Mortal (Q)ombat: Training an AI agent with Reinforcement Learning to Play a Brawler Game

Anthony Weng
Department of Computer Science
Stanford University
`ad2weng@stanford.edu`

Aviad Lengo
Department of Computer Science
Stanford University
`lengo@stanford.edu`

1 Abstract

Deep reinforcement learning formulates tasks as reinforcement learning environments and has agents leverage AI architectures to motivate decisions and learn from interactions. Here, we apply a popular DRL technique, deep Q-learning, in the context of a brawler-type video game. We train 2 agents: one which trains by fighting itself and one which trains by fighting a random agent. Both agents are evaluated against a random and human agent, both displaying better-than-random levels of skill but being unable to defeat the human agent once. Against the random agent, the AI trained by fighting itself outperforms the AI trained by fighting the random agent. Possible explanations for these results and recommendations for future deep learning implementations are discussed .

2 Introduction

"Brawlers" are a sub-class of video games in which a player pilots a character (or "fighter") against the fighter piloted by their opponent. Popular entries in sub-class include titles like *Mortal Kombat* (the namesake of this project), *Street Fighter*, *Tekken*, and *Marvel vs. Capcom* [1].

While they can be differentiated by their character rosters, visual aesthetic, etc., most brawlers share this feature: character actions cause deterministic effects, meaning the player with more "skill" will win a given game. In the context of brawler games, "skill" can be described as a combination of proactivity and reactivity, which describe a player's ability to predict what an opponent will do and plan accordingly and to make dynamic adjustments when an opponent deviates from said prediction, respectively. Here, we attempt **to utilize reinforcement learning to train an AI agent to play a brawler game with a better-than-random and (ideally) human-comparable level of skill.**

3 Background

Skill-based games have long been a popular arena for evaluating the capabilities of AI methodologies [1] [2] [3]. Some of the most successful game-playing AI are developed with games being formulated as reinforcement learning (RL) environments, such as in DeepMind's AlphaGo and *Stratego* engines [4] [5]. These observed successes have emerged from and contribute to a field dedicated to AI-as-RL-agents, deep reinforcement learning (deep RL, DRL). Deep RL now entails numerous methodologies and applications beyond game-playing, including robotics, natural language processing, education, etc. [6] [7]. Deep RL has been applied in the context of brawler games, but AI agents are typically trained against and have the objective of defeating pre-programmed game AI (whose behavior is often based on simple rules subject to some randomness) rather than humans [8] [9]. Thus, developing an AI agent whose objective is to compete with human agents (as is done in this work) represents a novel application of deep RL, possessing the potential of further illuminating the capabilities and limits of both AI and RL methodologies.

4 Methodology

We first provide a summary of the AI agent training and evaluation methodologies. In-depth descriptions and associated justifications for individual implementation aspects follow. If certain aspects are not discussed explicitly, the authors determined them through empirical search[2].

[1]Gameplay image credits (left to right): *Mortal Kombat*, *Street Fighter*, *Tekken*, *Marvel vs. Capcom*,.
[2]All project resources will be posted at `https://github.com/roflauren-roflauren/cs238`.

4.1 Overview

We create a simplified brawler game with limited mechanics and define the reinforcement learning environment. We employ a deep Q-learning approach to train AI agents. We train two different types of agents, one which corresponds to the AI agent training by fighting itself and one which corresponds to the AI agent training by fighting a random agent. Key choices in the implementation of the training process include: an ϵ-greedy action selection regime while training agents; use of training and target networks; and use of an experience reply buffer. Each type of trained AI agent is evaluated against two opponents, a random agent and a human agent.

4.2 Training methodologies

4.2.1 Simplified brawler game mechanics

To allow for greater control in defining the reinforcement learning environment and for the ease of identifying and describing emergent AI agent behaviors, we elect to create a brawler game from scratch. The salient mechanics of the game are as follows (key terms are *italicized*):

- Each agent pilots a *fighter* with a fixed set of *moves* and finite *health pool*.
- Each fighter has access to the same set of moves:
  - *Movement-based moves* move one's fighter around the gameplay screen and include having one's fighter: move left, move right, or jump.
  - *Weapon-based moves* have the fighter use of their weapons to administer/attempt to block an attack and include: attack 1 (a medium-damage, short-range attack), attack 2 (a low-damage, medium-range attack), and parry (a non-damaging move which can block incoming attacks).
- The *objective* of the game is for each agent to sequentially input moves for their fighter such that their opponent's fighter's health pool is reduced to zero before their own fighter's health pool is.
- Attacks (e.g., attack 1, attack 2) which connect with an opponent's *hitbox* (a rectangle used to represent an object's location and check for collisions with other hitboxes) decrease the opponent's health pool by a fixed amount (different for each attack) – i.e., they do *damage* to the opponent.
- A parry whose hitbox successfully collides with an attack's hitbox nullifies the attack – i.e., the attack is cancelled and no damage is done to the parrying fighter – and the attacking player is administered an *action cooldown* (a set number of timesteps which must pass before any movement- or weapon-based move can be performed).
- To prevent a situation whereby the winner is largely determined by whichever agent can provide move inputs (through key presses) to their fighter faster, move-specific cooldowns are also implemented for jumping, attacking (attacks 1 and 2 not distinguished), and parrying.
- Games consist of a single round. During agent evaluation, no time limit (i.e., the maximum number of timesteps – equivalently, gameplay frames – which can pass before the game is deemed to have ended in a tie) is enforced. During agent training, a time limit is enforced.

4.2.2 Reinforcement learning environment definition

In the context of the brawler game, we define the experience tuple $(s_t, a_t, r_t, s_{t+1}, d_t)$ elements for timestep t of the reinforcement learning environment as follows:

- State (s_t): a vector $\mathbf{v} \in \mathbb{R}^{43}$ with the following information ([descriptors] - [cardinality]):
  - the x- and y-coordinates of agent's and opponent's fighters - 4;
  - the absolute differences in the x- and y-coordinates of the fighters - 2;
  - the health pool values of the agent's and opponent's fighters - 2;
  - one-hot encodings of the agent's and opponent's status[3] in s_{t-1} - 16;
  - the agent's and opponent's action, jumping, attacking, and parrying cooldowns - 8;
  - agent and opponent binaries encoding: hit, running, jumping, attacking, or parrying truth values in s_{t-1} - 10;
  - the number of eclipsed frames in the current game - 1

[3]One of: idle, running, jumping, attack 1'ing, attack 2'ing, receiving a hit, dead, or parrying.

- Action (a_t): an integer value in the range 0 to 15 (inclusive) which maps to some combination of 4 possible movement-based moves (do nothing, move left, move right, jump) and 4 possible weapon-based moves (do nothing, attack 1, attack 2, parry). Index values are unraveled in row-major fashion to map to a tuple of coordinates in a 4×4 matrix representing all possible combinations of movement- and weapon-based moves. See here for details.
- Reward (r_t): for non-terminal states, $r_t =$ health of agent's fighter $-$ health of opponent's fighter $-$ the integer percentage of the maximum game frames which have passed. For terminal states, $r_t = 10,000$ for the player whose fighter is alive (health > 0), $r_t = -10,000$ for the player whose fighter is dead (health $= 0$), and $r_t = -5,000$ for both players if the the maximum number of gameplay frames occurs before either fighter dies.
- State terminality (d_t): $d_t = 1$ (i.e., s_t is a terminal state) if the health of either fighter is 0 or if the maximum number of timesteps allowed for a game has occurred. Otherwise, $d_t = 0$.

In the process of defining the RL environment, key decisions and justifications include:

- Due to compute restrictions, we manually define the information contained in the state vector; see Section 7 for commentary regarding the potential benefits of a more loosely-defined state vector.
- Given that the objective of the game is to reduce the opponent's fighter's health to zero, the reward function was crafted to reward agents who are: (1) more likely to achieve this objective (captured by a higher self-favoring health differential implying more future timesteps and available riskiness in executing future actions); and (2) able to achieve this objective more quickly (captured by the percentage of max frames penalty which increases as the additional timesteps pass).

4.2.3 Deep Q-learning architecture

The complexity of the environment's state-action space incentivized the use of model-free learning, and a lack of the expert knowledge helpful to defining exploration policies encouraged our use of an offline learning algorithm. Q-learning satisfied both of these criterion, but vanilla Q-learning (Q-learning with look-up tables) would require massive memory overhead given the magnitude of the state-action space. Given this and our desire to have agents simulate multiple layers of abstract decision-making, we choose to employ deep Q-learning for agent training[4].

Deep Q-learning is a version of Q-learning in which a neural network is used to represent the Q-function. The neural network accepts the state vector as input and provides as output the estimated Q-values for each possible action from said state. Our application specified a 4-layer neural network (input-hidden-hidden-output) structure with the layers having 43, 34, 25, and 16 nodes, respectively. Non-input layers receive a linear transformation of the prior layer's input, with the input and first hidden layers' post-linear transformation outputs also being passed through the ReLU activation function. Linear transformation parameters are initialized using Xavier uniform initialization.

In the context of a neural network, rather than conducting $Q(s, a)$ updates from singleton experiences, updates are performed over network parameters through backpropagation of network loss. We can formulate loss for a single state-action value as:

$$\text{Loss}(s_t, a_t) = [Q(s_t, a_t) - (r_t + (1 - d_t) \cdot \gamma \max_{a'} Q(s_{t+1}, a'))]^2 \tag{1}$$

The implied objective function in (2) is mean squared error, which is what we use as our loss function. We propagate network loss using the Adam optimizer (a variant of SGD, i.e., stochastic gradient descent) given Adam's tendency to converge more quickly than SGD [10] and the limited compute available for the project. We train two AI agents (i.e., two sets of neural network weights), one which is trained by fighting the AI agent itself and the other which is trained by fighting a random agent. The deep Q-learning architecture was implemented through PyTorch and NumPy.

4.2.4 Other key implementation choices

1. *Training an AI agent by playing itself:* Mentioned in Section 4.2.3, training AI agent by having it play itself is a paradigm popular in modern RL, with a recent example being DeepMind's Stratego engine, DeepNash [5]. In a press release about DeepNash's development, DeepMind explains employing this paradigm allows the engine's playstyle to converge to a Nash equilibrium which "is unexploitable over time" but "only if facing a similarly perfect opponent [i.e., an opponent

[4]For a more complete justification, see Appendix A.

playing the same Nash equilibrium strategy]" [11]. To better assess the robustness of this agent development strategy, we deploy it here since Nash equilibria can be hard to identify in brawler games and that our agent may face non-perfect opponents.

2. *Use of training and target networks:* One prominent issue in deep Q-learning is the "moving goalposts" problem; i.e., using the same neural network to both select and evaluate actions can result in an agent which is more likely to select overestimated values, resulting in even more overoptimistic value estimates upon network learning [12]. Repeated occurrence of this phenomenon can lead to learning instability in estimation of Q-values. To limit the extent to which this phenomenon takes place, we use training and target networks. For a set number of experiences, the target network weights are frozen and used to evaluate actions. During this time, the training network weights can be updated and used to select actions. The weights from the training network are periodically copied to the target network.

3. *Experience replay buffer:* Another significant issue with deep Q-learning is a tendency to learn correlation between states; i.e., updates to the network's weights using sequentially-occurring states can inadvertently cause the network to learn the correlation between states and their state-action values, leading to poor network generalization. To combat this, we employ an experience replay buffer. The buffer stores experience tuples which, once it exceeds some fixed `batch_size`, has `batch_size` tuples randomly sampled from it which are used to train the network weights. Use of an experience replay buffer not only de-correlates training data [13] but can also aid with data efficiency and catastrophic forgetting (since experiences can be sampled multiple times).

4. *ϵ-greedy action selection:* A classic issue central to many RL strategies and domains is the exploration-exploitation tradeoff. To help remedy early agent convergence on a locally optimal solutions, we have agents employ an ϵ-greedy action selection regime during training; i.e., with some probability ϵ, the agent selects a random action for execution instead of the greedy action. Additionally, we enforce $\epsilon > 0$ even after many training episodes as doing so agents may explore different solutions indefinitely [14].

4.2.5 Hyperparameters

Table 1: Summary of key training hyperparameters

Hyperparameter	*Description*	*Value*
`game.max_frames`	Max # timesteps allowed in one training episode.	10,000
γ	RL environment discount factor	0.99
`num_games`	Number of training episodes for each AI agent.	1,000
`lr`	Learning rate for Adam optimizer.	6e-4
`num_exprs_per_copy`	# experiences per training and target network weights sync.	20,000
`batch_size`	Batch size for experience replay buffer sampling.	64
ϵ	Exploration probability for ϵ-greedy action selection.	1

Comments:

- $\gamma \approx 1$ to discourage agent myopia given numerous timesteps in one training episode.
- One training episode consists of one game, which ends when either fighter's health reaches 0.
- ϵ decays by 0.001 for the first 950 training episodes before remaining fixed at 0.05.

4.3 Evaluation methodologies

In Section 2, we define our problem to be training agents to play the brawler game with a "better-than-random and (ideally) human-comparable level of skill." To that end, each AI agent (AI trained versus AI, AI trained versus random agent) is evaluated against a random agent for 20 games and a human agent for 20 games. Games consist of a single round. The recruited human agent possessed limited, casual experience with brawler games (<1 year). To incentivize effort, a small reward of \$1 USD was offered for each game the human agent won against either AI agent (for a total reward of up to \$40 USD). AI agents do *not* learn at evaluation time (i.e., all network weights are frozen). Win percentages is recorded for each AI agent and evaluation agent pairing.

5 Results

5.1 Training Characteristics

Figure 1: Average training loss by agent.

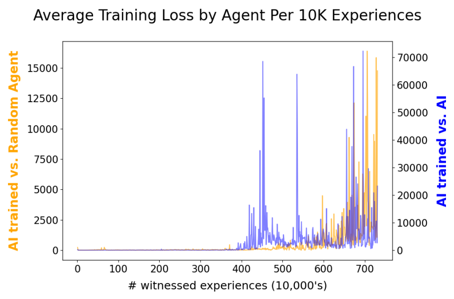
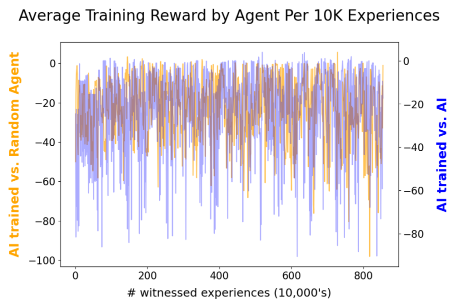


Figure 2: Average training reward by agent.



(Notation: we term the AI trained by fighting itself as the "self-taught AI" and the AI trained by fighting a random agent as the "random-taught AI." In both Figures 1 and 2, characteristics of the self-taught AI are depicted in blue and those of the random-taught AI are depicted in orange).

Training the self-taught and random-taught AI agents for 1000 games each required similar amounts of time: 23.299 hours and 25.739 hours, respectively. For both agents, we observe very little training loss through the first 4,000,000 experiences, and significant spikes afterward. For the self-taught AI, a few patterns of loss uptick-downtick pairings are present, whereas for the random-taught AI, only one significant, longer-lasting uptick takes place. Both agents present similar patterns in average training reward, with many consecutive uptick-downtick sequences taking place. We also note that the random-taught AI agent appears to have achieved the minimum reward for any collection of 10,000 experiences, but the self-taught AI agent appears to achieve a lesser reward more frequently.

5.2 Evaluation Games Summary

Table 2: Evaluation games win percentages by agent

Evaluated Agent	*Evaluated Against*	*Agent Win %*
Self-taught AI	Random Agent	75% (15/20)
	Human Agent	0% (0/20)
Random-taught AI	Random Agent	60% (12/20)
	Human Agent	0% (0/20)

Both the self-taught and random-taught agents were able to beat the random agent at a $> 50\%$ rate, thereby demonstrating a "better-than-random" level of skill. Interestingly, the self-taught agent performed better than the random-taught agent when fighting the random agent (presenting win rates of 75% and 60%, respectively), despite the random-taught agent's task-specific training. **Both AI agents were unable to beat the human agent a single time.** Though not rigorously timed, evaluation games against the human agents seemed to be of a much shorter duration ($\approx$ 1-2 minutes per game) than those against random agents ($\approx$ 3-5 minutes per game).

5.3 Qualitative Description of AI Agents' Behaviors

Against both evaluation opponents, the random-taught AI agent adopted the same strategy: stand on one edge of the screen, move very little, and repeatedly parry until an attack is successfully nullified before attempting an attack (a "peek-a-boo" strategy). Also against both evaluation opponents, the self-taught AI adopted a similar strategy to the random-taught AI with a notable change: rather than stay on one edge of the screen indefinitely, the self-taught AI agent would frequently run to the other side of the screen while administering attack 2's (the medium-range, low-damage attack) before resuming the peek-a-boo strategy. Both agents' did not exhibit well-timed evasive or defensive maneuvers when faced with the human agent's "rush-style" of play, consisting of running directly toward the AI agent while alternating between parries and attack 2's (the short-range, medium-damage attack) in rapid succession. In such scenarios, both agents would occasionally attack back but would mostly (unsuccessfully) attempt to parry.

6 Discussion and Analysis

6.1 On Deep Q-Learning and ϵ-Greedy

From Figure 1, we note that, rather than training loss decreasing over time, training loss increases on several orders of magnitude for both the self-taught and random-taught agents – especially in the last 1,000,000 experiences. This pattern of exploding loss is often attributed to the numerical instability of neural network weights or the gradients used for updating said weights during loss backpropagation. Valid strategies for remedying this issue include gradient clipping, normalizing network inputs, and otherwise limiting the magnitude of key components of the network.

However, it's worth considering why the issue of exploding loss arose in the first place. Note that the loss explosion takes place for both the self-taught and random-taught agents around training game 400-500 (1 training game $\approx$ 10,000 experiences). Given our ϵ-greedy ϵ decay scheme, at this point in training, agent behavior is close to becoming non-random for a majority of actions. This suggests that early on in agent training, the deep Q-network learned weights which predicted the state-action values of a largely random agent who encountered sparse rewards due to random actions not leading to frequent fighter interactions. Any reward accrued without fighter interaction (i.e., the $\%$ of max game frames eclipsed penalty) could be captured by the bias terms in the network, leading to network weights have extremely small magnitudes – thus, giving rise to the exploding loss observed later on in training when agent actions have become sufficiently non-random (causing fighter interactions to become frequent and the health differential component of reward non-sparse).

Though ϵ-greedy is a valuable strategy for ensuring state exploration, the training characteristics observed here suggest combining agent exploration regimes with some form of gradient clipping and/or input normalization is imperative for stable learning. Also, including more training episodes where the trained agent behaves randomly for a minimal proportion of actions may lead to more accurate state-action valuations as trained agents are not behaving randomly during evaluation.

6.2 Implications for the Self-Taught AI Training Paradigm

Two key results arise from comparing the performance of the self-taught versus the random-taught AI during the evaluation games: (1) the self-taught AI outperformed the random-taught AI when fighting a random agent; and (2) the self-taught AI did not fare any better against the human agent than the random-taught AI.

The first result is surprisingly given that task-specific training (here, training against a random agent) almost always leads to superior performance when evaluating multiple neural networks on the same training task. Thus, this suggests that the "train AI by having it play itself" paradigm can give rise to an agent with a highly-generalizable strategy. However, one must note that the observed out-performance by the self-taught AI may be noise given the small quantity of evaluation games.

The second result indicates that there are likely limitations to the generalizability of any self-taught AI's strategy. As noted in Section 4.2.4, DeepMind's *Stratego* engine, DeepNash, developed a Nash equilibrium strategy which was unexploitable over time but only if facing a "perfect" opponent who played the same Nash equilibrium strategy. Without injecting significant randomness, self-taught AI may converge to a Nash equilibrium playstyle during training. However, this learned playstyle may not be robust when: (1) evaluation takes place in a one-shot setting and (2) one's opponent does not play the expected Nash equilibrium playstyle - both of which are true for the human evaluation games in this case (since games consisted of a single round, and the human agent's "rush"-style of play was markedly different from the self-taught AI's "peek-a-boo" strategy). Under these conditions, and if multiple Nash equilibrium exist in a game, expecting a self-taught AI to achieve human-comparable performance may be unrealistic.

6.3 Key Hyperparameters to Optimize in Deep Q-Learning

Hyperparameter search and optimization is encouraged for training any neural network if maximal performance is desired, but some particularly important ones for deep Q-learning (as implemented here) likely include: `num_exprs_per_copy` and `batch_size` (see Table 1 for definitions). The exploding loss problem mentioned in Section 6.1 may (in part) be alleviated or exacerbated by selection of `num_exprs_per_copy` – syncing training and target network weights too frequently can lead to exploding state-action valuations (discussed in Section 4.2.4, item 2) but syncing infrequently may lead to intractable training times to convergence. Similarly, use of a low `batch_size` may lead to slow convergence and eliminate the results of using an experience replay buffer, but an extremely high `batch_size` may require excessive compute. In our work, we note alteration of these hyperparameters can significantly affect estimated state-action values and network convergence, and therefore recommend a principled search over them for future deep Q-learning implementations.

7 Conclusion and Future Work

We apply deep Q-learning to train AI agents to play a brawler game. Two agents are trained, a self-taught agent who learns by fighting itself and a random-taught agent who learns by fighting a random agent. We evaluate the performance of both against a random agent and a human agent, finding that both agents demonstrate a better-than-random level of skill but are unable to defeat the human agent even once. We also observe that the self-taught agent wins more against the random agent than the random-taught agent. Implications from and reasons for these observations are discussed.

For future DRL works in the brawler game space, there exist many promising directions for future research. Mentioned in Section 4.2.2, constructing a more loosely-defined state vector via feature templates could not only better approximate human processing (being holistic and more all-encompassing) but also lead to improved agent performance through greater information availability. Collating subsequent state vectors for input to a DRL architecture could encode an implicit notion of fighter motion and recent action history, further approximating human processing and potentially human performance. Beyond changes to the RL environment, one could make changes to the brawler game itself: e.g., one could create multiple characters with different abilities to see an AI agent can select a character to create a favorable match-up.

With this wealth of future directions, the authors hope this work can serve as both a baseline and inspiration for exploring the potential and limits of DRL techniques in ever-more-complex games.

7.1 Acknowledgements

Many thanks to: the CS 238 course staff for their support and feedback throughout Winter Quarter 2023; and Tenzin Yigha for being willing to play 40 rounds of the developed game in one sitting.

7.2 Work Division Statement

Both authors contributed significantly to the project codebase, report, and intermediate assignments. Anthony was responsible for implementing the deep Q-learning architecture classes. Aviad was responsible for conducting agents' trainings and evaluations. The authors co-wrote the final project report and intermediate assignments largely while working together in-person.

References

[1] Michael Freed, Travis Bear, Herrick Goldman, Geoffrey Hyatt, Paul Reber, and Joshua Tauber. Towards more human-like computer opponents. In *Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, number 1, pages 22–26, 2000.

[2] John E Laird. Research in human-level ai using computer games. *Communications of the ACM*, 45(1):32–35, 2002.

[3] Alexander Nareyek. Intelligent agents for computer games. In *Computers and Games: Second International Conference, CG 2000 Hamamatsu, Japan, October 26–28, 2000 Revised Papers*, pages 414–422. Springer, 2001.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[5] Julien Perolit, Bart De Vylder, Daniel Hennes, et al. Mastering the game of stratego with model-free multiagent reinforcement learning. *Science*, 378(6623):990–996, 2022.

[6] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[7] Yuxi Li. Deep reinforcement learning. *CoRR*, abs/1810.06339, 2018.

[8] Mehrabi Hasan. Street fighter ii is hard, so i trained an ai to beat it for me, May 2020.

[9] Andrew Ngai. Build an ai to play street fighter with python and stable-baseline (part 1), Mar 2020.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

[11] Mastering stratego, the classic game of imperfect information, Dec 2022.

[12] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[13] Shangtong Zhang and Richard S. Sutton. A deeper look at experience replay, 2017.

[14] Paavai Paavai Anand et al. A brief study of deep reinforcement learning with epsilon-greedy exploration. *International Journal Of Computing and Digital System*, 2021.

Appendix

A Deep Q-Learning Motivation

Given our definition of the reinforcement learning environment, there are approximately $|\mathcal{S}| \approx k^{43}$ potential states (where $k =$ the minimum number of values any element in the state vector can assume, i.e., 2). Then, defining a transition function $T(s'|s, a)$ for model-based learning could require on the order of $|\mathcal{S}|^2 * |\mathcal{A}| = k^{86} \times 16$ computations.

Given the intractability of creating a discrete state transition model, we elect to use a model-free algorithm. Q-learning was initially chosen due to its off-policy nature, which is advantageous as the we do not possess the requisite expert knowledge to define quality exploration policies for on-policy learning. Still, as Q-learning operates on state-action pairs, vanilla Q-learning (i.e., storing Q-values in a lookup table) would require a table with approximately $|\mathcal{S}| \times |\mathcal{A}| = k^{43} \times 16$ entries. To reduce the memory footprint and owing to our desire to create AI agents who could simulate multiple layers of abstract decision-making, we ultimately employ deep Q-learning for agent training.