figure 5.1b, for discount factors 0

to 0.7, training duration remains relatively the same. However, for discount factors 0.7 to 1, the training duration suddenly increases dramatically with a steep incline. Last but not least, according to figure 5.1c, the batch size hyperparameter's relationship with training duration is somewhat unclear since its graph shows a nonlinear plot.

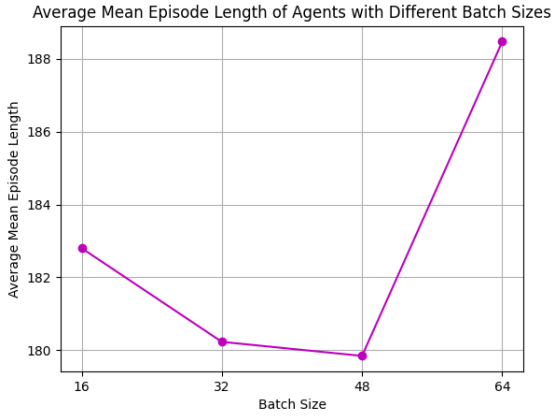
Another interesting find we want to point out is that the ERDF hyperparameter has the largest range of training durations. It also has the smallest range of tested values separated by the smallest intervals. By contrast, the batch size hyperparameter has the smallest range of training durations. Furthermore, it has the largest range of tested values separated by the largest intervals.



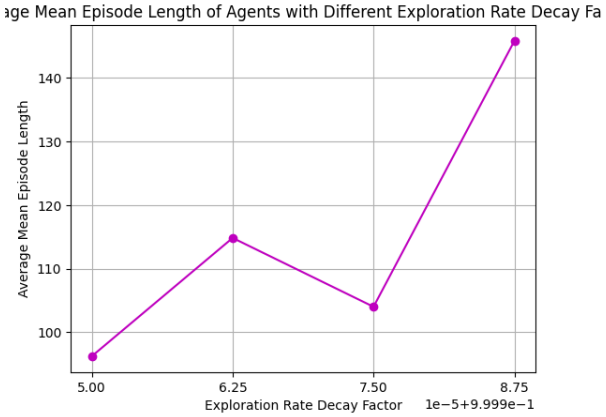
(a) Different Learning Rates



(b) Different Discount Factors



(c) Different Batch Sizes



(d) Different Exploration Rate Decay Factors

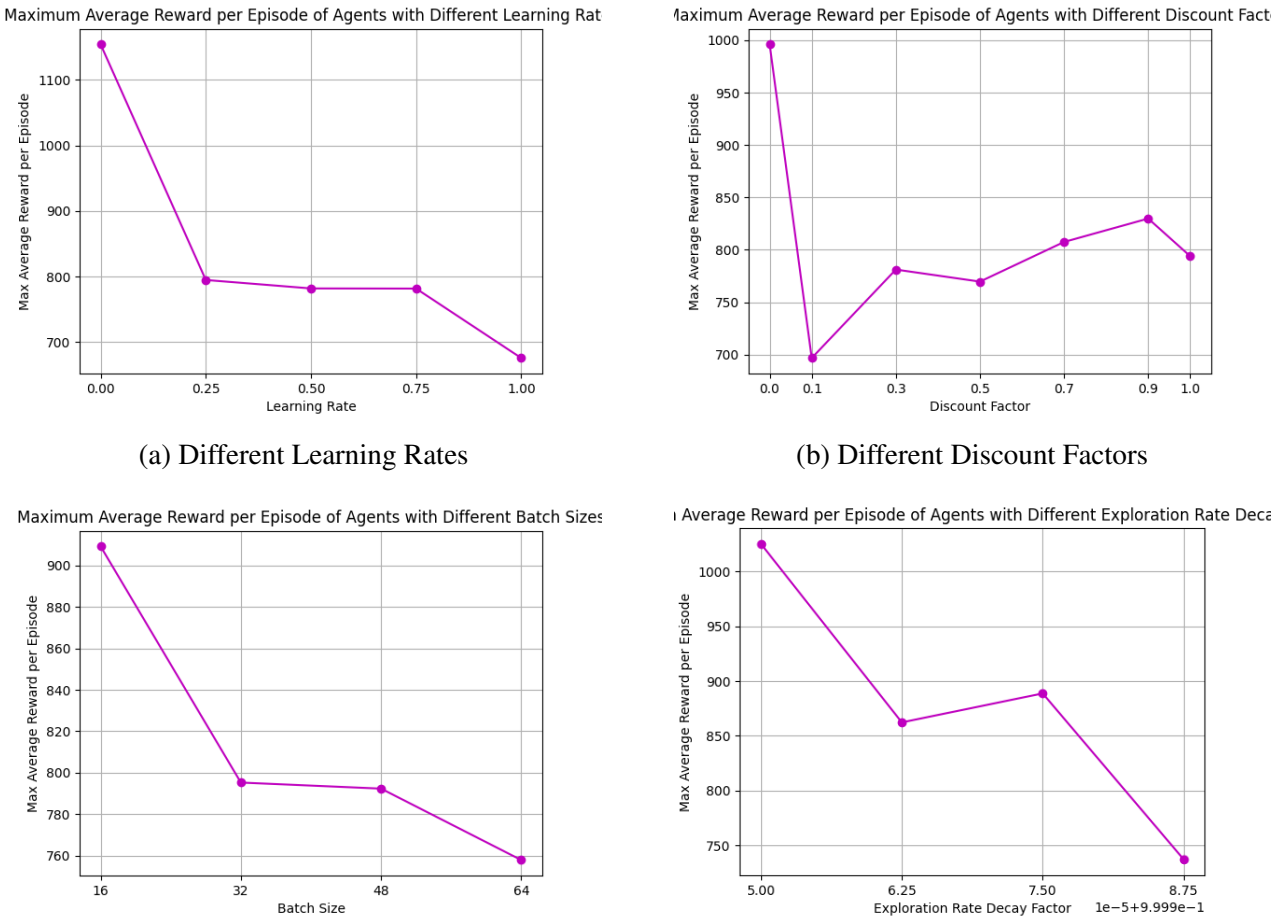
Figure 5.2: Average Mean Length of Episodes of RL Agents

Now, as shown in figure 5.2a, the learning rate hyperparameter appears to have a negative linear relationship with episode length. By stark contrast, according to figure 5.2d, ERDF seems to have a positive linear relationship with episode length. Unfortunately, as shown in figures 5.2b and 5.2c, batch size's and discount factor's relationships with episode length seems to be unpredictable. In figure 5.2c,

episode length decreases from batch size 16 to batch size 48, but then suddenly increases batch size 48 to batch size 64. Figure 5.2b's plot is even more difficult to decipher, with discount factor 0.1 corresponding to the smallest episode length and discount factor 0.7 corresponding to the largest episode length.

Interestingly enough, the same trends shown by the graphs in image 5.1 are also shown by the graphs in image 5.2. According to figure 5.2d, the ERDF hyperparameter appears to have the largest range in episode length. And in figure 5.2c, the batch size hyperparameter seems to have the smallest range in episode length.

5.2. Hyperparameters vs RL Performance



(a) Different Learning Rates

(b) Different Discount Factors

(c) Different Batch Sizes

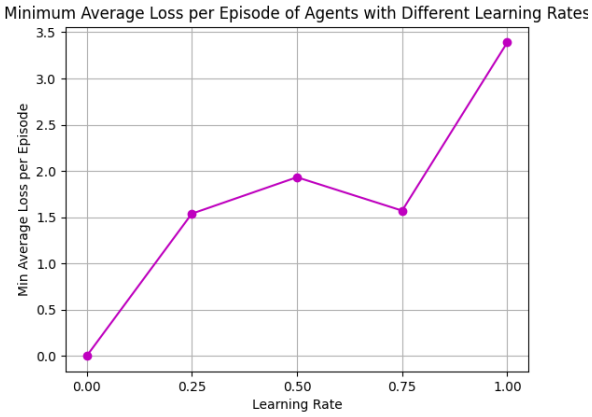
(d) Different Exploration Rate Decay Factors

Figure 5.3: Maximum Average Reward per Episode of RL Agents

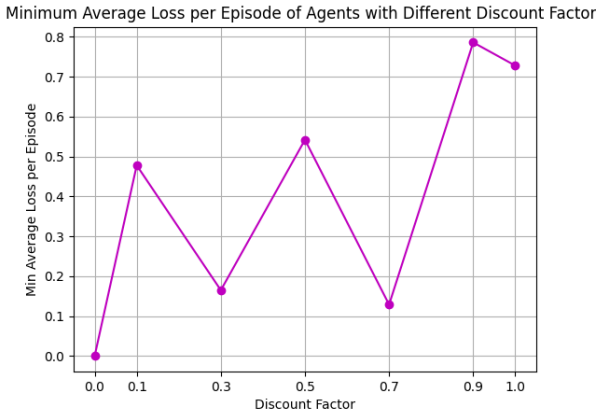
Now, we will evaluate some performance metrics of our 20 agents, as shown in images 5.3 (which shows the maximum average reward of different agents), 5.4 (which shows the minimum average loss of different agents), and 5.5 (which shows the time it took each agent to complete the 1st stage of Super

Mario Bros).

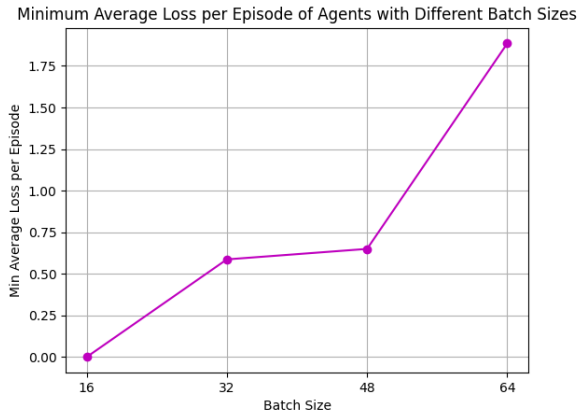
As shown in figures 5.3a, 5.3c, and 5.3d, the learning rate, the batch size, and ERDF hyperparameters seem to have a somewhat negative relationship with the maximum reward per episode. However, figure 5.3b shows a very unusual nonlinear relationship between the discount factor hyperparameter and the maximum reward per episode. Discount factors 0 to 0.1 shows a steep decrease in maximum reward per episode, but then discount factors 0.1 to 1 shows somewhat a steady increase in maximum reward per episode.



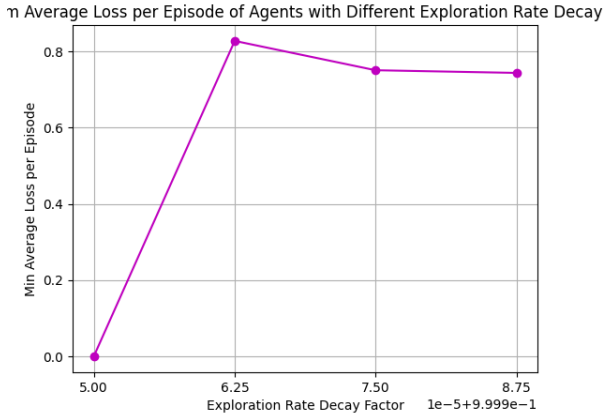
(a) Different Learning Rates



(b) Different Discount Factors



(c) Different Batch Sizes

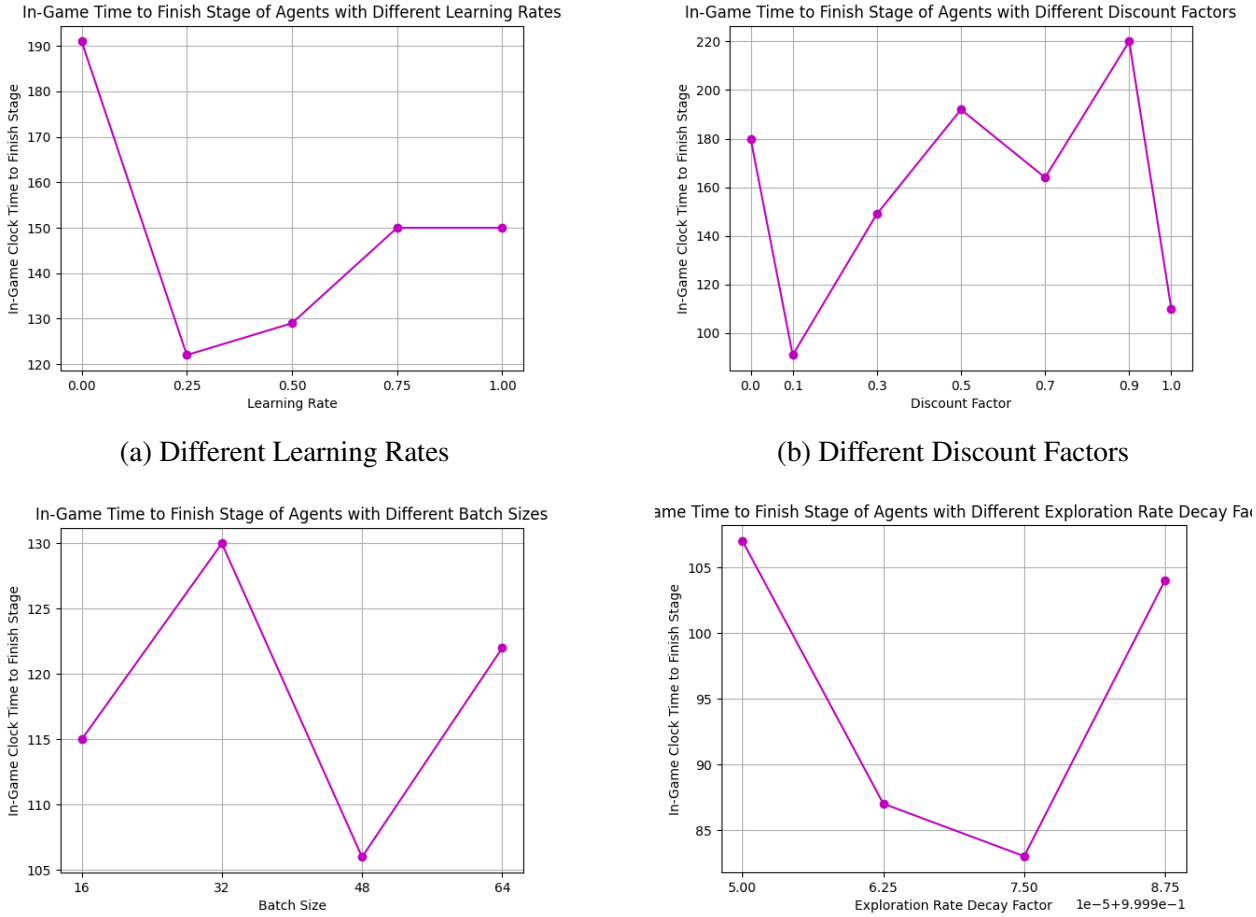


(d) Different Exploration Rate Decay Factors

Figure 5.4: Minimum Average MSE Loss per Episode of RL Agents

Now, in figures 5.4a and 5.4c, the learning rate and batch size hyperparameters seem to have a positive relationship with minimum loss per episode. In other words, small values of these two hyperparameters seem to correspond with small values of minimum loss per episode, while large values seem to correspond with large values of minimum loss per episode. As shown in figures 5.4b and 5.4d, the discount factor and ERDF hyperparameters have somewhat nonlinear relationships with the minimum loss per

episode. However, while the plot of the minimum loss per episode and discount factor seems to oscillate quite a lot throughout, the overall trajectory moves is a positive one. In other words, as the values of the discount factor increases, the minimum loss per episode seems to increase as well. Unfortunately, the trajectory in the ERDF and minimum loss per episode plot isn't as clear. There is a steep increase in minimum loss per episode from ERDF value 0.99995 to ERDF value 0.9999625, but then ERDF values from 0.9999625 to (approximately) 0.9999875 shows a slight decrease in minimum loss per episode.



(a) Different Learning Rates

(b) Different Discount Factors

(c) Different Batch Sizes

(d) Different Exploration Rate Decay Factors

Figure 5.5: Time to Complete Mario Stage of RL Agents

Now, as seen in image 5.5, all of the four hyperparameters seem to have pretty unpredictable, non-linear relationships with in-game performance time-wise. However, most of the graphs, except for the discount factor hyperparameter's graph in figure 5.5b, show an overall negative trajectory. Another find of interest is that the agent trained with discount factor 0.9 has the worst in-game performance, completing the stage in approximately 220 in-game seconds. Meanwhile, the agent trained with ERDF 0.999975 has the best in-game performance, completing the stage in less than 85 in-game seconds.

6. Discussion

According to our findings in the previous section, it appears that the exploration rate decay factor hyperparameter has the most significant influence on training speed. This is because it seems to cause significant differences in training duration and episode length with just small value differences. Recall that the exploration rate decay factor hyperparameter controls the balance shift over time between exploration and exploitation of each agent, or in other words, how fast the agent shifts from exploring more to exploiting more (Sheryl, 2024). Also recall that from figure 5.1d, we observed an overall negative trajectory in training duration against the tested values of the exploration rate decay factor hyperparameter. From all this, it is perhaps reasonable to think that a larger exploration rate decay factor, which speeds up the balance shift, may correspond to a shorter time span to learn to complete a task. Futhermore, the in-game performance metric of the exploration rate decay factor hyperparameter suggests that somewhat large exploration rate decay factor values might produce good-quality task performance. Therefore, we believe that this particular hyperparameter may require delicate fine-tuning to find just the right spot that yields both faster training speed and good-quality task performance.

From our findings, the discount factor hyperparameter appears to have the second largest influence on training speed, only from discount factors 0.7 to 1 (where we observed a steep increase in training duration). Perhaps that is because larger discount factors encourages the agent to learn more from future rewards, a process that might take up extra time to execute. However, as shown in figure 5.3b, this appears to help the agent to gain more rewards, even though it may not guarantee minimal loss nor good-quality task completion. In the long run, the discount parameter might aid in optimizing the completion of a task, but to learn to simply complete a task, smaller discount factors seem to contribute towards faster training speed, smaller loss, and, perhaps, better task completion performance.

Next on the list is the learning rate hyperparameter, which seems to have one of the least influences on training speed. That is because it seems to have caused noticeable but not super significant differences in training duration and episode duration with small, but not the smallest differences in value. However, we also observed that larger learning rates seem to correspond to shorter training durations. Since learning rates control the size of the steps taken during gradient descent, it's reasonable to think perhaps they also influence the speed of convergence. Therefore, larger values of learning rate corresponds to larger steps in gradient descent, which may lead to faster convergence and a shorter time span to learn a task.

Unfortunately, there is no clear relationship between batch size and training duration nor episode duration. However, they seem to have the smallest influence, since large variations in batch sizes doesn't seem to correspond to large difference in training duration nor episode duration. This is perhaps because they determine how much agents will randomly sample from memory, which is most likely a quick operation that won't effect execution much. They also deliver quite unpredictable task performance results, as shown in figure 5.5c. This metric may depend on how effective recalled samples are in teaching agents the correct actions, which is difficult to predict.

7. Conclusion

Hyperparameter optimization may be one of the key approaches to achieving faster training speed in deep reinforcement learning agents. We demonstrated four different common hyperparameters in deep reinforcement learning and their unique influences on the training speed of agents. It turned out that larger learning rates, smaller discount factors, and larger exploration decay factors may all help agents learn faster. However we also must be mindful of any performance tradeoffs that might occur as a result. If we had more time, we would love to not only test more hyperparameters, but also test them longer to observe how quickly they can converge to an optimal.

References

Achiam, J. (2018). Spinning Up in Deep Reinforcement Learning.

Agostinelli, F., Hocquet, G., Singh, S., & Baldi, P. (2018). From reinforcement learning to deep reinforcement learning: An overview. In L. Rozonoer, B. Mirkin, & I. Muchnik (Eds.), *Braverman readings in machine learning. key ideas from inception to current state: International conference commemorating the 40th anniversary of emmanuil braverman's decease, boston, ma, usa, april 28-30, 2017, invited talks* (pp. 298–328). Springer International Publishing. https://doi.org/10.1007/978-3-319-99492-5_13

Aljadery, M., & Sharma, S. (n.d.). The path forward: A primer for reinforcement learning. https://web.stanford.edu/~sidshr/uploads/rl-primer.pdf

Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., Essen, B. C. V., Awwal, A. A. S., & Asari, V. K. (2018). The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, *abs/1803.01164*. http://arxiv.org/abs/1803.01164

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, *8*(1), 53. https://doi.org/10.1186/s40537-021-00444-8

Amber. (2019). Deep q-learning part 2: Double deep q-network (double dqn) [Accessed: 2024-11-30]. https://medium.com/@qempsil0914/deep-q-learning-part2-double-deep-q-network-double-dqn-b8fc9212bbb2

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. https://doi.org/10.1109/MSP.2017.2743240

Ashraf, N. M., Mostafa, R. R., Sakr, R. H., & Rashad, M. Z. (2021). Optimizing hyperparameters of deep reinforcement learning for autonomous driving based on whale optimization algorithm. *PLoS ONE*, *16*(6), e0252754. https://doi.org/10.1371/journal.pone.0252754

Barreto, A., Hou, S., Borsa, D., Silver, D., & Precup, D. (2020). Fast reinforcement learning through the composition of behaviours [Accessed: 2024-12-01]. https://deepmind.google/discover/blog/fast-reinforcement-learning-through-the-composition-of-behaviours/?form=MG0AV3

Chadi, M.-A., & Mousannif, H. (2023). Understanding reinforcement learning algorithms: The progress from basic q-learning to proximal policy optimization. *arXiv preprint arXiv:2304.00026*.

Eimer, T., Lindauer, M. T., & Raileanu, R. (2023). Hyperparameters in reinforcement learning and how to tune them. *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:259063671

Feng, Y., Subramanian, S., Wang, H., & Guo, S. (2023). Train a mario-playing rl agent. https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html

FutureMachineLearning.org. (n.d.). The critical role of discount factor in reinforcement learning [Accessed: 2024-12-01]. https://futuremachinelearning.org/the-critical-role-of-discount-factor-in-reinforcement-learning/?form=MG0AV3

Ganeshkumar, M. (n.d.). Minimizing time in training deep reinforcement learning agents [Accessed: 2024-12-01]. https://nusit.nus.edu.sg/technus/minimizing-time-training-deep-reinforcement/?form=MG0AV3

GDJ. (2023). Nes controller video game [Accessed: 2024-11-26]. https://openclipart.org/detail/274411/nes-controller-video-game

Jang, B., Kim, M., Harerimana, G., & Kim, J. W. (2019). Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, *7*, 133653–133667. https://doi.org/10.1109/ACCESS.2019.2941229

Jayakody, D. (2022). Double deep q-networks (ddqn) - a quick intro (with code). %5Curl%7Bhttps://dilithjay.com/blog/ddqn?form=MG0AV3%7D

Jomaa, H. S., Grabocka, J., & Schmidt-Thieme, L. (2019). Hyp-rl : Hyperparameter optimization by reinforcement learning. *CoRR*, *abs/1906.11527*. http://arxiv.org/abs/1906.11527

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

Krichen, M. (2023). Convolutional neural networks: A survey. *Computers*, *12*(8). https://doi.org/10.3390/computers12080151

Li, Y. (2018). Deep reinforcement learning. *CoRR*, *abs/1810.06339*. http://arxiv.org/abs/1810.06339

Li, Y. (2022). Reinforcement learning in practice: Opportunities and challenges. *arXiv preprint arXiv:2202.11296*.

Mack, D. (2018). How to pick the best learning rate for your machine learning project [Accessed: 2024-12-01]. https://www.freecodecamp.org/news/how-to-pick-the-best-learning-rate-for-your-machine-learning-project-9c28865039a8/?form=MG0AV3

Majid, A. Y., Saaybi, S., van Rietbergen, T., François-Lavet, V., Prasad, R. V., & Verhoeven, C. J. M. (2021). Deep reinforcement learning versus evolution strategies: A comparative survey. *CoRR*, *abs/2110.01411*. https://arxiv.org/abs/2110.01411

Maluso, S. (2021). How to accelerate deep reinforcement learning training [Accessed: 2024-12-01]. https://community.intel.com/t5/Blogs/Tech-Innovation/Artificial-Intelligence-AI/How-to-Accelerate-Deep-Reinforcement-Learning-Training/post/1342629?form=MG0AV3

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*. http://arxiv.org/abs/1312.5602

Nair, S. (n.d.). Dqn and hyperparameters: Studying the effects of the various hyperparameters [Accessed: 2024-12-01].

Nintendo. (2016). Nes classic edition manual [Accessed: 2024-11-26]. https://www.nintendo.co.jp/clv/manuals/en/pdf/CLV-P-NAAAE.pdf

O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. *CoRR*, *abs/1511.08458*. http://arxiv.org/abs/1511.08458

Paszke, A., & Towers, M. (n.d.). Reinforcement learning (dqn) tutorial [Accessed: 2024-12-01].

Saha, S. (2018). A comprehensive guide to convolutional neural networks – the eli5 way [Accessed: 2024-12-01]. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Scholten, J., & Kuijt, A. (2023). How to speed up deep reinforcement learning by telling it what to do [Accessed: 2024-12-01]. https://www.xomnia.com/post/how-to-speed-up-deep-reinforcement-learning-by-telling-it-what-to-do/?form=MG0AV3

Shehu, Y. (n.d.). The history of reinforcement learning [Accessed: 2024-12-01]. https://researchdatapod.com/history-reinforcement-learning/?form=MG0AV3

Sheryl. (2024). Epsilon-greedy strategy in deep reinforcement learning [Accessed: 2024-12-01]. https://quantrl.com/epsilon-greedy-strategy-in-deep-reinforcement-learning/?form=MG0AV3

van Hasselt, H. (2010). Double q-learning. *23*, 2613–2621.

van Hasselt, H., Guez, A., & Silver, D. (2015). Deep reinforcement learning with double q-learning. *CoRR*, *abs/1509.06461*. http://arxiv.org/abs/1509.06461

Walther, P. (2021). Experimental quantum speed-up in reinforcement learning agents. *Nature*, *589*, 48–53. https://doi.org/10.1038/s41586-021-00001-2

Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, *415*, 295–316. https://doi.org/https://doi.org/10.1016/j.neucom.2020.07.061

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). Dive into deep learning. *CoRR*, *abs/2106.11342*. https://arxiv.org/abs/2106.11342

Zhao, X., Wang, L., Zhang, Y., Han, X., Deveci, M., & Parmar, M. (2024). A review of convolutional neural networks in computer vision. *Artificial Intelligence Review*, *57*(4), 99. https://doi.org/10.1007/s10462-024-10721-6