2022-09

ANALYZING HUMAN-INDUCED PATHOLOGY IN THE TRAINING OF REINFORCEMENT LEARNING ALGORITHM

Atkinson, Brian R.

Monterey, CA; Naval Postgraduate School

http://hdl.handle.net/10945/71107

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ANALYZING HUMAN-INDUCED PATHOLOGY IN THE TRAINING OF REINFORCEMENT LEARNING ALGORITHMS

by

Brian R. Atkinson

September 2022

Thesis Advisor:	Geoffrey G. Xie
Second Reader:	Marko Orescanin

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE ANALYZING HUMAN-INDUCED PATHOLOGY IN THE TRAINING OF REINFORCEMENT LEARNING ALGORITHMS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Brian R. Atkinson	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Modern artificial intelligence (AI) systems trained with reinforcement learning (RL) are increasingly more capable, but agents training to complete tasks in safety critical environments still require millions of trial-and-error training steps. Previous research with a Pong agent has shown that some human heuristics initially accelerate training but cause agent performance to regress to a state of performance collapse. This thesis utilizes the FlappyBird environment to evaluate if the pathology is generalizable. After initially confirming a similar pathology in an unaided agent, comprehensive experimentation was performed with optimizers, weight initialization methods, activation functions, and varied hyperparameters. The pathology persisted across all experiments and the results show the network architecture is likely the principal cause. At a high level, this work illustrates the importance of determining the inherent capacity of an architecture to learn and model complex environments and how more systematic methods to quantify capacity would greatly enhance RL.

14. SUBJECT TERMS reinforcement learning, RL, AI, artificial intelligence	**15. NUMBER OF PAGES** 101
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

ANALYZING HUMAN-INDUCED PATHOLOGY IN THE TRAINING OF REINFORCEMENT LEARNING ALGORITHMS

Brian R. Atkinson
Captain, United States Marine Corps
BS, University of Colorado, Boulder, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2022

Approved by: Geoffrey G. Xie
Advisor

Marko Orescanin
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Modern artificial intelligence (AI) systems trained with reinforcement learning (RL) are increasingly more capable, but agents training to complete tasks in safety critical environments still require millions of trial-and-error training steps. Previous research with a Pong agent has shown that some human heuristics initially accelerate training but cause agent performance to regress to a state of performance collapse. This thesis utilizes the FlappyBird environment to evaluate if the pathology is generalizable. After initially confirming a similar pathology in an unaided agent, comprehensive experimentation was performed with optimizers, weight initialization methods, activation functions, and varied hyperparameters. The pathology persisted across all experiments and the results show the network architecture is likely the principal cause. At a high level, this work illustrates the importance of determining the inherent capacity of an architecture to learn and model complex environments and how more systematic methods to quantify capacity would greatly enhance RL.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AI	Artificial Intelligence
CPU	Central Processing Unit
CNN	convolutional neural network
CDF	cumulative distribution function
DNN	deep neural network
DQL	deep q-learning
DQN	deep q-network
GD	gradient descent
GPU	Graphics Processing Unit
MDP	Markov Decision Process
MB	megabytes
MCTS	Monte-Carlo Tree Search
NPS	Naval Postgraduate School
PG	Policy Gradient
prng	pseudo-random number generator
PLE	PyGame Learning Environment
ReLu	rectified linear unit
RL	reinforcement learning
RMSProp	Root Mean Squared Propagation
TAMER	Training an Agent Manually through Evaluative Reinforcement

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I remember talking to friends while in undergrad about their master's theses and remarking how miserable it would be to have to write a hundred page research paper, thinking to myself I would never do that. Funny how the universe works. Thankfully, this entire process has been far more enjoyable than I ever anticipated, and that is entirely due to the invigorating mentorship of Dr. Geoffrey X ie. Without his guidance and steadfast positivity, a task that once looked daunting became my passion, and I couldn't have asked for a better advisor. Thank you for entertaining my weekly pivots and for letting me run to ground every single crazy idea I could think of.

I also want to thank Professor McClure for effectively acting as a co-advisor for the last two months of this project, despite having no official ties to this work. Thank you for the patience and guidance: it had an oversized impact this work. It's extremely rare to find people who are so willing and eager to take on extra work with no incentive, NPS and its students are fortunate to have you on board.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

In reinforcement learning (RL), an agent is trained to complete a task inside of an environment using a series of rewards (similar to how humans would train a pet dog). In training the agent is provided input from the environment which it uses to base a decision on what action it should take. After millions of training steps, these agents can achieve superhuman performance across many domains such as chess or stock trading [1]. One of the drawbacks of this training is how time and computationally expensive it is to complete, especially in safety-critical applications or environments. Researchers have been investigating methods for accelerating the learning process of these agents using small pieces of human logic applied to the environment as a way of assisting the agent in learning a proper policy [2] [3] [4] [5]. There are several ways to incorporate this human feedback into the training cycle as outlined in [4] and [2] such as manipulating the reward received and policy shaping to push the agent into a more desirable policy space. Previous work has been done to assess the effect of human assistance in training, which will be better understood with this thesis [3] [6].

1.1 Problem Statement

The focus of this thesis is the training of reinforcement learning agents, commonly referred to as agents, to understand why incorporating human heuristics into a trainer has caused some agents' performance to collapse. In this study, a trainer influenced by human heuristics is referred to as a human trainer and the resulting performance collapse a pathology. This thesis continues research conducted by Hee, whose experiments showed human heuristics adversely effected the lifetime performance of agents [3]. In this research, human logic that is applied to the training of an agent is referred to as a human trainer or human heuristic and the agent a human-trained agent, which does not involve an actual person in the training of the agent.

1.2 Research Questions

- Is the pathology observed in Hee's agent specific to the Pong game or generalizable to other application contexts?
- Is there a mathematical model to explain why Hee's agent permanently regresses in performance?
- Can we successfully incorporate a human trainer using a different network architecture and human heuristic mode as proposed in [4]?

1.3 Organization

This thesis is organized into five chapters. Chapter 2 includes background information in the field of RL as well as previous research done in the field pertaining to this thesis. Chapter 3 discusses the architecture implementation of the neural network, the test environment, experiments to be conducted, and analytical tools to be used. Chapter 4 contains the results from all experiments as well as their analysis. Finally in Chapter 5 are the conclusions drawn from this research as well as recommendations for future work.

CHAPTER 2: Background

2.1 Reinforcement Learning Concepts

2.1.1 System Components

In reinforcement learning literature, the term **agent** refers to an artificially intelligent entity trained to perform some set of tasks for a specific environment. An agent can be a player in a game, a robot that moves through a room, or a stock market trader [3]. In this research, the agent will be a player in the popular mobile game FlappyBird. In Figure 2.1 the agent is the bird. Future references to the bird, agent, and player all refer to the agent described above.

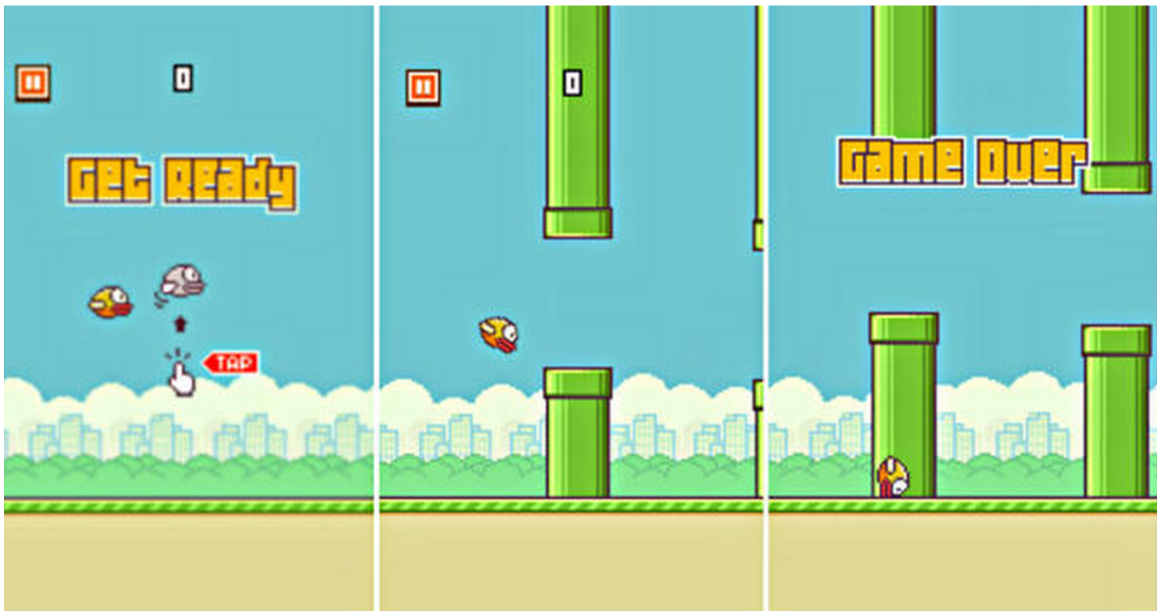


Figure 2.1: In this paper, the FlappyBird environment will be used to evaluate agents. Source: [7]

The **environment** is the world or space in which the agent is trained and operates within. This space can be real or virtual, and is described by how complex it is compared to others. An example of a simple environment would be within the Atari game Pong, which consists of three moving pieces: namely the two paddles and the ball. Conversely, a very complex environment would be the streets of New York City, where the integration of stoplights, vehicles, bicycles, pedestrians, and traffic create a chaotic world to learn to navigate through. The disposition of one of these environments at any point in time is referred to as a **state**, typically characterized by values of a set of environment variables. In this research, the environment is everything that is not the agent, and is shown in Figure 2.1. Key components of the environment are the green pipes, ground, and a noticeable gap in each vertical pair of pipes. The agent is attempting to navigate this pipe maze by flapping or not flapping its wings to pass through as many pipe-gaps as possible.

Throughout the training process, the agent will utilize observations of the environment to determine the appropriate **action** to take, for which it receives **rewards** in the form of a scalar value. Similar to how one envisions training an animal, RL uses this reward to either reinforce or disincentivize behaviors. These rewards can be received at any point in training, on which this paper will go further into detail in future sections. In some environments, the agent will take many actions before ever receiving a reward. In these situations, it is difficult to conclusively say which in the sequence of actions was responsible for the end-result and reward. To address this issue, the reward will be applied to all actions using an exponential decay that favors recent actions over older ones. The rate of decay of the reward signal is called a **discount factor**, whose value will range from 0 to 1. With this environment, the agent can receive 2 different rewards: +1 for successfully navigating through a pipe gap, and -5 touching the floor, a pipe, or going into the ceiling.

At each time step the agent observes its current state, and uses its **policy** to determine what action it needs to take next. There are many ways to implement this policy, with the main implementations being deterministic and stochastic. Deterministic policies can be viewed as static relations of states to actions (if in state X always take action Y). Stochastic policies utilize probabilities sampled from a distribution to determine what action to take. Though this may appear to be frivolous, this randomness serves to encourage policy-space exploration, preventing the agent from greedily focusing on a small subset of all possible strategies and helping to discover more effective ones. An additional benefit of stochastic behavior is that if the agent performs a balanced mixture of pursuing known strategies and randomly exploring new ones, it will discover all possible strategies given enough time.

An agent is trained inside of an environment over the course of many **episodes**. Each episode consists of the interaction between the agent and the environment for several states, where the agent perceives the environment at each state, decides what action to take, and does it. In this thesis, an episode is synonymous with the length of one game.

Simply put, an RL algorithm takes as input the current state of the environment the agent is in, and outputs an action for that agent to take in the environment. Every action the agent makes is rewarded or punished based on the rules of the environment. The agent uses its policy to decide which action to take in the current environment. Some actions will cause the process to change state and some of those state transitions will yield a reward. After taking an action from a state, the agent repeats this observe-decide-act cycle until it reaches a terminal state. A visual depiction of this process, known as a Markov Decision Process (MDP), is shown in Figure 2.2. In the case of FlappyBird, the agent has 2 possible actions: flap its wings to move up, or do nothing and allow gravity to pull it down.

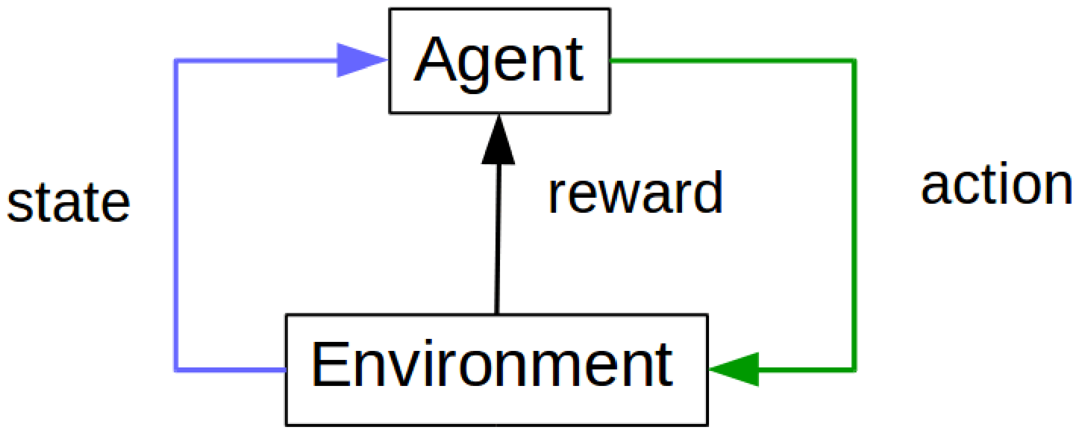


Figure 2.2: A simple depiction of a Markov Decision Process. The bird agent will be given a current game frame, use it to determine what action to take (i.e. flap its wings or let gravity pull it down), passes its action to the environment, and will receive some form of reward. Source: [7]

2.1.2 Training Methodology

The reward signal is the most critical component of training an agent. RL takes many different forms, but they all use the reward signal to augment their policy. In general, training an agent is a greedy optimization problem, where the agent seeks to find an optimal solution to its task by choosing the action with the highest expected future return. This is something that humans do intuitively, judging what to do based on what the perceived final outcomes would be and choosing to do what we believe will yield the best result.

This is simple enough in concept, but becomes increasingly difficult as the agent may sometimes be in an environment where a slight misstep early in an episode yields catastrophic results many time steps in the future, as is the case in games like chess. The task of evaluating the reward potential of an action can be intractable depending on the environment, but the need to judge future reward necessitates the innovation of several ways to do so. The following subsections will briefly explain some of the main variations of RL that all attempt to optimize the reward signal through different means.

2.1.3 Q-Learning

This first method of estimating future rewards is simplest in concept. The agent is initialized with what can be thought of as a totally-defined action map that defines an action to take for every possible state. This action map includes an expected value r for the discounted future reward of each entry, which is referred to as a quality-value or Q-value. An optimal Q-value for a state-action pair is determined by taking the sum of discounted expected average future reward. The discounted future reward is calculated by taking the maximum Q-value for each available actions resulting next state and applying the discount factor to it [8]. As the agent goes through training episodes it updates the Q-values for the state-action pairs it encountered. This process requires a reward signal (which in some environments only occurs at the end of the episode) to be realized, and as a result many episodes are required to refine the set of Q-values to a level that is acceptable for the agent to stop training.

As part of this training methodology, the agent needs to interact with the environment in a way that adequately discovers all possible state-action pairs and accurately estimates what their expected returns are. This is done through a random play strategy, which attempts to map the entire state-action space. For environments with small state-action spaces, this method of training agents works very well, but as the environments become even slightly more complicated the state-action space becomes impossibly large to map with satisfactory depth of exploration. The shortfall of is that it effectively optimizes a large lookup table of actions based on training data, leading to poor generality beyond the original task. A high performing FlappyBird agent trained by Q-learning would perform miserably at a simple game like Pong because the two sets of environment states are totally dissimilar.

2.1.4 deep q-learning (DQL)

To address the issue of large state spaces, a variation of Q-learning was created to model a function whose output is the estimated Q-value instead of directly mapping Q-values to state-action pairs. When it was invented, DQL leveraged advances in the efficacy of deep neural network (DNN)s. In training, the agent learns how to approximate future rewards for any state-action pair, because the weights of the neural network work symbiotically to approximate a hidden function. In reality there is no obvious mathematical function to model Q-values, but the trained network will create one to best fit the state-action pairs to future rewards. With this advance the agent does not have to experience every possible state-action pair multiple times to learn how to properly navigate the environment, it just needs to iterate through a reasonably diverse set of state-action pairs to estimate how they are related to the expected future reward.

Though DQL solves some of the issues presented by large state spaces, it struggles in other ways. DQL is optimized for environments where the available actions are discrete and involve relatively low dimensions, as is the case in most legacy video games. It is not nearly as effective when learning to control an agent whose actions are non-discrete with a high dimensionality such as with the joints or limbs of an human. As an example, the human arm has no less than 7 degrees of freedom, with a continuous range of actions [9]. For DQL to learn to control a human arm, it would have to attempt to map optimal state-action pairs over an enormously large space, which would require an incredibly inconvenient amount of iteration and resources.

2.1.5 Policy Networks

The need to address high-dimensional environments and agents with non-discrete action spaces prompted exploration of a new training methodology called Policy Gradient (PG). Similar to a DNN, the policy network contains multiple hidden layers of neurons bridging the input and output neuron layers. The output layer can take many forms, but in simple environments like FlappyBird, the output layer can be a single neuron that represents a choice: have the agent go up or do nothing. Many implementations use a stochastic policy where a probability of the agent moving up or down is produced by the output layer. The actual decision-making occurs when the algorithm draws a random sample taken from an arbitrary number distribution such as Normal or Uniform and compares the sample against the produced probability from the network. If the sample value is less than or equal to the network output, the agent will flap its wings, otherwise it will do nothing and let gravity take effect [10].

One major drawback to value-based methods like DQL is that in learning, adjustments to the expected value function only truly affect change on the output for the singular state-action pair pertaining to that frame, with changes to other state-action Q-values being sparse and very small in scale. Because of this pseudo-independence between state-actions with respect to updates, value-based models take considerably longer to have updates propagate through the network, once again drastically increasing training time [11]. Bearing this fact in mind, the benefits of policy-based models become much more valuable: PG uses the modeling capability of networks to parameterize the policy itself [11]. By choosing to use the network as a representation of the policy, training can be further accelerated by using gradient descent (GD) to optimize the reward received by making changes across all the parameters of the network.

2.2 Performance Collapse

A critical component to achieving general intelligence in agents is the ability for an agent to continually learn new tasks while maintaining its previous level of mastery in those it already can perform. Layered neurons in the neocortex of the mammalian brain form what are referred to as a neocortical circuit, which holds on to critical components of information used to do tasks that may have been learned long ago. These circuits are a centerpiece of the human brain's high-level computing ability and allow us to interrupt our learning of a task for a length of time before resuming it again (like the learning of an instrument, where incremental improvements are made over several years as the student takes periodic lessons) [12]. As a task is learned, additional excitatory synapses are formed between neurons to create a task-specific pathway for signals to propagate through the layers of the brain. Unfortunately for neural networks, it is not necessarily practical nor scalable to create additional connections between layers as tasks are learned, as the number of parameters for the network would quickly become too large to store in main memory.

At the moment, neural networks struggle to be able to completely learn multiple tasks in sequence without making significant sacrifices in their ability to perform any task other than the one it was most recently trained on. During training, the network will make slight modifications to the weight matrices that it is comprised of during backpropagation, and those incremental changes will eventually become destructive to the previous task. As seen in sequential learning of multiple tasks, it is common for agents to suddenly reach a point where they will no longer be able to complete tasks it previously mastered, a phenomena that is termed *Catastrophic Forgetting* in the RL fields of continual and transfer learning.

In Hee's research, human logic was used to assist the agent in action selection. In testing regular and human-assisted agents were trained to play Pong with the objective of understanding how human logic could decrease the time and number of repetitions required to learn the game. As seen in Figures 2.3-2.5 Hee's human-assisted agents (referred to as Hee's agents) initially learned Pong at a drastic rate, but began to collapse around the 20,000 episode point whereas those unassisted agents continued to learn [3].

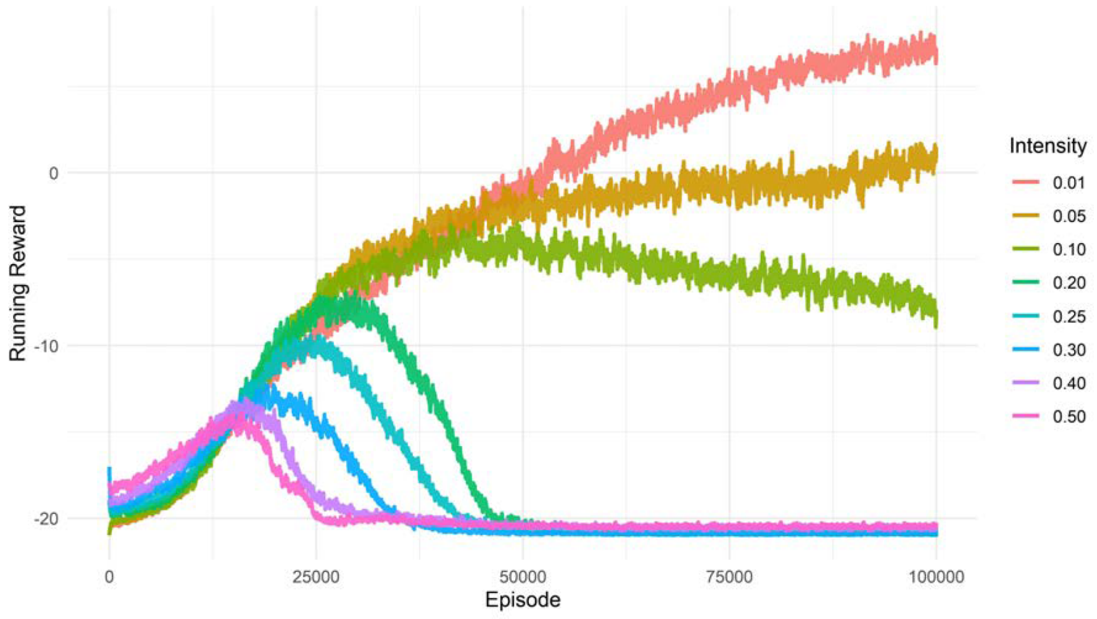


Figure 2.3: Agents with human intensity levels greater than 0.1 without decay all experienced forgetting. Source: [3]

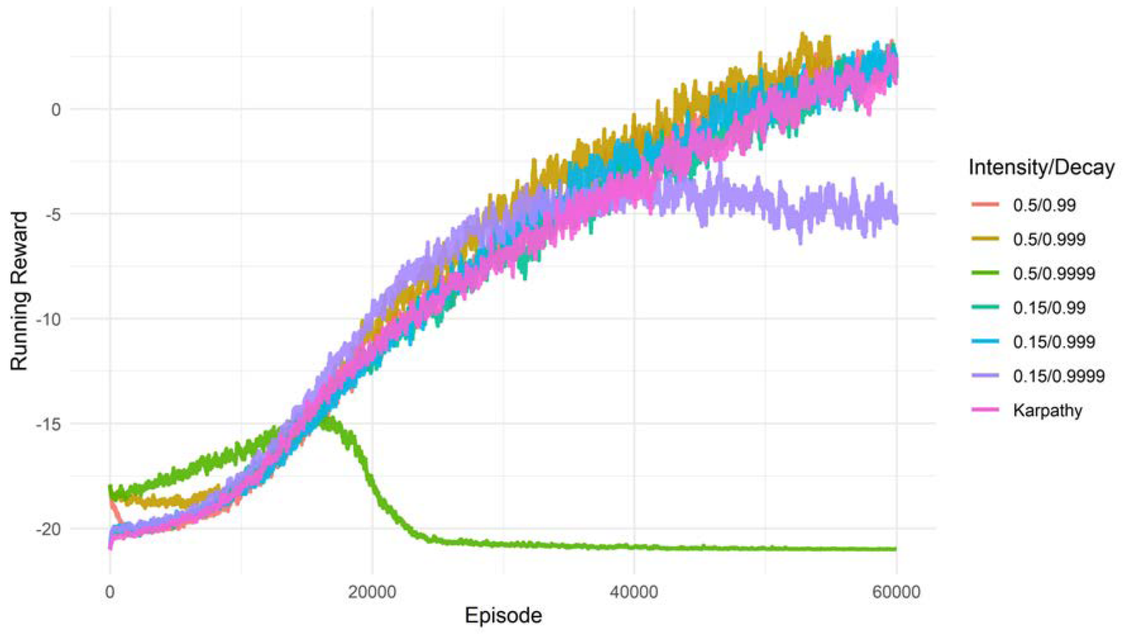


Figure 2.4: Agents trained with near-zero intensity or moderately low intensity and high decay levels did not collapse, but in effect were more akin to regular agents than being assisted by human logic. Source: [3]

Hee's initial experimental results can be seen Figure 2.3, where the impact of human logic was left unchanged throughout the course of training and whose magnitude was varied over several agents. After concluding that forgetting could not be overcome by modifying the intensity alone, Hee implemented a human decay hyperparameter, with the hypothesis that human dropout could give the agent higher initial performance while preventing eventual the eventual collapse human logic appeared to induce. The results of this experiment can be seen in Figure 2.4, where many of the agents performed well at low intensity levels.

2.3 Related Work

Over the past decade there have been many significant advances in methods for training agents to play various games. Not surprisingly there are many innovative ways to engineer an environment and design an algorithm to extract meaningful features from pixel data. This section focuses on the three most relevant algorithm design methodologies to provide insight into potential implementations for a FlappyBird network: how the algorithm manages feature extraction, how the algorithm receives input, and the magnitude of human influence.

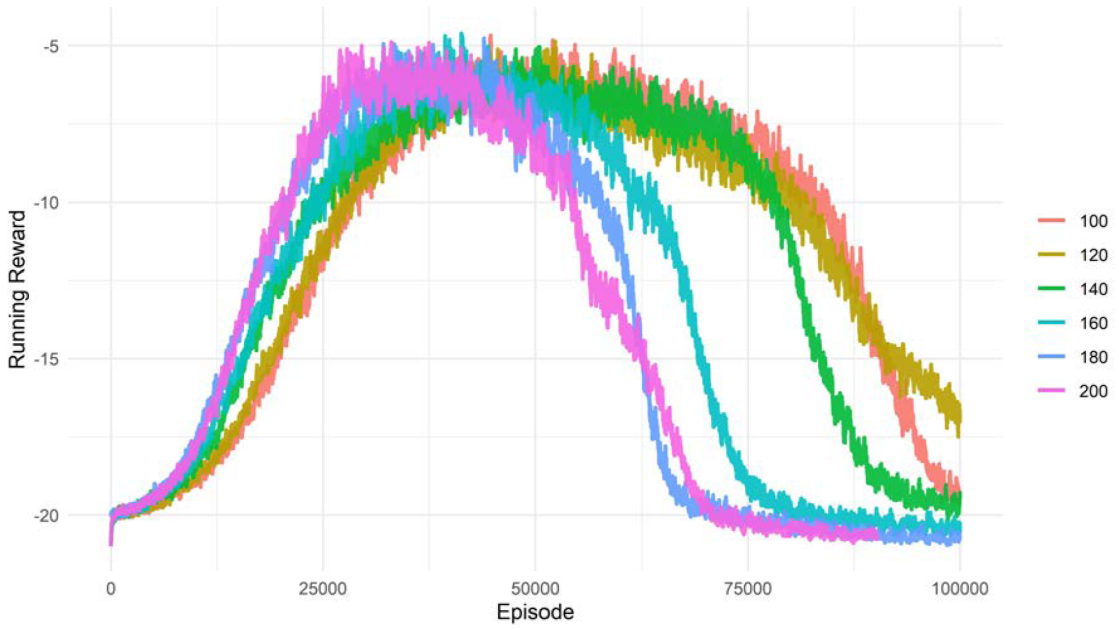


Figure 2.5: Hee's agents trained using different numbers of neurons per hidden layer all experienced performance collapse at approximately 50,000 episodes. Source: [3]

2.3.1 Feature Extraction

Traditional Q-learning algorithms required hand-crafted features or a small enough environment that an agent could exhaustively explore it in a reasonable amount of time. Many successful works have been published demonstrating the effectiveness of agents trained in such a manner, including the Training an Agent Manually through Evaluative Reinforcement (TAMER) agent [2]. Examples of these features in the context of FlappyBird would be the metrics that are returned by the environment to describe the current state (lateral distance in front of the bird to the closest pipe, the bird's vertical position and velocity, the top and bottom height of the gap the bird needs to navigate through, etc). Some environments such as Pong do not provide that utility, and additional work must be done by the programmer to search the environment space for key items such as the paddles and ball. Before being able to do any learning, Hee's algorithm first had to scan the entire frame for the location of the agents paddle as well as the ball, omitting the other players paddle as a potential feature [3]. This has worked for simple environments where there are a limited number of potential features to inference on or where there are features of obvious importance. As [13] discussed, one key drawback to custom-engineered features is that they lead to networks that are unable to generalize across multiple distinct environments. The task of identifying and codifying specialized features for each environment an agent could be expected to operate within grows far too quickly to always rely on humans to do feature engineering.

Because a computer lacks that same learning functionality, we cannot conclusively say that what we think is useful (or useless) is equally so for a computer. How can we be certain that we can identify the *most* meaningful features in a complex environment? This is what brought about the introduction of convolutional neural network (CNN)s to RL. Designed to mimic the function of the human brain, CNNs have gained significant traction in the world of deep learning for their ability to extract features from images. Mnih et al first began to utilize convolutional layers as an intermediate step between the initial stage of image pre-processing and what used to be their second stage where several fully-connected layers were stacked to provide the estimated Q-value for each potential action $\hat{Q}$ [14]. Silver et al utilized CNNs to create a 19x19 representation of a position in the Chinese board game Go when creating AlphaGO, which allowed them to significantly reduce the search space for their algorithm [1].

In trying to generalize artificially intelligent agent performance across multiple domains, Mnih et al. introduced the concept of a deep q-network (DQN), where a CNN was used to extract relevant information about the game state from an input of raw pixels from the game itself [14]. This model became powerful enough that it allowed the network to be trained to outperform professional human players across multiple distinct environments in the Atari 2600 system [13]. With the incorporation of CNNs into the DQN, training was no longer reliant on depth of exploration nor on selectively engineered features because of the agents ability to extract meaningful features from the pixels themselves. Although the filters of a convolutional layer can be custom-defined by the programmer to extract very specific features, they are also able to learn the values of the filters on their own to be able to extract the most features from their inputs. This self-learning behavior enables agents to be able to emulate human cognitive behavior to identify the key components of its environment.

2.3.2 Ordering of Training Inputs

As discussed in Section 2.2, performance collapse is a common issue the field of RL is currently contending with. The root cause of it is reasonably well-understood when training an agent to perform multiple tasks (i.e. to play multiple games) on the same synaptic weight set because the weights needed for one game will be modified when training on the second game. As seen in the case of Hee's agent, performance collapse can also happen when training on a single game in some instances. Mnih et al. postulated that the common method of training off of games being played straight through was potentially contributing to the instability of learning in the network. Up to this point, learning was being done by having the agent play and update its policy in real-time. In their 2013 paper, Mnih et al. propose a new method of training that has been widely implemented in advanced DQL systems: the replay buffer [14]. Their training methodology began by having an agent play 50 games utilizing a purely random policy to store a number of frame-experience pairs where each frame was saved along with an experience tag that conveyed if the game ended after that frame and the reward received after that frame. Once the replay buffer was filled the algorithm would randomly sample frame-experience pairs from it and perform gradient descent based on the experiences saved with the frame. By providing input to the network in a randomized order, correlation between frames was significantly decreased which also made the network less prone to over-fitting patterns and training to learn the actual strategy of the game.

In [10] and [3] training was conducted by having the agent play through an entire game before conducting gradient descent on the policy network. Training in this way required 100,000 games to sufficiently explore the entire state space, and each frame was only slightly different from the last. Karpathy's purely autonomous agents that were trained in this manner experienced no long-term decline in performance similar to Hee's human-influenced agents. Naturally this begs the question of whether a similar implementation of the input ordering could assist the human-influenced agents in learning their tasks.

Some algorithms may be run in a resource-constrained environment that is not capable of holding 50 episodes-worth of frames in main memory. Experience replay training methods require large computational and resource overhead and off-policy learning algorithms. To gain the benefits of replay while reducing the required overhead, Mnih et al. propose a new training design is proposed where multiple agents simultaneously utilize the same policy in separate environment instances (this can be thought of as having multiple agents read from the same policy instance, where changes made by one agent are seen by all) [11]. This form of on-policy training allows for the de-correlation of events while also allowing the algorithm to train on the policy it would be evaluated over (as is the nature of on-policy training). This new training method was efficient enough to be run on a multi-core Central Processing Unit (CPU) rather than being limited to a Graphics Processing Unit (GPU). This implementation gives the ability to have multiple training agents to explore different parts of the environment and to have varying exploration policies as a way of ensuring state-action diversity across the entire training population and dataset. Because the agent experiences are independent of one-another, so are their corresponding online policy updates, which are propagated to all agents. This pseudo-randomness eliminates the need for a replay buffer entirely. Gradients are accumulated over multiple time steps and used to update the policy in batches to prevent overwriting.

2.3.3 Human Influence in Training

Utilizing Pong as an educative example for RL, Karpathy created an agent that learned on policy-gradients purely through unaided exploration. Instead of choosing to have the agent train on a random policy, the agent would use its current policy through each episode and conduct gradient ascent to maximize the expected score. Agents trained in this manner never experienced catastrophic forgetting of any kind and were able to continuously improve with more training [10].

In [15] a human evaluator was tasked with assessing the correctness of each action the agent took. Griffith et al postulate that human feedback is too inconsistent to be converted directly to a scalar and used to augment the environmental reward (known as Reward Shaping). Human evaluators are prone to judgement and focus lapses, and also tend to adjust what they believe to be good as the agent gets better (which makes positive feedback harder to receive as the agent improves, even though the actions may still be good). This paper introduces the concept of *Advise* where the trainer's feedback is used to determine if the action was simply good or bad. By not converting human feedback to a scalar Griffith et al are able to achieve better game performance in Frogger and Pac-Man than Action Biasing and Control Sharing from [4] and [15].

Knox and Stone proposed a series of 8 methods to incorporate human feedback into RL utilizing a new agent which they named TAMER. Though the methods were specified for Q-learning, they have been used as a foundation for other implementations like PG. Most of these implementations utilized the human feedback as a scalar value to augment the environmental reward for a state-action pair or to build a function that estimates the human reward itself (which would be what is used to determine which action is best to take after training). In their work, Knox and Stone utilize a TAMER agent to model the human feedback estimation function $\hat{H}$, which is used to either modify the Q-network of another RL algorithm (in their paper they used $SARSA(\lambda)$) or provide additional information about the state-action pair available [4]. Their sixth proposed implementation merits special attention, as it was used as inspiration for the works of Walton and Hee in training Pong agents using policy networks [6]. In their tests, this algorithm performed better than all others throughout training, which is why it was used in Hee's research as will be discussed shortly.

Warnell et al. built off of the TAMER framework to incorporate a live human evaluator into training. This was done by having the trainer observe actions taken by the agent and provide a scalar value as feedback to the agent as a way of communicating the quality of decision made. Trainers/evaluators were given 10 minutes on a practice computer to rehearse giving feedback to the algorithm, after which they were given 15 minutes to train an agent how to play Atari Bowling. As the agent played in training, it would take the feedback it received from its trainer to conduct gradient descent on the weighted difference between the estimated reward it calculated and the adjusted value from the human. In this way the training process mirrors supervised learning, where the human trainer serves as a form of truth-label for the network to learn to estimate [5]. This method appears to have yielded positive results with 8 of 9 agents outperforming their trainer after only 15 minutes as seen in Figure 2.6. It is important to note that although the training period yielded very positive results, all agents exhibited extreme volatility in performance, which the authors did not evaluate or attempt to explain.

In their initial work creating agents to play Go, Silver et al elected to break their training pipeline into three distinct stages. The first stage was purely supervised learning on the policy network directly from expert human moves. In the second stage the algorithm shifts to a RL policy that improves upon the supervised learning performance by optimizing the final outcome to deliberately shift performance away from simply trying to maximize predictive accuracy. Finally, a value function is used to predict the of games played amongst itself [1]. Using 30 million distinct positions recorded from expert games, the algorithm was trained to predict the next move as a way of priming the network's ability to inference prior to reinforcement training whose policy network weights were initialized to the final weights of the trained supervised learning network. In this implementation, recorded human experience was used to help the network quickly understand basic decision-making and was limited to the initial stages of training.

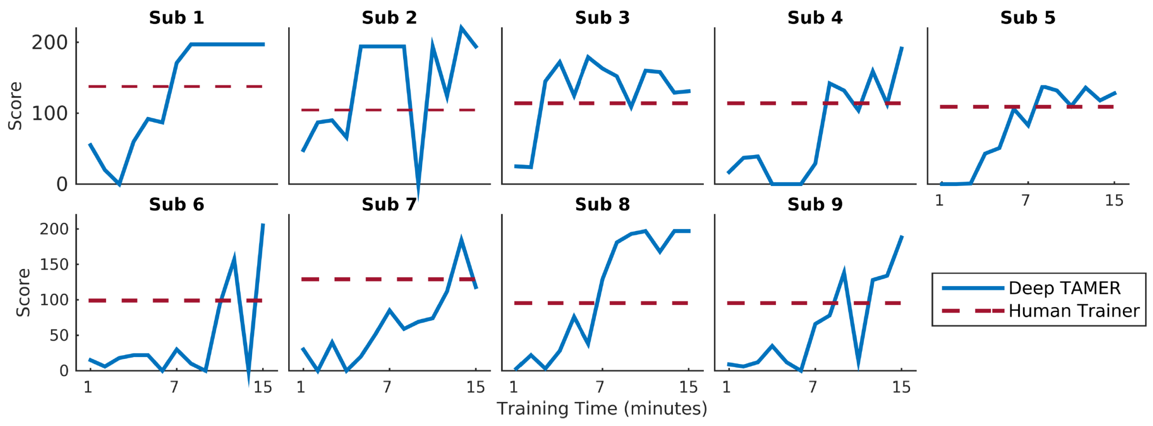


Figure 2.6: Source: [5] 9 Deep TAMER agents were trained for 15 minutes. Their performance as measured by score per game over time trained is shown against the best score of the person who trained the agent.

Evolving from *AlphaGO* to utilize Monte-Carlo Tree Search (MCTS) and no prior knowledge of game tactics the *AlphaZero* algorithm was able to learn how to play Chess, Shogi, and Go to a superhuman level of performance. In training, the agent would execute a blend of depth and breadth-first searches along trees whose roots were the various positions of the game. As with AlphaGO, the best agent was kept to play against the current policy and would be replaced by the current agent only if it lost by a 55% margin. This human-free algorithm was able to outperform the leading engines in existence [16].

In [3] the Pong agent was trained using one of the eight human-inclusion methods tested in [4] with a few changes. As outlined in algorithm 6 of the first TAMER paper, the agent would select an action with the highest combined state-action value. The degree of human influence w is specified as another hyperparameter of the model and can be any value in the range $0 \leq w \leq 1$. Hee's agent utilized a variation of this algorithm, where the action probability itself would be augmented prior as a threshold value to test against a random sample from a uniform distribution, as shown in Algorithm 1.

In his first experiments, the agent was exposed to human influence over the entire training period and showed consistent performance collapse between 20,000 and 30,000 episodes. To explore prevention methods, further experimentation was done in which the agent was initially aided by the full weight of human influence and would progressively be given increasing autonomy by exponentially decaying the human influence at a specified rate. As discussed in Section 2.2, Hee's agents performed best when the degree of human influence was in the range $0.1 \leq w \leq 0.5$ and the rate of decay was between 0.9 and 0.9999.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: Methods

This chapter presents the research methodology and the experimental design for this thesis. Section 3.1 presents the policy network structure and techniques to convert FlappyBird game frames into training input. The Section 3.2 details steps to refine the game environment and the baseline reinforcement learning code toward reproducible results per random seed. Section 3.3 describes human heuristics to be tested, including a new heuristic based on trajectory projection. Section 3.4 describes four methods to analyze collected training results to characterize agent performance from multiple perspectives. Finally, Section 3.5 outlines the plan for experiments.

3.1 Design of Policy Network

As a first effort to generalize the pathology in Hee's work, the framework of the FlappyBird environment as well as agent training closely resemble that of Hee's. In order to determine if the pathology can be generalized to human involvement in training, the first chosen step was to implement the same policy in a different environment. As such, the forward and backward propagation algorithms are exactly the same as Hee's. In this section, design aspects that are dissimilar from previous work will be explained. Key aspects for the design of experiments are the network architecture, environment engineering, input preprocessing, reward scoring, and model optimization.

3.1.1 Configuration of Layers

In his work, Hee experimented with various hidden layer sizes, but the majority was done using a hidden layer of 100 neurons because it exhibited increased performance longevity as compared to other hidden layer configurations, as seen in Figure 2.5. Due to FlappyBird's environmental complexity, the hidden layer of the network is comprised of 200 neurons using the rectified linear unit (ReLu) activation function for the neurons in the layer. For inputs of the size that environment provides, a single hidden layer of this size would likely not be large enough to sufficiently model the policy. In order to use a hidden layer of this reduced size, the original environment frame will be downsampled and preprocessed before being inputted to the network, as discussed in Section 3.1.2. We believe this gives the network enough computational flexibility to adequately model the space without overfitting.

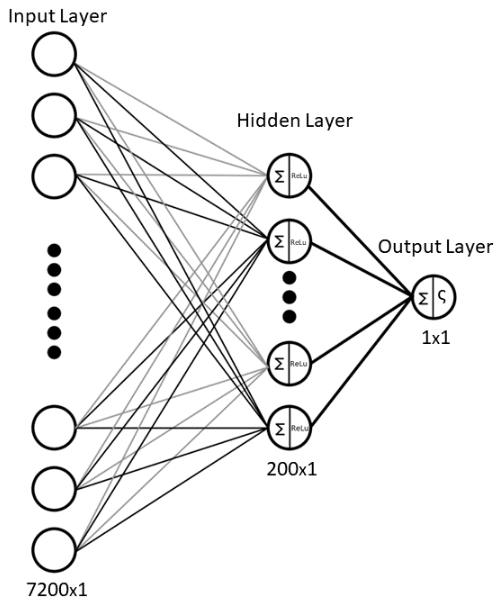


Figure 3.1: The network architecture consists of an input layer with a neuron per frame pixel, a hidden layer, and a single output layer. The hidden layer uses ReLu activation and the output layer used a sigmoid activation function.

As shown in Figure 3.1 the hidden layer is connected to a singular neuron, despite the fact that the FlappyBird game allows the agent to take three different actions: move up, do nothing (and let gravity accelerate it downward), or deliberately move down. In order to best mirror Hee's experimental setup the FlappyBird network was designed as if the agent had a binary action choice: move up, or do nothing (and let gravity accelerate it downward). This was chosen because the end-result of doing nothing and explicitly moving down are very similar, and we believed that giving the agent the ability to do nothing would be more powerful than strictly moving up or down, as doing nothing allowed the agent to "coast" along its current direction of travel, which benefited upward motion as the agent would experience the same outcome from each flap. Due to the choice of action being binary, the network was built with a single output neuron using a sigmoid activation function whose output is the probability of the agent moving up in the next step.

Network weights are initialized using the Xavier method [17], where each weight in the first layer is initially drawn from a uniform distribution with zero mean and variance of $\sqrt{\frac{1}{7200}}$ and the weights of the second layer drawn from a zero mean uniform distribution with variance of $\sqrt{\frac{1}{200}}$. Initializing in this manner ideally maintains a standard variance of activation across each layer, and helps to prevent vanishing or exploding gradients.

3.1.2 Observing the Environment

The FlappyBird environment can be configured to provide the agent observations in the form of a dictionary of key metrics describing the agent's position within the environment (vertical position, vertical velocity, lateral distance to the next obstacle, the upper and lower bound of the opening in the next obstacle, etc.) and also a *numpy* array containing the red-green-blue encoding of a game frame. The environment library provides the ability to encode the pixels of the game frame in the standard three-channel color or a grayscale frame, which is derived from the original three-channel frame. To reduce network size, Andrej Karpathy chose to utilize only one of the three color channels for the network to train and inference on, which reduced the number of parameters between the input and hidden layer by 1,280,000 for his 80x80x1 environment frame.

In this study the original input frame was 512x288 pixels in three-channel color (for a total shape of 512x288x3), and storing the first layer of parameters for this image using *numpy* 64-bit floating point numbers would require 3.54 megabytes (MB) of storage and necessitated decreasing. To do so, the image was first cropped to remove the bottom-most 112 pixels which depicted the ground and neither changed nor were pertinent to gameplay as they were out of bounds for the agent to even reach. After cropping, the red channel of the image was selected and downsampled twice by a factor of 2 (for an overall reduction in size by 4), resulting in an image of shape 100x72x1. Figure 3.4 illustrates this process from an example 16x16 input image and the resulting 4x4 output. It is important to note that we chose not to utilize any form of smoothing or blurring function prior to downsampling. Though this is certainly unconventional and resulted in loss of some data, we elected to pad the original agent image with additional border pixels prior to downsampling, and the outcome fidelity was high enough that we felt comfortable the network could still identify where the agent was in space. The original and padded agent images can be seen in Figures 3.2 and 3.3. Each episode consisted of a minimum of 8 frames (the agent dies the quickest if it continuously flaps from the start and hits the ceiling), each of which would require preprocessing before being inputted to the network. Blurring each 400x288 frame with a 3x3 Gaussian kernel and a stride of 1 would require 113,828 additional computations per frame. Aggregating this over 100,000 episodes of at least 8 frames in length, and the number would become staggeringly large. Considering each of the first 10,000 episodes lasts 1.5 seconds and more than 55 frames, the runtime is significantly shortened by using an agent image with enhanced border thickness, which adds no additional computation in runtime.

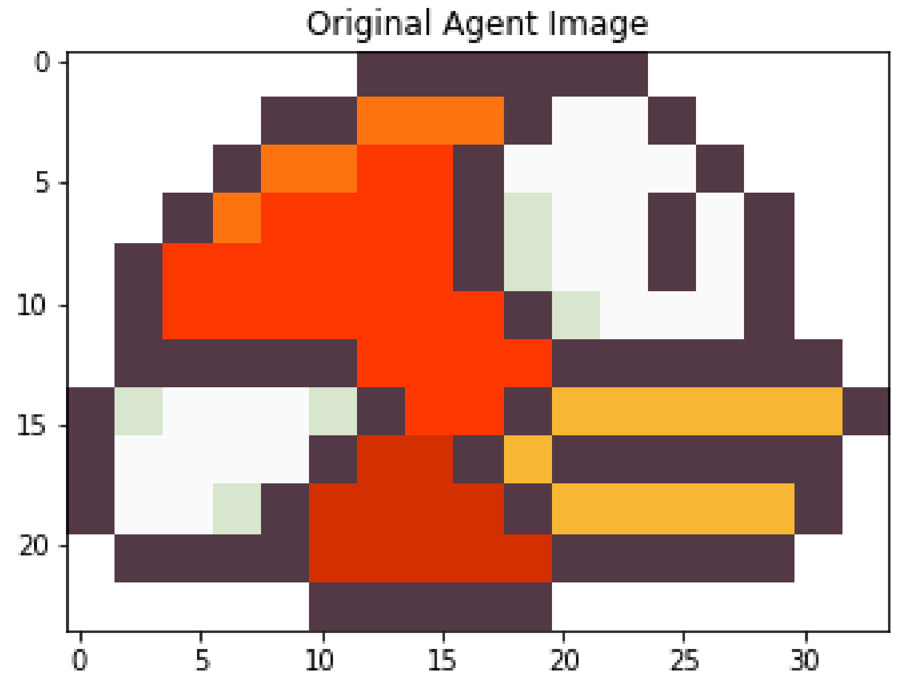


Figure 3.2: The original image used for the agent with reduced borders.

In the last steps of preprocessing, the background pixels in the frame were set to 0 to make objects stand out to the network as well as prevent the network from making inferences off of what would normally be ignored by a human player. In-situ identification of appropriate pixels to be set to zero was initially difficult, as the images were very vibrant. In order to reduce the logic required to find it, the stock background image in the environment library was modified prior to execution to set the “sky” in the frame to a dark blue color (blue channel value of 33, red and green channel value of 0) that was unused by both the agent and pipe images, which made the process of identifying the background very precise. During frame processing, pixels of this color were zeroed in all channels, and all other pixels were set to 1 in the red channel. Once this was done, the red channel was inputted to the network for inference. A visualization of the downsampled agent avatar is shown in Figure 3.5.

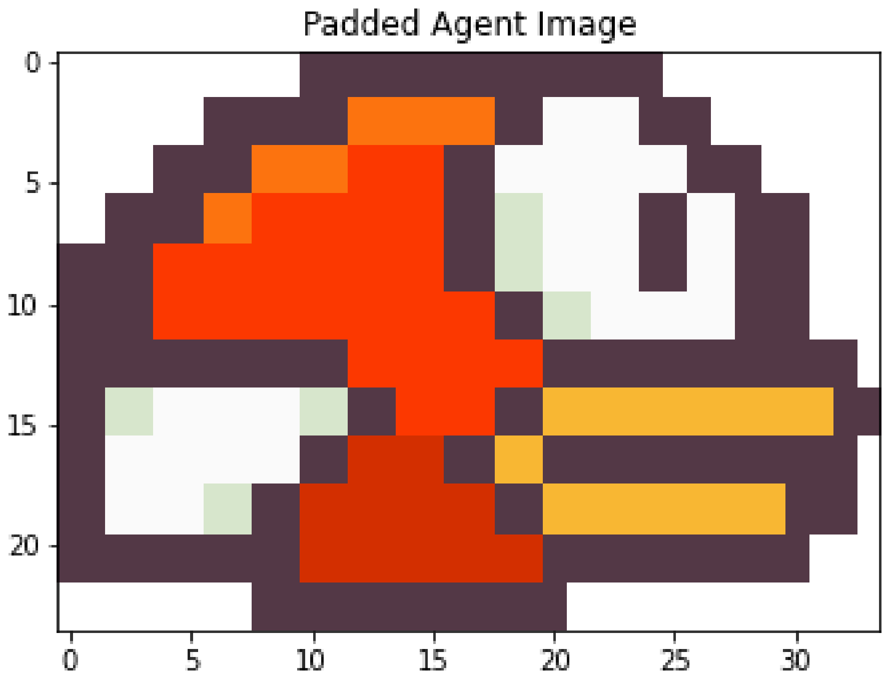


Figure 3.3: The original agent image was enhanced to have additional dark pixels around its edges to give a higher resolution product after downsampling.

As previously mentioned, the projectile motion involved in playing this game requires a sense of the agent's current trajectory. A few examples of this would be if the agent was located just above the bottom of the opening and moving downward, the correct decision to make would be to flap its wings and move up. Inferring this from a single frame as provided by the environment would not be possible with neural network because of the networks inability to recall past events. Without changing the network design to be a Recurrent Neural Network, a different method is needed to provide the network insight on the motion of the agent. This is accomplished by subtracting the pixel values of the previous frame from the current. The resulting 'difference' frame, as shown in Figure 3.6 will have pixel values of zero in locations where the pixels were constant across both frames, and non-zero values everywhere else in the frame (lower values in places objects were, and higher values where the objects are currently). In this manner the frame encodes the past and current positions of all objects without having to explicitly provide two simultaneous inputs.

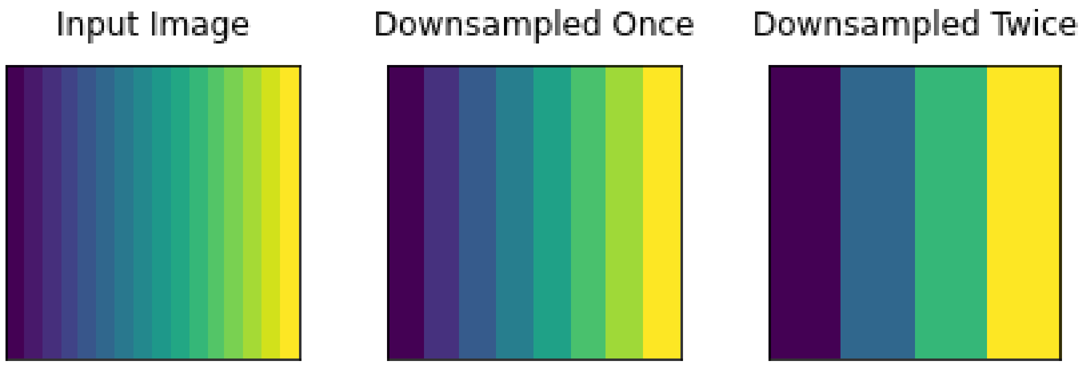


Figure 3.4: An example 16x16 image throughout downsampling. The resulting 4x4 is simply every fourth pixel from the original image.

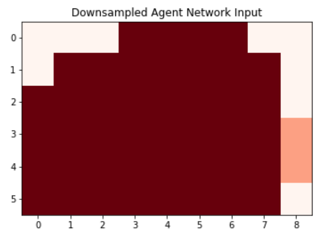


Figure 3.5: The final shape of the agent as seen by the network. In this image, the agent is made of dark colors, and the background is light.

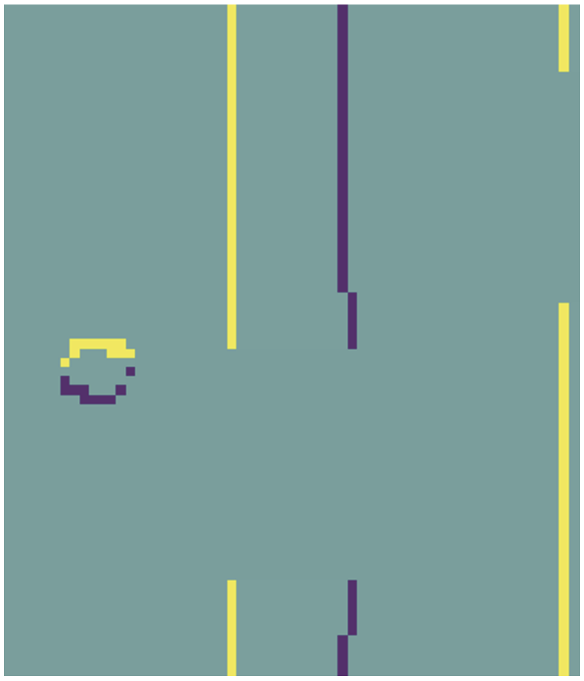
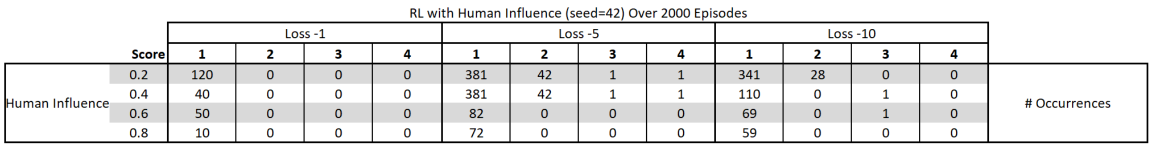


Figure 3.6: An example of the difference frame the network receives. There are only three pixel values in this frame: -1, 0, and 1. Positive values are yellow, negative values are dark blue, and zero is plotted as cyan. The direction of motion in any frame is from dark blue to the yellow. In the image, the left and right edge of the pipe are shown, as well as the left edge of the second pipe and the outline of the bird. In this image, the bird is moving upward (as indicated by a yellow top edge and dark bottom) and the pipes are moving right-to left.

3.1.3 Reward Scoring and Discounting

Before discussing the calculation of rewards, it is important to understand how training is organized for calculation. During training the agent will play thousands of games where the agent has only its one life, and when it dies the game ends. Though it has no impact on training, the abstraction of an episode is used in calculating the results over time in order to best mirror Hee's scoring methodology for this environment and will consist of 20 sequential games.

One of the main concerns when constructing this environment for RL was how infrequently the agent would receive a positive reward from successfully navigating through one pipe gap (initially set to +1), and how the negative reward it received at the conclusion of each game was five times higher (initially set to -5). Smaller experiments were done to search the hyperparameter space comparing varying degrees of human influence and negative rewards for the agent dying. As shown in Figure 3.7, the agent performed best using the original environment penalty value of -5, and a human influence of 0.2 and 0.4.



RL with Human Influence (seed=42) Over 2000 Episodes

		Loss -1				Loss -5				Loss -10				
	Score	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	
Human Influence	0.2	120	0	0	0	381	42	1	1	341	28	0	0	# Occurrences
	0.4	40	0	0	0	381	42	1	1	110	0	1	0	
	0.6	50	0	0	0	82	0	0	0	69	0	1	0	
	0.8	10	0	0	0	72	0	0	0	59	0	0	0	

Figure 3.7: Results of agent learning over 2000 episodes using varying human influence magnitudes and negative reward values.

During training, the algorithm applies a discount to the rewards returned from the environment using a *gamma* of 0.99, which is identical to Hee's procedure. In Hee's work, the agent played an environment-provided computer agent, where the winner of the episode was the first player to win 21 games. This is important because the conclusion of each game would serve as a reset point for the calculation of the discounted rewards over the entire episode, because at the conclusion of each game the agent would receive a +1 or -1 depending on whether it won or lost. This allowed the agent to observe the actions in each game in isolation, and limited the effects of actions in a game to that game alone, whereas calculating the discounted reward over the entire episode would result in miniscule incidental rewards over the hundreds of actions between all the games. This same principle is applied to the FlappyBird environment with some slight modifications. In FlappyBird, the episode consists of a single game, and so the discounted reward calculation is reset for every instance the agent receives a non-zero reward (i.e either it navigates through a gap or dies), treating each pipe as Hee did a game. An example of the discounting procedure is shown in Figure 3.8. These discounted rewards along with their corresponding move choices would be used to calculate the network policy gradient. This gradient is what is used to backpropagate parameter updates through the network. This backpropagation is what allows the network to learn a model that best fits the training environment.

3.1.4 Root Mean Squared Propagation (RMSProp)

In addition to standard practice network backpropagation, RMSProp is an optimizer that assists the network in moving to the local minimum. Similar to conventional backpropagation, RMSProp (shown in Equations 3.1 and 3.2 [18]) calculates a moving average of the square of the gradients over a batch of episodes and then normalizes the gradient by the square of the moving average [19].

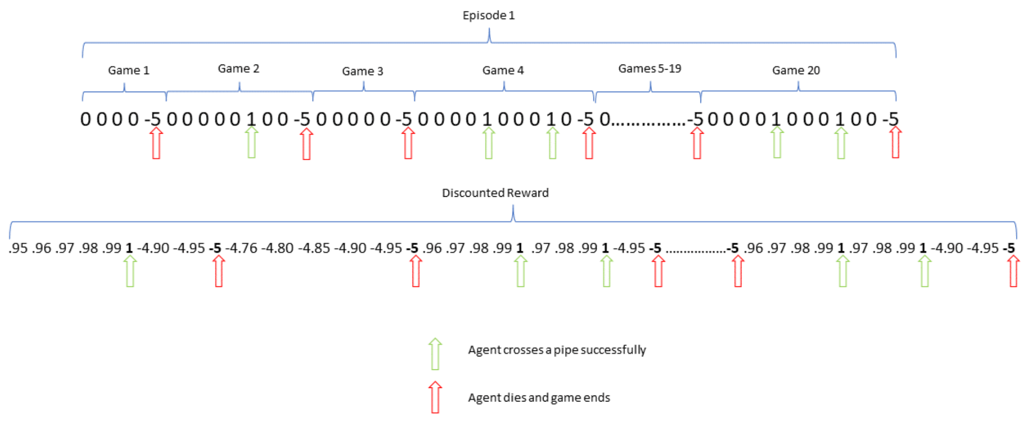


Figure 3.8: The agent is trained by playing a series of games that are independent of each other. To aggregate results and visualize similarities to Hee's research, the abstraction of an episode consisting of 20 games is used in calculating the running average and evaluate performance over time. As shown above, an episode consists of 20 sequential games. The end of each game can easily be identified by a reward of -5. Each game ending in this example episode is marked with a red arrow, and frames where the agent successfully passed through a pipe gap are marked by green arrows. To understand how the discounted reward is calculated, the discounted rewards for games 2 through 20 in this example episode are shown on the second line using the discount value discussed in Section 3.1.3.

$$E[\nabla^2]_t = \gamma E[\nabla^2]_{t-1} + (1-\gamma)\nabla_t^2 \tag{3.1}$$

$$w_{t+1} = w_t + \frac{\eta \nabla_t}{\sqrt{E[\nabla^2]_t + \epsilon}} \tag{3.2}$$

After a batch of episodes has been aggregated the optimizer updates the model weights using the calculated gradients. As the gradients begin to consistently take on the same direction (negative or positive value), the algorithm exponentially decays the running average in order to refine the updates and center the network at the (ideally global) minimum. RMSProp requires a history of updates to run properly, and the final value of the running average is written to disk in the event that training needs to be resumed later.

3.2 Environmental Reproducibility

In the original FlappyBird module, a single instance of the environment was maintained over the course of training. The environment consisted of two main modules: *PyGame Learning Environment (PLE)* (which contained the FlappyBird game class) and *pygame* (which served as the parent class that PLE and its subclasses inherited from). In order to play the game, a PLE instance as well as a FlappyBird object had to be instantiated, and at the conclusion of an episode the PLE object could be called to reset the environment. Unfortunately, this reset method simply moved the agents position back to the start, zeroed its score, and generated a new set of pipes back to the starting locations. These pipes were generated from a shared pseudo-random number generator (prng) instance inside the FlappyBird class that was maintained throughout training due to performance considerations of going through the calling procedures of re-initializing a new FlappyBird object with each episode. Because the FlappyBird class maintained state across episodes, the prng object it contained would likely provide a completely different set of pipes for each episode (different in this context refers to the vertical position of the gap in each pipe). In order to expedite training time, a separate prng object strictly for pipe data generation was added to the FlappyBird class along with a new public method that deleted the existing set of pipes, reset the pipe-specific prng to its original state, and had the FlappyBird object re-initialize the set of pipes. Each prng in the environment accepts a seed in the form of an integer to allow for consistent outputs sequences. To make the setup as straightforward as possible, a single seed can be specified when executing the python script, and this seed is successively passed to each object such that all prngs in the experiment utilize the same seed.

Equally important to the consistent generation of pipes is the agents decision-making, as the agent must be able to make the same decision every time given an identical set of network parameters, input values, and decision number (how many prior decisions it has made). This is achieved by maintaining another prng for the experiment script, which provides samples from a uniform distribution against which the probability outputted from the network is compared, as well as the initial weights for the network parameters. Consistency in decision making is achieved in that these samples will be identical so long as the seed is maintained across experiments, as the seed is what guarantees identical initial network weight values and follow-on decision-making samples.

3.3 Human Heuristic Used

As part of Hee's work, a piece of human logic was used to slightly modify the probability of the agent moving up or down, based on the contents of the frame. The intent of the heuristic was to utilize a simple piece of logic that required minimal calculation, because a human player does not actively calculate outcomes when they play. For Pong, the logic is almost obvious: if the agent's vertical position was below the ball, move the agent up, and vice versa. The intent of this section is to outline a FlappyBird heuristic that mimics the simplicity of Hee's, but in the FlappyBird environment simple yet useful heuristics are more difficult to produce due to the implementation of gravity. In Pong there is no gravitational force that affects the trajectory of the ball in-flight, resulting in a very linear (and easily predictable) trajectory. In FlappyBird gravity is constantly pulling the agent toward the bottom of the frame, and also introduces projectile motion into the experiment. The logic used in this experiment was to have the agent flap its wings if it vertical position was below the gap in the pipes, otherwise it should simply let gravity pull it down for a time step.

3.3.1 Probability Augmentation

The final step in the human-assisted learning process is to translate the recommended action from the heuristic into an action by the agent. This is done by augmenting the policy network output of the probability that the agent should flap its wings. If the recommended action is to flap the network output is increased, and if the recommendation is to do nothing the network output is decreased. One of the hyperparameters of this experiment was the magnitude of human influence, which is used to calculate the amount the network output is increased or decreased, depending on the recommendation. This human influence magnitude is limited between 0 and 1, and effectively denotes the percentage the original network output is increased or decreased by, as shown in Algorithm 1. Hee experimented with decaying the human influence over time as a way to make the network more resistant to forgetting. In his later experiments, the human influence was exponentially decayed each episode by a factor of 0.99. Sections 3.3.2-3.3.4 discuss how the human recommended move is generated.

Algorithm 1 Hee's Human Heuristic Implementation [3]

```
Initialize $w$ to human influence
$p_{up} \leftarrow$ output value from policy network
**if** human action = down **then**
    $p_{up'} = p_{up} \times (1 - w))$
**else if** human action = up **then**
    $p_{up'} = p_{up} \times (1 + w)$
**else**
    $p_{up'} = p_{up}$
**end if**
**return** $p_{up'}$
```

3.3.2 Original Heuristic

The original heuristic used for FlappyBird attempted simplicity: if the agent is vertically aligned or below the bottom edge of the gap, it should flap in order to better align itself with the middle of the gap. Otherwise it should do nothing and allow gravity to pull it down. Initial proof of concept trials run for 100,000 episodes yielded results where the agent did not show meaningful and lasting amounts of learning, with agents at the end of the training period performing nearly identically to the untrained agent. There was no rise in agent performance when plotted over time, which indicated that the heuristic potentially needed amplifying. The agent experienced issues in regions of the environment where it was close to the pipes and just at the decision threshold of the heuristic. In this region, the agent is in a position where not flapping would lower it to potentially collide with the top edge of the bottom pipe. But the heuristic would recommend the agent do nothing and allow gravity to pull it down.

3.3.3 FlappyBird Physics

In order to understand the logic of the human heuristic, the basic physics principles of the environment must be explained. At startup, the agent is initialized to a (x,y) coordinate of (57,256) and a vertical velocity of 0. Though in the implementation the agent never moves left or right in the screen (the pipes moves from right to left across the screen to give the impression of agent movement), we give the agent an abstracted horizontal velocity, which is a constant value of 4 pixels per step and equals the rate of pipe movement in the opposite direction. Gravity accelerates the agent downward throughout the game at 1 unit velocity per step, and when the agent flaps it receives an immediate acceleration of -9 (which is upward because the positive y, and velocity axes point downward). At any time step *t* the agents position and velocity are calculated as shown in Algorithm 2. The intricacies of this environment stem from how flapping and downward speed are handled, where a limit is placed on consecutive flaps as well as the maximum downward velocity. For any two sequential actions there are four permutations of action combinations for which Algorithm 2 calculates positional data for. There are three non-intuitive cases are when the agent either: does nothing for some time and then flaps, flaps twice in a row, or flaps and then does nothing. The first potential event sequence of interest is when the agent does nothing for a sequence of steps (accruing some positive velocity toward the ground) and then flaps its wings (beginning on line 9 of Algorithm 2). In the second case (beginning on line 13 of Algorithm 2), the agent would gain 8 pixels of height from the first flap while achieving a velocity of -8, and then neither sink nor have velocity as result of the second flap, finishing the second step with a velocity of -7. The last case (beginning on line 17 of Algorithm 2) is of concern because the amount of impacted state change due to one flap is variable, with the outcome being fixed (a velocity of 8 toward the ceiling). After studying the environment infrastructure explained above, the design of a more complex heuristic was feasible.

3.3.4 Trajectory Projection Heuristic

The difficulties introduced in FlappyBird arise from the agents position relative to the obstacles. An easy example of this would be if the agent was 2 time steps (8 units away in the x axis) moving downward with a velocity of 3, and a vertical position 7 units above the bottom edge of the gap. Because the agent is above the bottom edge, the original heuristic would recommend it do nothing and ultimately reduces the likelihood it flaps its wings (the method by which is discussed in 3.3.1), which for illustrations sake we will assume is what happens. Following Algorithm 2 for 1 step, the agent is now 1 time step (4 units in the x axis) from the pipe, and 4 units above the bottom edge of the gap with a downward velocity of 4. Once again the heuristic recommends the agent do nothing, as it still is aligned above the bottom of the gap, and once again we assume it indeed does nothing (based on the random policy). After the second step, the agent is 0 units from the pipe in the x axis and is

-1 units above the bottom edge (in other words, it is hitting the left edge of the pipe 1 unit below the bottom of the gap) and dies. In this example the heuristic did not achieve what it is intended to do, which is to keep the agent toward the middle of the gap. After revisiting the issue, we decided that a better heuristic would be to provide recommendations to the agent based on its projected trajectory through space.

The revised heuristic takes the current game state into account, including the velocity of the agent and position of the agent with respect to the nearest obstacle and calculates the trajectory of the agent if it were to do nothing until it has moved beyond the pipe gap. Using the game physics from Algorithm 2, the position of the agent at each future time step is calculated until it reaches the closest edge of the pipe. At this step, the projected position is checked to see if the agent is above or below the gap (hitting the top or bottom pipe), and if the position is still above the ground. No checks are done to see if the agent hits the ceiling because the entire calculation is done assuming it never flaps. If none of the previous checks return hits, the agent is known to be entering the gap and the heuristic repeats the top and bottom pipe checks after each future step until the trajectory is calculated beyond the farthest edge of the pipe. Due to the no-flapping assumption for the calculation, once in the gap only the bottom pipe is checked for contact with the agent. If the agent passes beyond the far edge of the pipe without contacting the pipe, the heuristic returns a do nothing command.

3.4 Characterization of Network Policy

This study will leverage several techniques to visualize the progress of the network and justify the networks' actions. These methods help to better understand why the network learns the way that it does, what it is learning to focus on observing, and whether the network is learning at all. The visualization methods to be used will be discussed in this section.

3.4.1 Moving Average Reward Curves

Due to the volatility of agent performance as it undergoes training, plotting the end-of-game scores yields little information other than a distribution of scores and basic score trends. A better way to visualize agent progress is to calculate a moving average of the agent score. Moving averages are commonly used in the investment industry because they weight current events more heavily than those of the past, which make it a useful tool to forecast the behavior of the data. To further smooth noise from the dataset, the episode abstraction was created to aggregate the scores from 20 consecutive games into a single total score, which was used to calculate the moving average.

3.4.2 Perturbation-Based Saliency Maps

Though easy to compute, reward curves only provide insight into two aspects: whether the agent is improving, and when the agent reaches a certain state (i.e. when it suddenly changes from improving to worsening). The average reward curve is used to locate critical episodes in the experiments where additional information is needed on the state of the network and what the agent is learning to observe. As introduced in [20], the relative importance of each pixel in the input image can be ascertained by modifying it to observe how the network output is affected. In [20], the input logits of the softmax layer were used in place of the network output because they produced a "sharper" image (this is because the logits were not normalized, unlike the output of softmax). The network in these experiments has no normalization layer after the hidden layer, so the network output will be used instead of pre-softmax logits. To calculate the importance of one pixel, the downsampled (72x100) input image is given to the network and produces a network output p. To determine the relative importance of some pixel (i, j), the image is zero-padded to preserve output size and a *OpenCV* Gaussian Blur is applied to that pixel using a kernel size of 3x3 and stride of 1. After blurring the pixel, the modified image is inputted to the network which produces a new probability $p'_{i,j}$. The pixel importance is then scored using the scoring function $S(i, j) = \frac{1}{2}||p - p'_{i,j}||^2$ and written into a 72x100 matrix. Once all pixel scores have been calculated, the score matrix is plotted using a heatmap.

3.4.3 Feature Extraction Maps

A common technique in understanding the performance of CNNs is to try to visualize the features being learned by a layer in the network. As a way of observing the features being learned by the network, the same approach as defined in Section 3.4.2 was used with a modified scoring function where the value of the pixel to be analyzed is logically negated and then converted to an integer instead of applying a Gaussian Blur. In the Python language, any variable or value that is not 0, None, or False is logically evaluated to be True, and in Python True is represented by the integer 1. This novel method of perturbing the frame pixels meant that no pixels in the frame would be subject to the values of their neighbors, meaning that a 5x5 grid of zeros would no longer evaluate to 0 before being inputted to the scoring function. Once the individual pixel values were modified, the scoring function and plotting methods from Section 3.4.2 were used.

3.4.4 Neuron Activations

A common technique when using CNNs is to plot activation values of a specific layer in order to learn about which features were being extracted from the inputted image [21] [22]. Due to differences in architectures, it is not possible to plot the features being learned by the neural network, but a similar technique may be applied to the weights themselves. To better understand how the network is behaving during training, activation maps will be generated to visualize the layer outputs leaving the ReLu activation function and determine if there are any exploding or vanishing nodes. Hee introduced plotting the network weights using a box plot to do this.

3.5 Training Configuration

3.5.1 Experiment 1 - Reproduce Pathology

Human influence was set to 0.4 with no decay. This was to see if the pathology would occur, with plans to increase the magnitude of human influence if the pathology were not reproduced in the first round. The human-aided agent was to be trained for 200,000 episodes alongside a non-aided agent to serve as a performance benchmark. The environment penalty for dying was set to -5, and the reward for navigating one pipe successfully was +1. Backpropagation was conducted at the conclusion of every episode with RMSProp conducting fine-tuned gradient descent every 200 episodes. The rate of learning for this experiment will be set to 0.0001 and the discount factor for reward discounting set to 0.99.

3.5.2 Experiment 2 - Generalize Pathology to Range of Hyperparameter Values

If Hee's pathology is reproduced in experiment 1, the second experiment will be run with increased magnitudes of human influence up to 0.9 to mirror Hee's work. The intent of this experiment is to conclusively determine that the performance collapse pathology from experiment 1 accurately represents the behavior of the policy network when using human heuristics and not simply an outlying phenomena.

3.5.3 Mathematical Analysis

The results of experiments 1 and 2 will be used as a baseline for analysis of the network behaviors with the intent of explaining why the FlappyBird network performed the way it did. Once the FlappyBird environment can be sufficiently explained, the Pong environment and Hee's results will be further analyzed to determine what contributed to the performance collapse pathology he experienced.

3.5.4 Evaluation Methodology

The simplest metric to quantify agent progress is the reward curve as introduced by Mnih et al. and is calculated as $r_t = 0.99 \cdot r_{t-1} + 0.01 \cdot r_{sum}$ where r_{sum} is the agent score for an episode [13]. In an episode where the agent successfully navigates 4 sets of pipes before dying, the r_{sum} is -1. Experiment 1This running reward provides a visual for agent performance over time and weights future rewards more heavily than those further in the past. The running reward is calculated per episode and is stored for later analysis.

Algorithm 2 Algorithm used by FlappyBird to calculate new position and velocity data

```
$y_0 = 0$
$v_0 = 0$
$t = 0$
$gravity = 1$
$lastAction = None$
while game not over do
    $t = t + 1$
    get action decision
    if $action = flap$ and $lastAction = None$ then
        $v_t = -9$
        $y_t = y_{t-1} + v_t + gravity$
        $lastAction = flap$
    else if $action = flap$ and $lastAction = flap$ then
        $v_t = 0$
        $y_t = y_{t-1}$
        $lastAction = None$
    else if $action = None$ and $lastAction = flap$ then
        $v_t = v_{t-1}$
        $y_t = y_{t-1} + v_t$
        $lastAction = None$
    else
        if $v_{t-1} = 9$ then
            $v_t = 10$
            $y_t = y_{t-1} + v_t$
        else
            $v_t = v_{t-1} + gravity$
            $y_t = y_{t-1} + v_t$
        end if
    end if
end while
```

Algorithm 3 Trajectory Projection Heuristic

```
get agent and pipe states
calculate agent y position when x is aligned with left edge of pipe
if $agent.y$ ≤ top of gap or $agent.y$ ≥ bottom of gap then
    return flap
else
    while $agent.x$ is not beyond the right edge do
        if $agent.y$ ≥ bottom of gap then
            return flap
        else
            calculate new $agent.y$, $agent.x$, $agent.vel$
        end if
    end while
end if
return none
```

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

A central theme for this chapter is the performance of the unaided agent. The unaided agent exhibited performance collapse after moderate learning initially. Consequently, the scope of our work has deviated from that described in Chapter 3. The new experimental focus is on pinpointing the root cause of the observed pathology that is not influenced by a human heuristic.

The chapter is organized as follows. Section 4.1.1 presents the initial pathology with the unaided agent and results from similar experiments with human heuristics. The second section reports a comprehensive effort to resolve the pathology with the unaided agent, including adjustments to input encoding, network activation, weight initialization, training optimization and hyperparameters. The results from this effort confirm the persistence of the pathology. Section 4.2 contains a detailed analysis of the training results at neuron, feature, and gradient levels. Finally, Section 4.3 summarizes key observations and findings from this chapter.

4.1 Observed Pathology

Multiple rounds of experimentation were completed as our understanding of the problem shifted. Each experiment brought with it a new set of questions that required answering, and in this section the results of each experiment will be discussed along with the concerns that arose from them. After initial experimentation was conducted, the environment was adjusted for reduced complexity, followed by experimentation with different difference frames, activation functions, and weight initialization methods. The hyperparameters used across all experiments are shown in Table 4.1.

4.1.1 Initial Experiments

Original Heuristic

The resulting performance of the both the baseline and test agents was poor, as no agent was ever able to successfully navigate more than four pipes. In this set of results the baseline outperformed the test agent across the entire episode, which led us to believe that our heuristic was either flawed or too simple to provide use to the agent. In offline tests done to visualize what was happening during the test, we also discovered that the environment was producing random pipe locations for each episode, meaning that it was fairly unlikely any two episodes shared the same set of pipes.

Table 4.1: Common experimental hyperparameters

γ	0.99
Learning Rate	0.0001
Decay Rate	0.99
Batch Size	200
Gap Reward	+1
Loss Reward	-5
Human Influence	0.4

Interestingly, the agents universally peaked relatively early in training characterized by a steep ascent in performance for the first 800 to 900 episodes, after which all experienced a sharp decline in performance very similar to that of Hee's agents. This performance collapse was initially expected based on Hee's results, but only in the test agents. In this experiment, the baseline agents all experienced performance collapse and in some cases, actually fell further than the test agents, which contradicted what we had hypothesized would happen. The running reward for these agents is shown in Table 4.2.

From all of these observations we decided to redesign the environment to produce a consistent set of pipes each episode in order to make the game easier to learn for the agent. To address the potential for an improved heuristic, a new one was implemented that was inspired by the concepts of projectile motion in physics, where the agents position and velocity as well as the location of the next pipe gap were used to calculate the trajectory of the agent assuming it never flapped again. If the projected trajectory of the agent either collided with the bottom pipe or hit the ground before crossed through the gap, the heuristic would recommend the agent flap to move upward. In all other conditions the heuristic recommended that the agent simply allow gravity to pull it downward, because the applicable trajectories either collided with the upper pipes and needed to be adjusted downward, or the agent was projected to be successful assuming no flaps. Implementing this heuristic was challenging due to the physics of the environment, which did not behave as predictably in all cases of the agent flapping its wings. To better understand the underlying cause of the poor initial experiment results, the next round of experiments utilized pipe sets that were consistent across episodes as well as an improved heuristic.

Table 4.2: Original experimental results

Agent	**Episode Number**							
	0	**2500**	**5000**	**7500**	**10000**	**12500**	**15000**	**175000**
H1	0.00000	0.13358	0.10337	0.08702	0.06525	0.04088	0.02866	0.01324
H2	0.00000	0.15303	0.11491	0.10875	0.08182	0.06570	0.03482	0.01584
H3	0.00000	0.15071	0.10597	0.05780	0.03287	0.02126	0.01077	0.00719
H4	0.00000	0.13098	0.09715	0.08219	0.07176	0.05627	0.04150	0.03357
H5	0.00000	0.15768	0.13050	0.10576	0.09918	0.07805	0.05515	0.05093
B1	0.00000	0.24693	0.19686	0.18689	0.16645	0.14388	0.14019	0.12440
B2	0.00000	0.18030	0.14678	0.10209	0.09454	0.05843	0.01787	0.03473
B3	0.00000	0.17279	0.07003	0.00699	0.00007	0.00044	0.00033	0.00708
B4	0.00000	0.12849	0.08010	0.07084	0.05756	0.04508	0.04824	0.04775
B5	0.00000	0.19165	0.12345	0.03316	0.03649	0.03723	0.02644	0.02346

The running reward results for all five original agents every 2,500 episodes. These should be compared against the baseline agents B1-B5 in this table. The reward for games beyond this was not shown as it monotonically decreases.

Projection Heuristic

Comparatively speaking, the results of this experiment were worse than its predecessor. Our hypothesis was that the improved heuristic would significantly increase the human-aided agent's performance, but this was not the case as it appeared that the new heuristic did more to hinder agent performance than help it. On closer examination, we believe that the way that the environment enforces flapping made the heuristics calculations inaccurate. In an effort to simulate the effects of time and the reality that a flap of a wing is not instantaneous, the agents flap implementation does not treat all flapping actions equally. This is what makes projection difficult because as we shall discuss it generates four potential two-action possibilities: flap-flap, flap-no flap, no flap-no flap, no flap-flap (cases 1, 2, 3, and 4 respectively). In cases 2, 3, and 4 the trajectory projection of the second action is simple to calculate. Case 1 on the other hand is not as trivial, because after the first flap in the sequence, the agent will gain an upward velocity of 9, and if it flaps again for the second action, the agents upward velocity becomes 0 and it does not move vertically for that time step.

Consider the scenario where after flapping its wings, the agents projected trajectory is still too low. Human intuition tells us to recommend flapping again right now, but that would put the agent into case 1, resulting in an upward velocity of 0 while moving 4 pixels closer to the pipes and making it harder to gain a favorable position. The correct action to recommend for the agent is do nothing (the case 2 sequence) in order to coast upward 9 pixels and prime it for the next recommendation which would allow it to flap again (so the flap, no-flap, flap sequence could be viewed as a flap followed by case 2 followed by case 4). This would give the trajectory the adjustment it needs, but was not trivial to understand. Because this environment does not treat every two-action sequence the same, forecasting the trajectory of the agent was difficult and likely led the agent to learn a sub-optimal policy too quickly. Another curious result is the similarly poor performance of the baseline agents as shown in Table 4.3. If the baseline agents performed poorly without the use of a human heuristic, we can logically conclude the heuristic is not the original cause of agent performance regression.

Environment Adjustment

After these two rounds of experimentation it was clear that something more powerful than the human heuristic was influencing the network behavior because none of the baseline agents used it and were still collapsing. Our first hypothesis was that the environment was too complex for the agent to learn and needed to be made less so. To do this, the environment source code was modified in two key ways that will be kept for the remainder of future experimentation: the size of the gap in pipes was increased by 50% and the sequence of pipes was made to be the same across games.

In the original environment the pipe gap was 100 pixels in size, and increasing it by 50% still kept the task from being too easy to randomly achieve while requiring less precision on the part of the network actions. The main environmental engineering stemmed from managing how pipes were propagated in the frames. In the original implementation each training period used a single instance of a NumPy prng that generated pipe metadata on an as-needed basis to support game play. As a result the location of the gap in the first pipe

Table 4.3: Results with alternate heuristic

Agent	**0**	**2500**	**5000**	**7500**	**10000**
H1	0.000	0.0454	0.004	3.0e-3	2.94e-3
H2	0.000	0.029	0.033	0.003	2.0e-3
H3	0.000	0.0466	.006	3.31e-4	6.04e-5
H4	0.000	0.0432	0.010	1.34e-4	4.47e-4
H5	0.000	.0344	.002	8.96e-4	2.76e-4
B1	0.000	0.138	0.112	0.078	0.050
B2	0.000	0.108	0.078	0.050	0.036
B3	0.000	0.125	0.049	0.018	0.007
B4	0.000	0.126	0.102	0.066	0.026
B5	0.000	0.122	0.074	0.041	0.017

The running reward results for all five trajectory projection agents every 2,500 episodes. These should be compared against the baseline agents B1-B5 in this table. The reward for games beyond this was not shown as it monotonically decreases.

significantly varied between games. To address this, a "fresh" instance of the same prng was instantiated at the beginning of each game to ensure the set of pipes experienced across any two games from an experiment were identical.

Once these two features were implemented, a final hyperparameter search was done to see if there was a better reward function to use in training. The default penalty the agent receives on dying is -5, and a positive reward of +1 on crossing through a gap. In this test, the value of the positive reward was varied from +1 to +10, and we discovered that the default value pair of -5/+1 yielded the best results.

As stated earlier, we began to turn our attention to the root cause of this behavior in all our agents, and in the following sections our results will only show the baseline agents (though the behaviors they exhibit were also observed in the human-aided ones) in an effort to show how the pathology responded to various changes. It should be noted that although the baseline agents outperformed the human-aided ones before the changes made to make the environment less complex seemed to make the human agents more effective, as shall be shown in the following results.

4.1.2 Alternative Difference Frame

At this stage in experimentation, we were confident that the original heuristic was best-suited for training the agent but still lacked a reasonable explanation for the poor performance of all agents. Having verified the forward pass and back propagation algorithms were both mathematically sound, we developed a hypothesis that the network was not being provided enough information and thus was being starved of meaningful information. As described in Section 3.1.2, a difference frame was used as input to the network by subtracting the previous frame from the current.

As discussed, the 50 pixel-wide pipes in the frame move at a constant 4 pixels per step from right to left. Recalling that the game frame is pre-processed to have pixel values of either 0 or 1, the majority of pixels associated with pipes would become zero (a pipe that is 50 pixels wide and moves 4 pixels to the left would have only the edge pixels potentially change values, resulting in 8 pixels in the space having non-zero value). A smaller example of this is shown in Figure 4.1 with a 10x10 frame that moves 4 pixels right to left as in the FlappyBird environment. This could potentially cause the network to perform worse, as it appeared that the majority of frame information was being lost in pre-processing. To estimate the correlation of the difference to agent performance, three additional tests were run where the difference frame was calculated with fraction of the previous frame. Instead of the original method of $current - previous$ three others were tested: $current - .25 * previous$, $current - .5 * previous$, and $current - .99 * previous$. These values were chosen to do a breadth-first search across the range of potential frame discount values while guaranteeing their resulting difference products would have non-zero values for pixels that did not change state between frames.

Our hypothesis was that the agents being passed a difference frame that where only a fraction of the previous state was subtracted would outperform those using the difference frame used in Hee and Karpathy's work, but the opposite was true. This experiment revealed that the agent performs better when the difference frame has the full pixel values from the previous state subtracted from the current, such that all pixels in the frame are either -1, 0, or 1. In this experiment, agents using a 25% difference frame (100% current - 25% of the previous frame) performed the worst with an average score of 1.4 points per game and a median of 1. Second worst were those using a 50% difference frame, whose average score was 2.1 points per game and had a median score of 2. The best performing agents were those who used a 99% difference frame with an average score of 4.4 points per game and a median score of 4. The data very conclusively shows that the original data encoding method was more than sufficient to convey the current and past game state to the network. The running reward curve for the 0.99 difference frames is shown in Table 4.4 to show the rise and fall of performance. The running reward curves for the 0.25 and 0.5 difference frames would normally be shown, but those agents performed so poorly that the running reward was less than a millionth after roughly 25 episodes of training. Their tables would be effectively zeroed out and for the sake of cleanliness they were omitted.

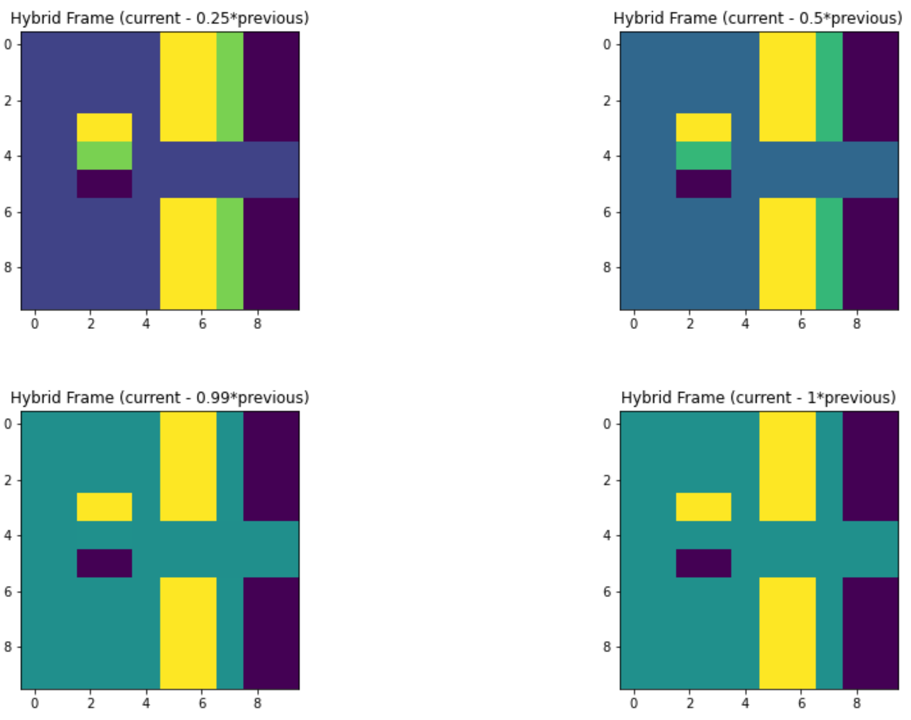


Figure 4.1: To minimize data loss in the input, several experiments were conducted providing the network with difference frames that biased toward current state. The difference shown on the top left was calculated by subtracting a quarter of the previous frame from the current. All examples shown in this figure are from a notional 10x10 frame. Yellow indicates a value of +1, dark blue indicates a value of -1, and green and cyan indicate the pixel value 0. Notice how if the agent in the example (the square on the left of the frame) does not move positions, then it will disappear for differences subtracting the full previous state values.

Table 4.4: Results of alternate difference frame experiment

Agent	0	2500	5000	7500	10000	12500	15000
D.99-1	0.000	0.072	0.046	0.033	0.024	0.022	0.029
D.99-2	0.000	0.068	0.074	0.072	0.078	0.068	0.054
D.99-3	0.000	0.089	0.049	0.057	0.044	0.049	0.044
D99-4	0.000	0.084	0.048	0.009	0.006	0.003	0.004
D.99-5	0.000	0.077	0.046	0.022	0.020	0.031	0.024

The running reward results for all five agents using a partial difference frame every 2,500 episodes. These agents calculated the difference frame by subtracting 99% of the previous game state from the current.

4.1.3 Alternative Activation Functions

After the tests conducted in 4.1.2 yielded similar results to previous experiments, we began to believe there was something even more subtle that was influencing the outcomes. We had learned that the network performed best with frames encoded to -1, 0, and 1, but felt that this caused most of the neuron activations to be 0. In doing reading for a class, we had the idea that our activation function may have been causing the network to miss opportunities to refine its policy through backpropagation.

ReLu was originally chosen as the activation function because of its non-linearity property (which in turn makes the neural network capable of modeling complex non-linear environments such as FlappyBird) and its computational simplicity, but a potential shortcoming of ReLu is that it is possible for neurons to “die” during training. As the name indicates, a critical component to the use of gradient ascent is the existence of a gradient, of which there is none for a neuron when its activation is 0. This happens because any non-positive input to the ReLu produces an activation of 0, which produces no gradient and is irreversible by gradient ascent [8]. If the weights for a neuron consistently produce non-positive activations, that neuron will cease to be updated by gradient ascent and will soon have a 0 activation value from then on.

With the overwhelming majority of pixels in the difference frame being either 0 or -1, we realized that the majority of neuron activations could be trending to zero and thus could potentially explain the networks permanent decline in performance for both the experimental and baseline agents. After further research we decided to implement Leaky ReLu as the hidden layer activation function because it ensures that the only time the neuron outputs zero is if the sum of inputs is zero: this in turn would increase the percentage of frames that the neuron could be updated by backpropagation and (in theory) build better performance.

The results of this test showed that using a more robust ReLu variant had minimal effect on the agents performance, which contradicted our hypothesis. Agents using pure ReLu achieved an average score of 4.3 points per game whereas those using Leaky ReLu achieved 2.8 points per game. Though the highest score achieved was the same for both activation functions, Leaky ReLu consistently performed worse across its ten agents, and also suffered from the same performance collapse that the ReLu agents were experiencing, as shown in Tables 4.2 and 4.5. Agents using pure ReLu exhibited better average performance in this test and had the highest game score achieved, though by a margin of 1 point. The results of this test did not indicate that the activation function being used was the source of the network behavior, and as with the original heuristic tests the agents all began to collapse relatively early with the majority reaching peak performance before episode 1000 and a single agent peaking at 2200.

Table 4.5: Running reward results using Leaky ReLu

Agent	**0**	**2500**	**5000**	**7500**	**10000**	**12500**	**15000**	**175000**
Leaky1	0.00000	0.04775	0.05644	0.05630	0.06220	0.05497	0.04732	0.01988
Leaky2	0.00000	0.09521	0.03138	0.03812	0.04046	0.04750	0.04545	0.03814
Leaky3	0.00000	0.00849	0.00149	0.00073	0.00092	0.00034	0.00045	0.00045
Leaky4	0.00000	0.25093	0.24561	0.26718	0.26750	0.26134	0.27183	0.22050
Leaky5	0.00000	0.06099	0.03774	0.01666	0.00707	0.00963	0.00533	0.00628

The running reward results for all five agents using Leaky ReLu every 2,500 episodes (which translates to 50,000 games). These should be compared against agents B1-B5 in Table 4.2.

4.1.4 Alternative Weight Initialization

While conducting the experiment discussed in Section 4.1.3, we encountered literature that discussed better weight initialization methods for use with neural networks experiencing dying neurons. The literature we read claimed that Xavier weight initialization was prone to experiencing dying neurons due to the fact that it produced random outputs that were too symmetrical (equal numbers of positive and negative values very close in magnitude) and that by using a less-symmetrical initialization function would produce a more robust network [23]. Additionally, He et al. assert that Xavier initialization for networks with ReLu is flawed because in the Glorot and Bengio proposal for Xavier initialization, it was assumed that the activations are linear and it is well known that ReLu is not [24] [17].

Though early analysis did not indicate that a significant number of neurons in the network were dying, we wanted to conduct one last experiment to see if the network would perform better using the proposed Kaiming He implementation. The central idea of this weight initialization is to prevent exploding or vanishing gradients due to the extreme large or small values of the weights themselves, and this initialization method aims to manipulate the variance of the weights in a manner that compliments the activation function being used in the network, which is ReLu [24]. Under this implementation, the weights are sampled from a 0-mean Normal distribution whose variance is calculated as $\sqrt{\frac{2}{n}}$. In this equation n represents the number of neurons in the hidden layer, which in this experiment is still 200. Several tests were completed to determine if the use of the He initialization could improve the performance of the network and our hypothesis for these tests was that the weight initialization method would delay if not prevent the collapse in performance from happening.

The weight initialization method had some impact on network performance but is likely not the root cause of the behaviors being observed as agents initialized with He's method performed somewhat equivocally to those initialized using Xavier's method. Table 4.6 shows the running reward for all five different seeds. In this experiment, Xavier agents scored an average of 4.2 points per game with a median score of 4, whereas the He agents achieved an average score of 4.8 and a median score of 5. These metrics show a noticeable improvement in the He agents, but their reward curves and game scores remain close to the Xavier-equivalent agents'. Agents using He's initialization experienced a performance peak at an average episode of 255 whereas the Xavier agents experienced the same peak at episode 327 on average. This means that though the He agents achieved higher single game scores, the Xavier agents crossed more pipes in total across all games in all seeds.

Table 4.6: Running reward using He-initialization

Agent	0	2500	5000	7500	10000	12500	15000
He1	0.000	0.072	0.046	0.033	0.024	0.022	0.029
He2	0.000	0.068	0.074	0.072	0.078	0.072	0.068
He3	0.000	0.089	0.049	0.057	0.044	0.049	0.044
He4	0.000	0.084	0.048	0.009	0.006	0.003	0.004
He5	0.000	0.077	0.046	0.022	0.020	0.031	0.025

The running reward results for all five He-initialized agents every 2,500 episodes (which translates to 50,000 games). These should be compared against the original agent results in Table 4.2.

4.2 Analysis

The intent of this research was to generalize the results of Hee's research, where human logic initially increased improvement in agent performance but eventually resulted in the agent regressing back to its original untrained state. As shown in Section 4.1 the agent using human logic performed as expected, but the baseline agent using pure RL unexpectedly exhibited the same pathological performance. In this section, both the baseline and experimental agents will be analyzed to better understand the cause of the pathology in all cases. In this section various analytical methods will be used to compare relative performance among the different experiments and agents of varying configurations, and for the sake of brevity Table 4.7 lists them all out. References to experimental agents made through the remainder of the chapter will be made with respect to this table (i.e. a reference to agent A2 corresponds to entry A2 in the table). Agents A2-A6 do not use human heuristics because the focus of experimentation shifted to understanding the cause of unanimous performance collapse.

As an initial analytical step, the experimental code was verified for correctness. From examining the running reward curves as well as the best game scores of each experiment, it is clear that the network is initially learning and that what it is learning appears to be good. Over all configurations, the average best performing human-aided agents successfully navigated through 3.51 pipes per experiment, and the average best baseline agent score was 3.23 pipes per experiment. Put another way the best human-aided agents from each experiment had an average score of 3.51, and 3.23 for all baseline agents. The "best agent" from an experiment is determined by the agent whose game score was the highest (the same is true of the "worst agent" but with the lowest score). During training, the model is automatically saved whenever an agent achieves the highest score of the experiment to that point (the model state is saved at all local maxima). The worst agent can be selected among thousands of agents that achieve the lowest score possible in a game, and for ease of selection the most recent low-scoring agent is selected for analysis.

Leading in to the data analysis, our hypotheses were that one of the following was contributing or causing the network to collapse: neurons in the hidden-layer were experiencing extreme activation values (either high or low), the network parameters were learning to focus on the incorrect region of the frame, the gradient was decreasing too quickly, or the network was overfitting to the environment (memorizing a sequence of actions). With these hypotheses in mind, we structured our analyses to cover three stages at varying granularities which we termed neuron-level, feature, and policy overfitting analyses. Each of the sections will analyze network performance through their own lens using a common frame from the game to provide continuity between tools. Before applying the tools, the reader may benefit from seeing the raw frame which is shown in Figure 4.2.

4.2.1 Neuron-Level Analysis

Activation CDFs

In analyzing the behavior of the hidden layer, the first step was to visualize the activation value of each neuron to observe any trends. For this task, a cumulative distribution function (CDF) plotting mechanism developed by Dr. Xie was applied to pickle files containing the 200 hidden neuron activation values for the best and worst agent in each experiment. These graphs plot the 25th, 50th, and 75th percentile activation values for one frame across all hidden neurons, over the course of one game in training and from these CDFs we can infer the range of the remaining activation values not plotted.

To understand how the model may have developed with training, the same frames were used to generate the CDF for the model immediately after initialization as shown in Figure 4.3. As expected, Figure 4.3 shows a very small level of variance in activation values due to the random initialization of weights centered on 0. Given the initialization method and lack of training it can reasonably be expected that there be approximately equal numbers of positive and negative weights in the network (from the initialization we also know that the values should also be similar), and without training these should approximately cancel each other out in early games. Given this beginning state of the network the analysis of the best and worst model states show that the activation values themselves grow with training.

A noticeable trend in both the human and baseline agents was the variance of activation values between best and worst agents: an example of this variance is shown in Figure 4.4. This figure informs us that in the best performing state all activation values are reasonably bounded but there are still a large number of neurons that are producing zero output and can lead to the neurons dying as previously mentioned in section 4.1.4.

From these plots we see that the neuron activations steadily grow in magnitude over time, though the cause of that growth is unclear. At first glance the reader may question why there are no negative activation values, but recall that these networks used traditional ReLu where all non-positive activations are set to 0. To provide additional insight into the network behavior using this other activation function the Leaky ReLu variation of the same agent from Figure 4.4 is shown in Figure 4.5. As expected, it is clear that many activations are non-positive, which also explains why the preponderance of activations previously shown were 0 as ReLu will set those non-positive values to 0.

Weight Visualization Matrices

From reviewing the spread of activation values the next component that merited analysis was how the weights themselves were configured within the network. We believed that it would be very revealing to see if a plot of the weights by pixel were clustered at all and if they were, where were they clustering? In his original blog post, Andrej Karpathy included a figure in which he plotted the individual weights for some of the hidden neurons in his network to show what in each frame the neurons were trained to respond to [10]. Plotted in black and white, these graphics can be used to better understand how a neuron is trained to respond given a stimulus in one region of the frame. A tool developed by Dr. Xie was used to provide the same plots for our network, and an example weight visualization matrix is shown in Figure 4.6.

As expected, the random initialization process yields a weight matrix that appears to be purely noise with no obvious weight clusterings. The weights of the best performing agents from Figure 4.3 are shown in Figure 4.8. Given that the network is responding to the location of the pipes and agent it stands to reason that with more training time the formation of positive and negative clusters would become more prominent, oriented in parallel with the pipes, which is what we see in Figures 4.7 and 4.8. From these images we see that the weights are aligned by cardinality in parallel with the pipes, with pronounced horizontal lines of mostly solid black or white forming in each neurons plot.

4.2.2 Feature Analysis

Saliency Maps

Once the hidden layer had been visualized the next step was to attempt to identify how the exploding or diminishing activations translated to network actions in the game. In order to do this, saliency maps as described in Section 3.4.2 based on methods from Hee and Greydanus et al. were generated. The objective of these plots is to assign a measure of importance to each pixel in a given frame, which we will use to potentially triangulate the principle cause of the regressing network performance in all cases. The best and worst performing agent models for each seed from each experiment were used to generate these maps using a common set of frames. Because the test frames were controlled, differences across any two maps could only be attributed to differences in the models themselves.

Pictured in Figure 4.9 are example saliency maps for each experiment conducted. It is clear even without frame overlays that the network is learning to focus on the correct pixels within each frame, as it is clear in each map where the agent as well as pipes are. From these maps we can also ascertain a level of confidence the network has in the location of the pipes are based on the apparent blurring (similar to a halo effect) around each pipe edge in the maps: networks that are more certain will have a less fuzzy halo surrounding the pipes. In some cases such as with the hybrid frames, the maps produced by the network are seemingly incoherent, with no apparent meaning or values to be extracted from the large homogenous blocks of color, though with some context it is still possible to guess where the edges of the pipes are in the map. As we will see with the feature maps, agents that performed worse overall appeared to more heavily weight pixels that were further from it in the future, as opposed to those immediately near it. Two examples of this are the hybrid frame maps shown on the bottom row of Figure 4.9.

To illustrate the differences between agents who performed well and those who did not, Figure 4.11 shows the best and worst models for agent A0. In the worst agent's map, a larger halo can be observed around the edge of the pipe and a large variation in values can be observed along the length of each pipe edge, which could mean that the network is forgetting to look for straight geometric shapes, or that the policy being learned has become noisy. In contrast the best model for that agent assigned an equal importance to all pixels along each pipe edge shown, and the objects each appear more crisp in the image. Another example of a high performing model's saliency map is shown in Figure 4.10.

Feature Maps

Unfortunately the application of a Gaussian Blur to most pixels in the frames yields an uninteresting result of 0, as the majority of pixels are 0 and are also surrounded by 0 pixels. Taking inspiration from feature visualization methods common to CNNs, the a new scoring function was applied to the same frames used in generating the saliency maps in order to visualize how the network weights viewed each pixel in the frame as discussed in Section 3.4.3. Figure 4.12 shows examples of what we call feature maps for each of the experiments completed in this study.

As expected, the clusters of high importance pixels are concentrated in areas where pipe pixels have been previously found (near the top and bottom of the frame), and there are considerable areas of low importance near the middle of the frame height, where the gap in the pipes is often found. What may be indicative of a suboptimal policy being learned is the location of the furthest forward high-importance pixels. Agents A5's map in Figure 4.12 shows well-defined pipe-shaped clusters of similarly colored pixels, but the pixels of highest importance are actually behind where the agent can ever reach. This was certainly not expected, as our natural intuition led us to believe that those pixels should be assigned no importance at all (because they are in the agents past). Was the agent learning to focus on mostly the right pixels initially and over time allowing itself to be influenced by even more?

This is curious, but we believe that looking beyond the best-performing agent is even more revealing. As shown in Figure 4.13 the feature map of the agent with the worst performance is fairly dull, with all pixels having a nearly negligible individual impact on the action decision of the network (judging by the scale being in the one-millionth range). In addition, the only pixels of distinguishing value are found near where the agent is expected to be, but no pixels beyond that are found to be important. This could potentially explain why the agents are performing so poorly, but not how they came to be so. We were beginning to wonder why the agent would have such a diverse feature map in the beginning, and then over time have it smooth out until almost no pixels seemed to matter.

4.2.3 Policy Overfitting Analysis

One potential explanation was that the game environment was simply too complicated for this specific network architecture. It could be that there was nothing *wrong* with the architecture, but that it simply needed more layers in order to effectively learn how to play. Another potential explanation was that the network was simply overfitting to a suboptimal policy, and because of the length of training was potentially memorizing a poor sequence of moves.

One way to see if the environment was too complicated would be to simply build a new network using a more complex architecture and try training again, but that would not effectively explain why it was the case. As a final approach to conclusively determine if the environment was too complicated for the network, we decided to investigate the possibility of overfitting by examining the amount of exploration the agent was doing.

Exploration Quantification

In this phase we analyzed the action sequences taken by the agents over the course of training, treating the sequence of each game as a binary character string (1 is a flap up, 0 is do nothing). Our hypothesis was that if the agents were indeed overfitting to the environment, we would see a small number of unique strings compared to the number of games played in that experiment. Additionally, we hoped to see a drastic change in exploratory behavior that coincided with the peak in performance and beginning of decline. In this analysis, the shortest action sequence length was found and all action strings for the entire experiment were truncated to that length before being tracked for number and times of occurrences in that experiment. In this way, we were able to create a CDF showing the percentage of all explored action strings over the course of the experiment as a way of visualizing how the exploration/exploitation ratio that the agents exhibited.

It is difficult to definitively say what the ideal plot should be because there is no golden ratio of exploration to exploitation in RL, but we can say that a bad curve will show a high degree of initial exploration followed by a long period of slow growth as if approaching an asymptote. Figure 4.14 shows examples of these curves across the range of experiments conducted in this study.

Several interesting observations can be made from these curves. The first is that curves belonging to experiments that yielded better high scores tend are closer to a straight line as is shown in the top left image of Figure 4.14 and curves that appear to approach an asymptote at $y = 1$ tended to belong to experiments that yielded lower scores. Most notable in the aforementioned figure are the curves belonging to agents A0 and A6, which had a unique sequence of actions 74.51% and 31.96% of the time, respectively. This is by far the largest amount of exploration of any of the agents, as the rest tended to hover around 10% exploration over their experiments.

In many of the worst performing agents, the exploration curves reach 90% exploration before even completing 15% of their training (by game number 25,000). This does seem to indicate that the network is choosing to focus on a policy, as the graphs indicate it is spending nearly 85% of its training time in an incredibly narrow subset of the total action space.

To further illustrate the differences across the agents and implementation methods, the best and worst of each experiment are compared side-by-side in Figures 4.15 and 4.16. Beginning with the 25% difference frames on top of Figure 4.15 the best agent explored new action sequences 2.33% of the time, having completed 90% of its exploration within 10% of its training time. The worst agent spent 2.31% of training exploring, and completed 90% of its exploration within the same amount of time. Therefore it is not surprising that these agents performed the worst of any experimental configuration. On the bottom of the same figure we see the curves for the 50% difference encoding, where once again the difference between the best and worst agents is very small. The best agent only explored 2.97% of the time, and the worst explored only 2.80% of the time. Both of these agents also completed 90% of their exploration within approximately the first 10% of training time. Once again it is somewhat logical to see how these were a close second-worst to the 25% difference agents.

On examining Figure 4.16 the curves of agents A5 and A6 shift downward and to the right, a clear indication that exploration is drawn out over a longer period of training time. In the A5 agents, between 40-50% of training time is used to explore new action sequences, and it takes between 16-25% of training time to reach 90% of their terminal exploration. The best of these agents achieved the second-highest single game score and consistently produced running reward curves with the second-highest peak values before collapsing. On the bottom row the A6 agents show the best overall exploration curves, using nearly the entire training period to explore new action sequences in the environment and reaching 90% exploration in 66-75% of their training time. These agents produced the best running reward curves as well as single-game high scores of all the experimental agents.

We can use these curves to crudely predict which architectural modifications can have the largest impact on both score and exploration (as these two are reasonably linked). From this analysis, it appears that the weight initialization method and activation functions used in the hidden layer have the largest impact on network performance and can greatly improve results. Still this does not explain the peak in performance followed by a total regression, and though it appears that the network is doing what it should in the high-performing agents, something is still causing it to regress.

Gradient Analysis

At this point in the analysis we have studied the behavior of the hidden layer and how its weights interact with the input frame, how the output is affected by slight modifications in input, and how the network as a whole attempts to understand the environment it is in; and the only conclusion we can draw is that whatever is causing the network to do poorly in the long term must be external to the individual components of the architecture (and possibly even their sum). After this we are fairly confident in concluding that the environment is too complex for this network architecture to learn, but as a final measure we analyzed the behavior of the gradient over the course of training.

On the recommendation of Professor McClure at Naval Postgraduate School (NPS), the gradient for each layer was treated as a vector whose magnitude could be plotted over time. If the network was learning properly the gradient should be relatively large in early games and decrease with time as it approaches 0. Our hypothesis was that the gradient was actually doing the opposite, and was exploding over time (which could explain why the agent would never improve beyond a certain point). Several experiments were re-run using agents A7 through A10. In all instances the gradients were shown to be exploding in a nearly monotonic manner as shown in Figures 4.17 and 4.18. These plots indicated that the network was attempting to move in the direction of a more optimal policy, but was being prevented from doing so by RMSprop and in turn was simply producing larger gradients to combat the smaller steps allowed by the optimizer.

Our original idea was to try to correlate the agents peak performance with the corresponding point on the gradient plots to see if some trend could be identified, but in all cases there was no distinguishing feature in the magnitude plots to indicate a drastic or sudden change in network state. To be thorough, the point on the gradient curves that corresponds to the performance peak of the agent is marked by a thin red circle in each of the plots. From these plots we hypothesized that if we could better control the gradient in training we may be able to prevent the agent from collapsing, or at the very least understand why the collapse was happening. As a final step in studying the network (and once again at the recommendation of Professor McClure) L2 Normalization was applied to the model as a way of forcing the network weights to be as close to 0 as possible, which we believed would cause the gradients to be smaller in value. For this test we ran a series of A7 agents varying the L2 constant to be a power of 10 between 0.1 and 0.00001 in order to find the ideal hyperparameter value for L2 Normalization.

Our hypothesis that L2 Normalization would correct or improve the network behavior was incorrect, as it seemed that L2 Normalization decreased the size of the gradient vector, but did not prevent it from exploding still. Figure 4.19 shows the gradient plot over time using an L2 constant of 0.1, which was the most effective of the set of tested values. From analyzing the gradient plots before and after RMSprop is applied it appeared that the network was attempting to learn a better policy, but was becoming frozen by the decayed learning rate as a result of the optimizer. As a final effort four differently-seeded agents of types A1, and A5-A10 were run for 100,000 episodes without using RMSprop with the goal being to understand how a non-decaying learning rate would affect overall performance. We believed that we would see the corresponding gradient vectors initially spike and then gradually decrease over time as the network was able to make constant updates to its parameters toward an optimal policy. But our hypothesis was incorrect, and the results showed that removing RMSprop from the network actually increased the size of the gradient vector. When the optimizer was being used the magnitude of the gradient first gradient vector ranged from 3,000 to 20,000 but without RMSprop the same magnitude ranged from $1x10^{30}$ to $1x10^{90}$. In addition to massively exploding gradients was the increased overall performance of the non-optimized agents with respect to running reward and single game high scores. The non-optimized agents exhibited a high degree of volatility with many exaggerated local minima and maxima in running reward plots but did not experience permanent performance collapse in all agents unlike those agents using RMSprop. Given the constant growth of the gradient vector in all agents (optimized or not), we believe that the environment is too complex for a network of this architecture. Our objective was to understand a previously observed behavior, not to successfully train an agent to play FlappyBird and because we were observing the pathology of interest in all cases we felt that the next step would have to be re-designing the network architecture to include more hidden layers.

4.3 Summary

After the first round of experiments showed that the baseline agents performed better than those using the human heuristics. This led to the customization of the FlappyBird environment to make the pipes consistent across games with gaps that were 50% larger. Using this new environment configuration it was found that the original method for calculating the difference frames yielded the best performance as compared to agents who more heavily weighted pixels from the current state. We did find that the choice of more robust activation function and weight initialization methods had a positive impact on the network performance, but did not solve the problem of performance collapse.

From analyzing the individual neuron activations it is clear that something is causing the neurons to fire at higher and higher values the longer they train causing an evident explosion. Plots of the individual pixel weights for the hidden neurons provided little additional information as to the cause of the collapse, but did show that the worse-performing agents exhibited large homogenous bands of weight values. This clustering could indicate that the network is becoming fixated on a few regions instead of looking at the frame as a whole.

In studying the effect the individual pixels had on network decisions, there was an evident trend that stronger agents more heavily weighted the pixels closest to the agent. Stronger agents also had obvious regions of non-importance that clearly corresponded to parts of the frame where gaps in the pipe were often found (if even just partially). The worst agents appeared to treat the pixels in input frames almost equally, and certainly displayed less confidence in the location of the pipes or agent when present in the frame.

After applying many commonly-known tools to the network with no success, the possibility of the environment being too complex for a network with this topology grew. The list of potential causes of the pathology at this stage were effectively limited to the model overfitting to the environment, and the model underfitting to the environment (i.e. the environment was too complex). If the agent were to be overfit to the environment it would be expected that it would consistently achieve a certain score later in training, which did not happen. Though lower, individual game scores achieved by the agents were fairly volatile even through the end of training.

To be even more certain overfitting was not happening, the action sequences of the agents were analyzed. This analysis showed that the worst agents tended to complete all their exploration early in training and would become fixed in a small subset of the overall action space. In addition to this is the fact that even the agents that explored nearly continually through training still experienced the same performance collapse, though at a slightly later point. We hypothesize that an association can be made between performance and degree of exploration, though the cause of the collapse is still unknown.

Finally, the gradients themselves were analyzed to observe overall trends that may have been occurring. This analysis showed that the magnitude of the gradient vector was exploding as training continued, but the effect of which was being masked through the use of RMSprop. As a last test to see if a controlled gradient could alleviate the performance collapse, L2 Normalization was used on multiple agent configurations. In each of these agents the normalization only managed to decrease the size of the gradient vector, but did not prevent the same explosion over time.

Table 4.7: Table of agents tested in experimentation

Agent Tag	Heuristic Used	% Difference	Activation Func	Init Method	L2 Norm
A0	Original	100	ReLu	Xavier	False
A1	Projection	100	ReLu	Xavier	False
A2	None	99	ReLu	Xavier	False
A3	None	50	ReLu	Xavier	False
A4	None	25	ReLu	Xavier	False
A5	None	100	Leaky ReLu	Xavier	False
A6	None	100	ReLu	He	False
A7	None	100	ReLu	Xavier	True
A8	None	100	Leaky ReLu	Xavier	True
A9	None	100	ReLu	He	True
A10	None	100	Leaky ReLu	He	True

The six different experimental configurations tested in this thesis. As an example of how to read the configurations, agent A4 did not use a human heuristic, calculated a difference frame by subtracting 25% of the previous from the current frame, used ReLu activation, and had its weights initialized using Xavier initialization and no normalization techniques applied.

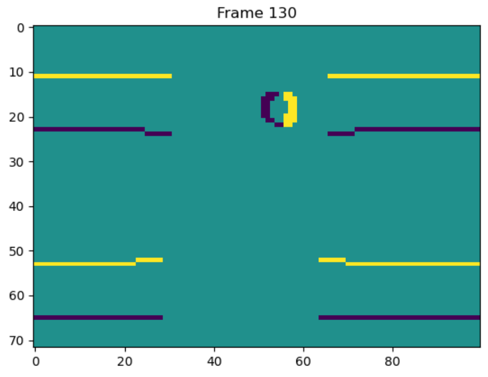


Figure 4.2: The frame to be used in the following sections for analysis and comparison.

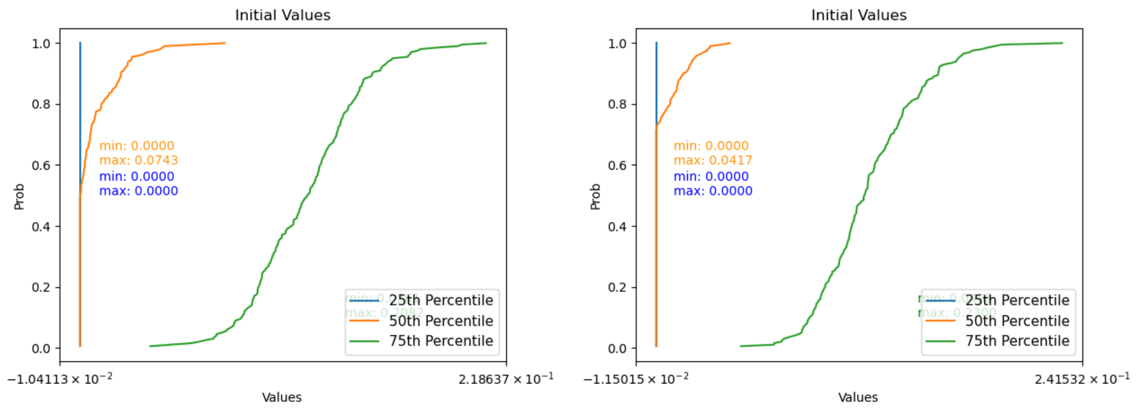


Figure 4.3: The CDFs generated by the network activations for the first game immediately after network initialization for two A6 agents using different seeds. Activation values trend toward 0 across most neurons, which may be caused in part by the frame containing mostly pixel values of 0, as well as the initialization method creating weights centered on a 0 mean.

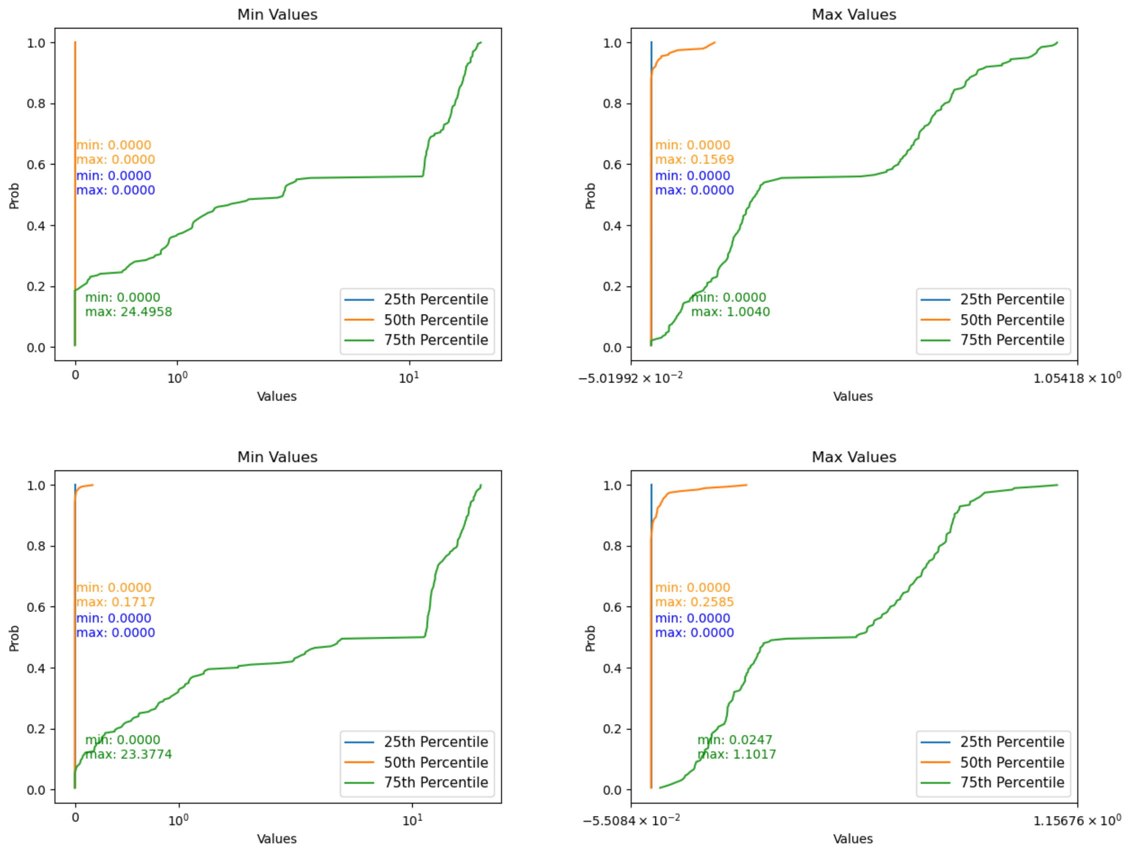


Figure 4.4: On the top-left are the activation values for the worst performing model state for agent A6 (see Table 4.7) with a seed of 5. On the top-right is the best performing model state for the same experiment configuration. On the bottom-left are the activation values for the worst model state for agent A6 with a seed of 1, whose best model state is depicted on the bottom-right. Notice how in the worst performing states at least half of the neuron activation values were 0, and the maximum activation value is nearly 24 times larger than the maximum activation value of the best agents in both cases. The minimum and maximum values of each percentile are denoted next to their corresponding lines in the same color.

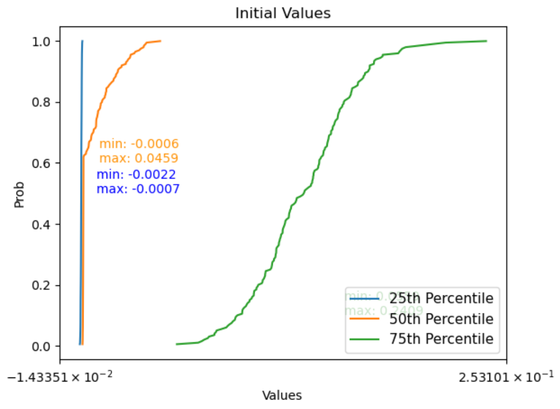


Figure 4.5: Shown on the left are the initial activation values for agent A5. Shown on the right are the activations for the worst performing state of that model in training.

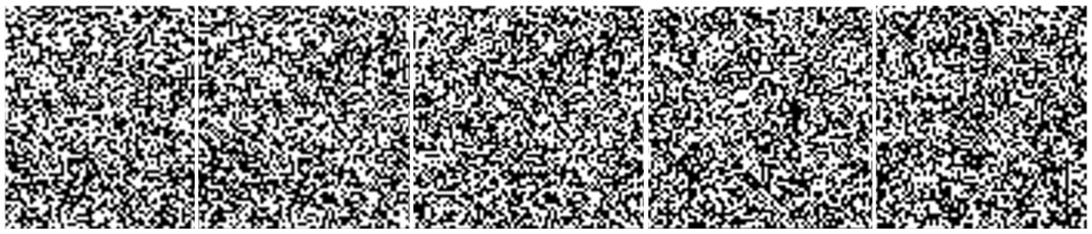


Figure 4.6: The neuron weights were reshaped to match the input frame shape and then black-white encoded based on cardinality.

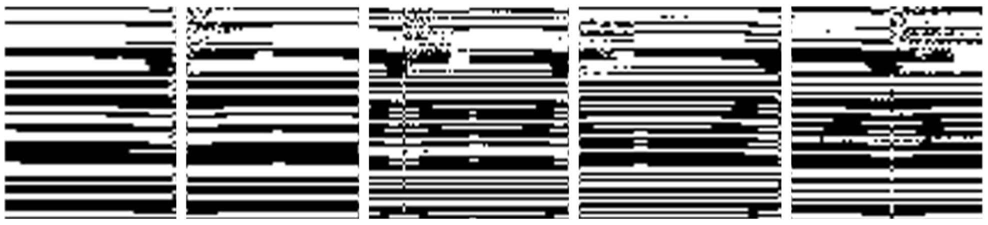


Figure 4.7: The weight visualization matrix of the best overall agent according to running reward and single-game score, which belonged to the A4 agent group.

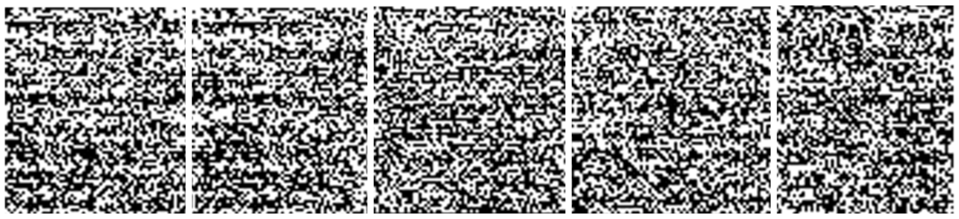


Figure 4.8: The weight visualization matrix of the best overall agent according to running reward and single-game score, which belonged to the A6 agent group.

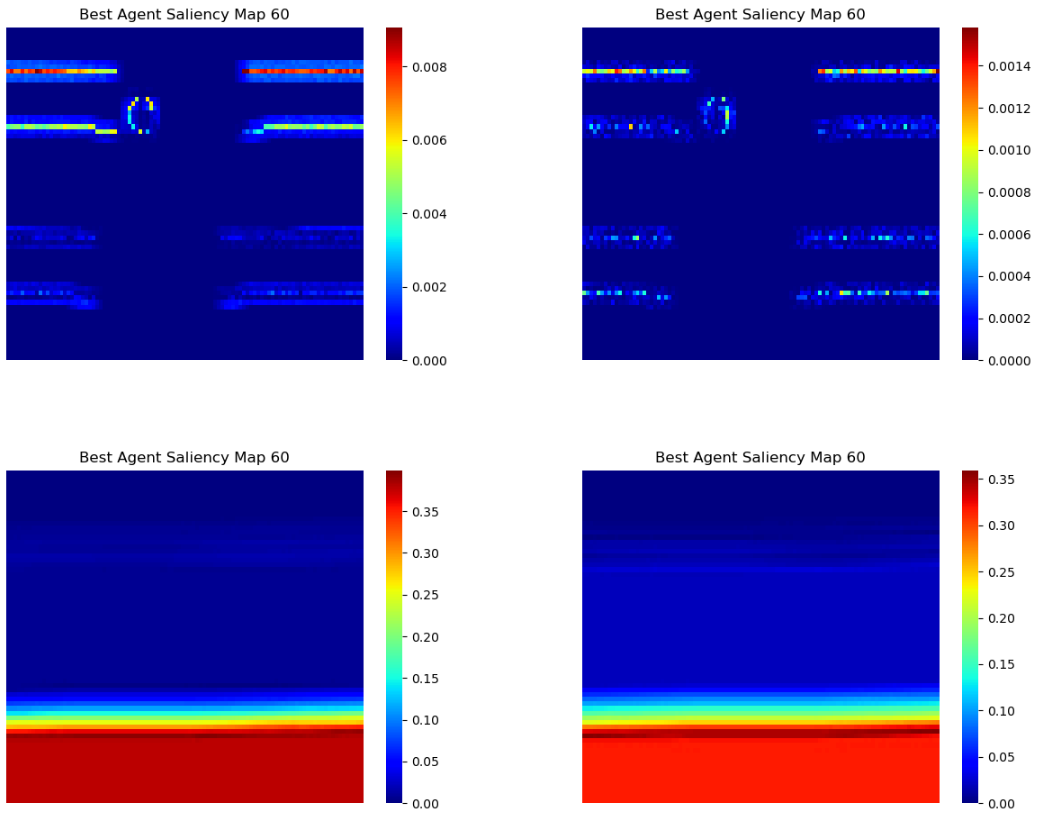


Figure 4.9: Shown are example saliency maps for the activation function, difference frame, and weight initialization experiments. Each map was generated using the same frame and the best model from that experiment. From left to right on the top: agents A5 and A6. On the bottom row are the saliency maps for agents A4 and A3.

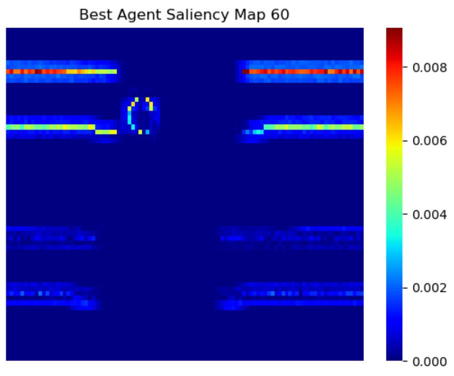


Figure 4.10: The best saliency map produced by any agent was from agent A5. This is considered the best map as it has the tightest halo around the pipes, which are all very clearly depicted in various shades of reddish-orange or yellow-green. Notice also how the most important pixels in the map are associated with the edges of the closest pipe to the player.

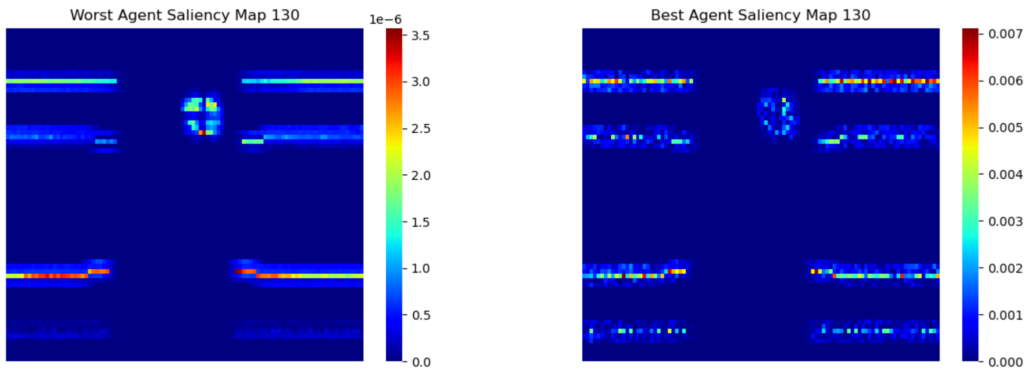


Figure 4.11: Shown on the left is the saliency map of the worst performing A6 agent, whose best map is on the right.

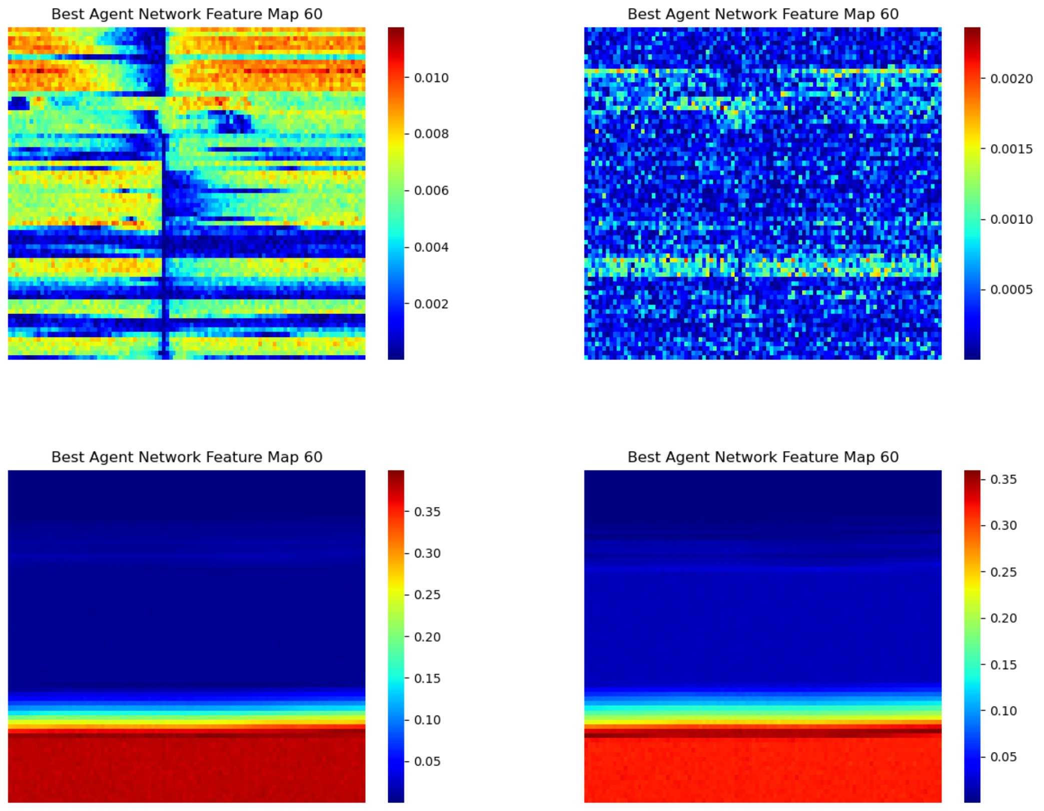


Figure 4.12: shown are example feature maps for the activation function, difference frame, and weight initialization experiments. Each map was generated using the same frame and the best model from that experiment. From left to right on the top: agents A5 and A6. On the bottom row are the feature maps for agents A4 and A3.

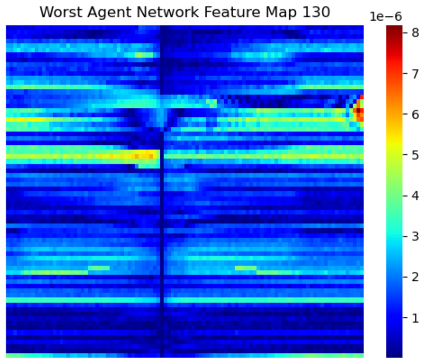


Figure 4.13: The feature map of the worst performing A4 agent.

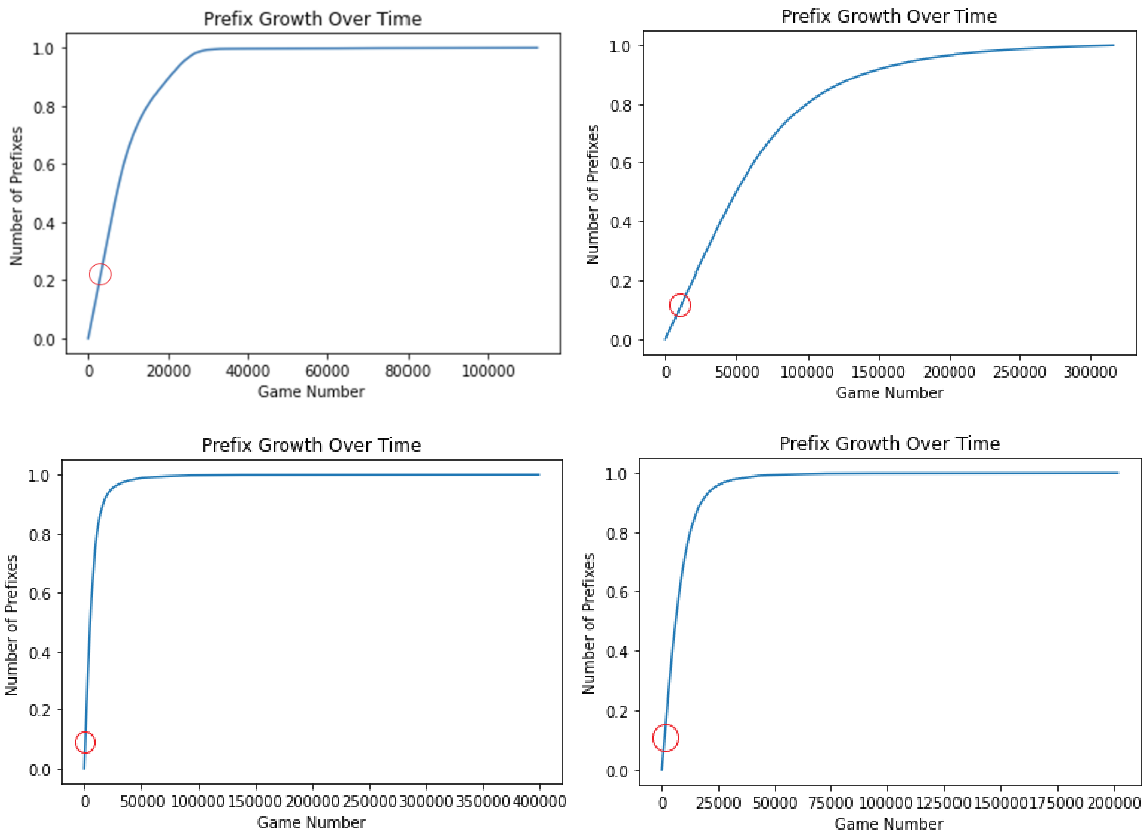


Figure 4.14: shown are example exploration curves for each of the experiments conducted. Each graph depicts the magnitude of the set of unique action strings over the course of the experiment. From left to right on the top: agents A5 and A6. On the bottom row are the exploration curves for agents A4 and A3.

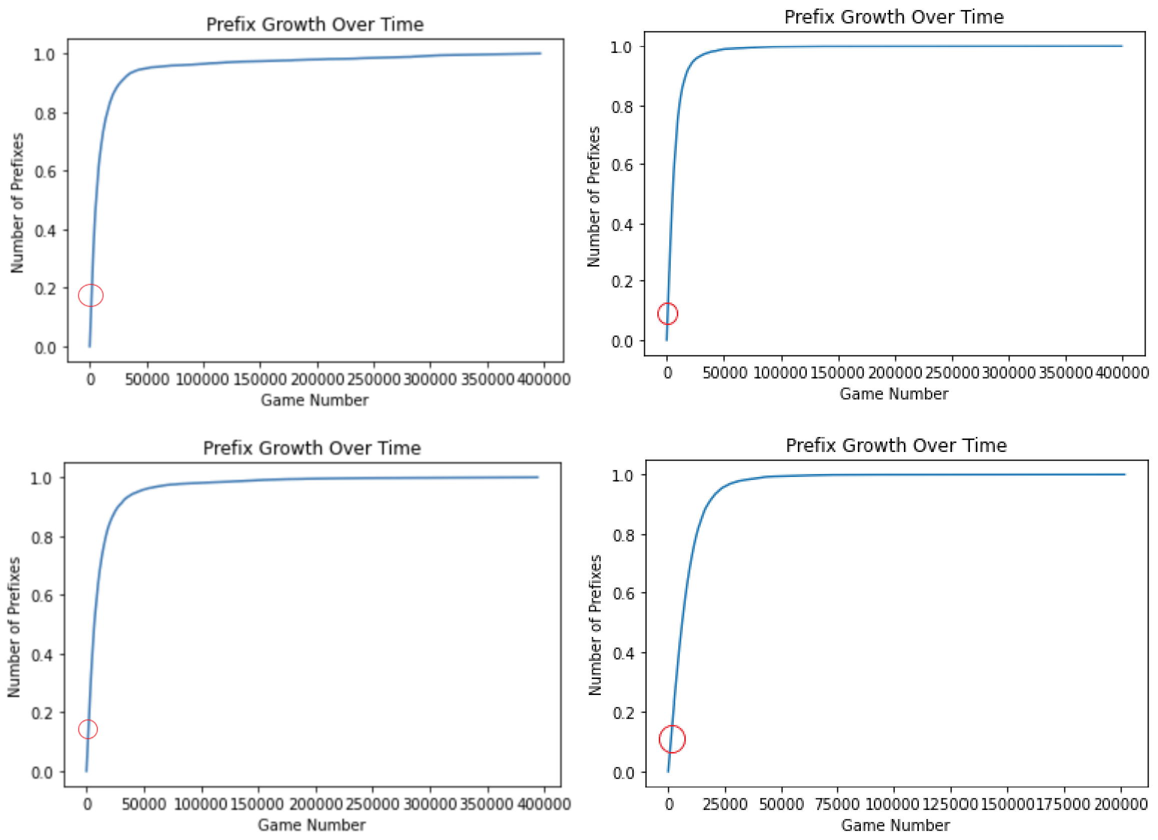


Figure 4.15: A comparison of agent explorations with each row containing the best and worst for each experiment, with the left column being best and the right column the worst. On the top row is the exploration curve for agent A4, on the bottom is agent A3.

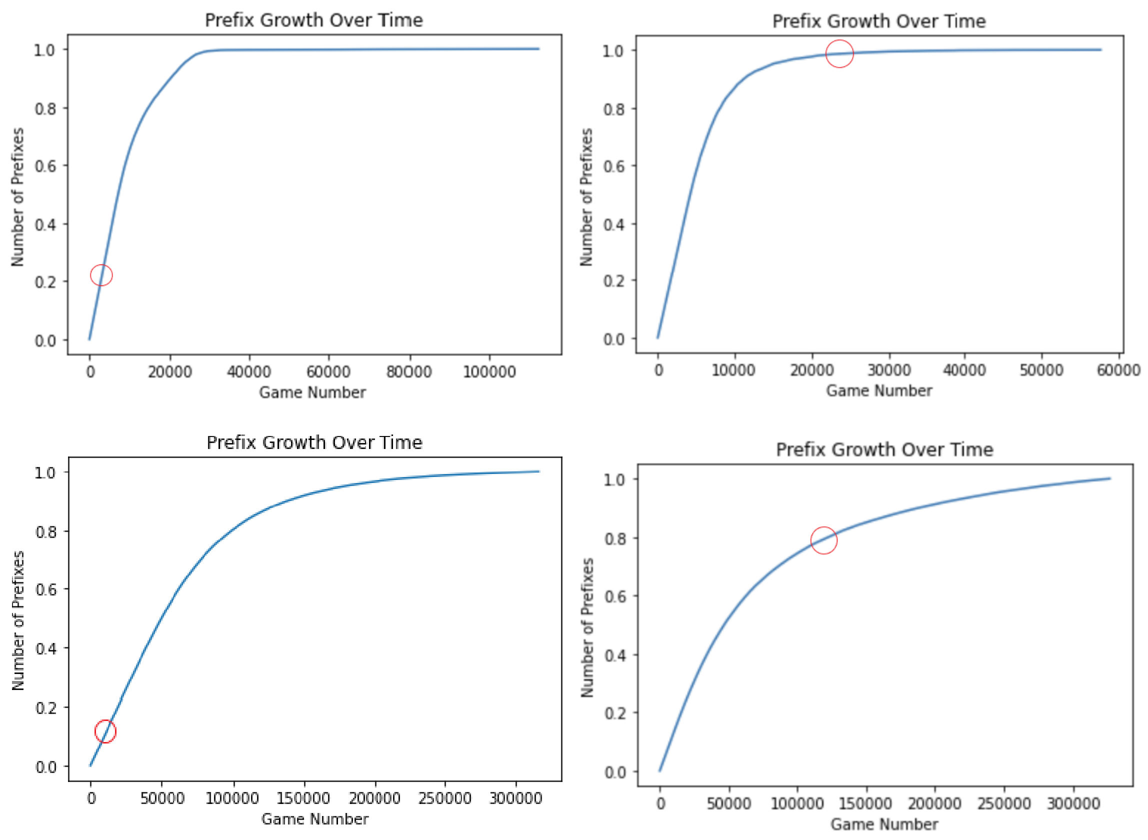


Figure 4.16: A comparison of agent explorations with each row containing the best and worst for each experiment, with the left column being best and the right column the worst. On the top row is agent A5 and on bottom is A6.

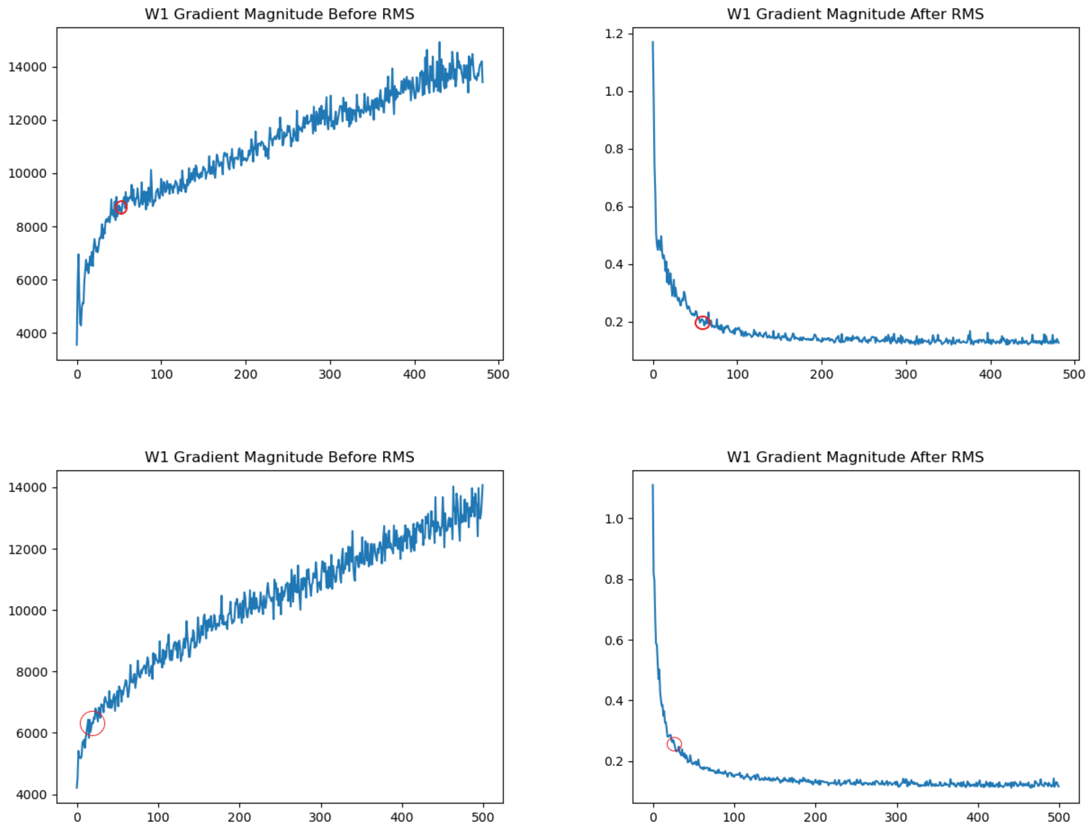


Figure 4.17: The magnitude of the gradient vector before RMSprop was applied and after for weights connecting the input to the hidden layer are shown above. On the top row are the plots for agents A5 and A6. These plots show that RMSprop is functioning properly, but that the network is actually moving further from an optimal policy with time.

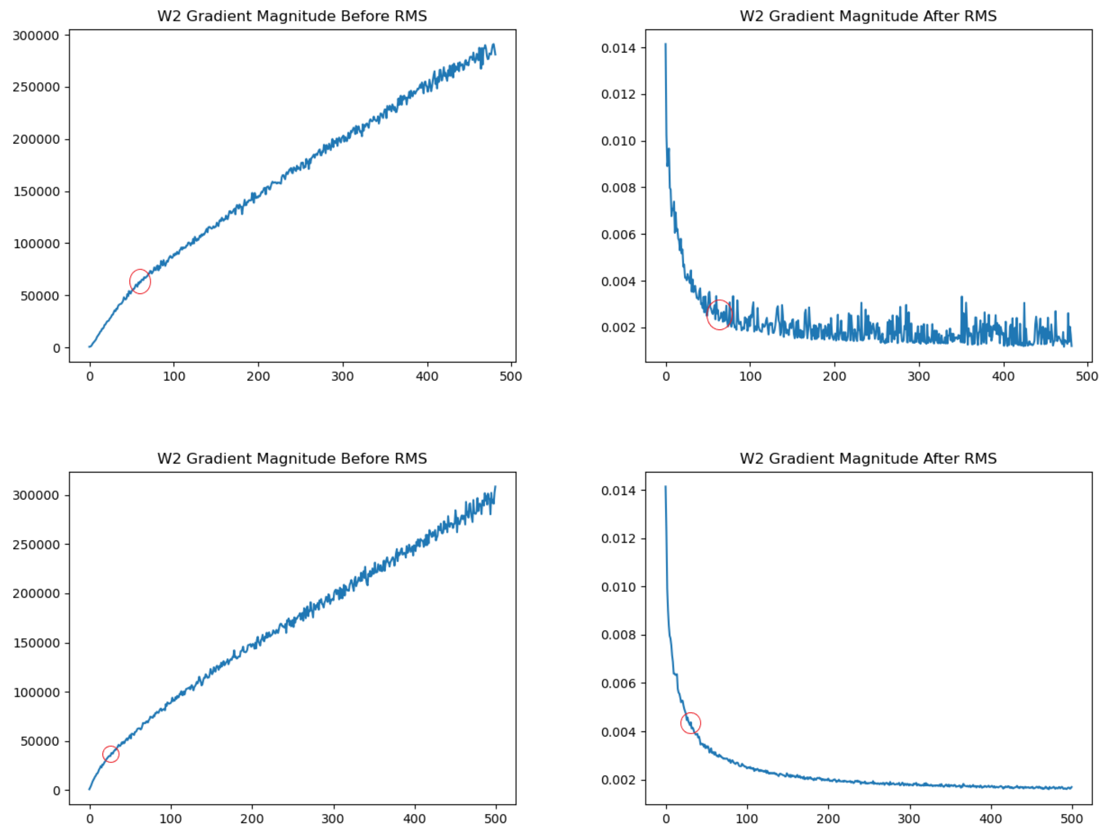


Figure 4.18: The magnitude of the gradient vector before RMSprop was applied and after for weights connecting the hidden layer to the output neuron are shown. On the top row are the plots for agent A8 and on bottom are the plots for agent A10. These plots also show that RMSprop is functioning properly, but that the network is actually moving further from an optimal policy with time.

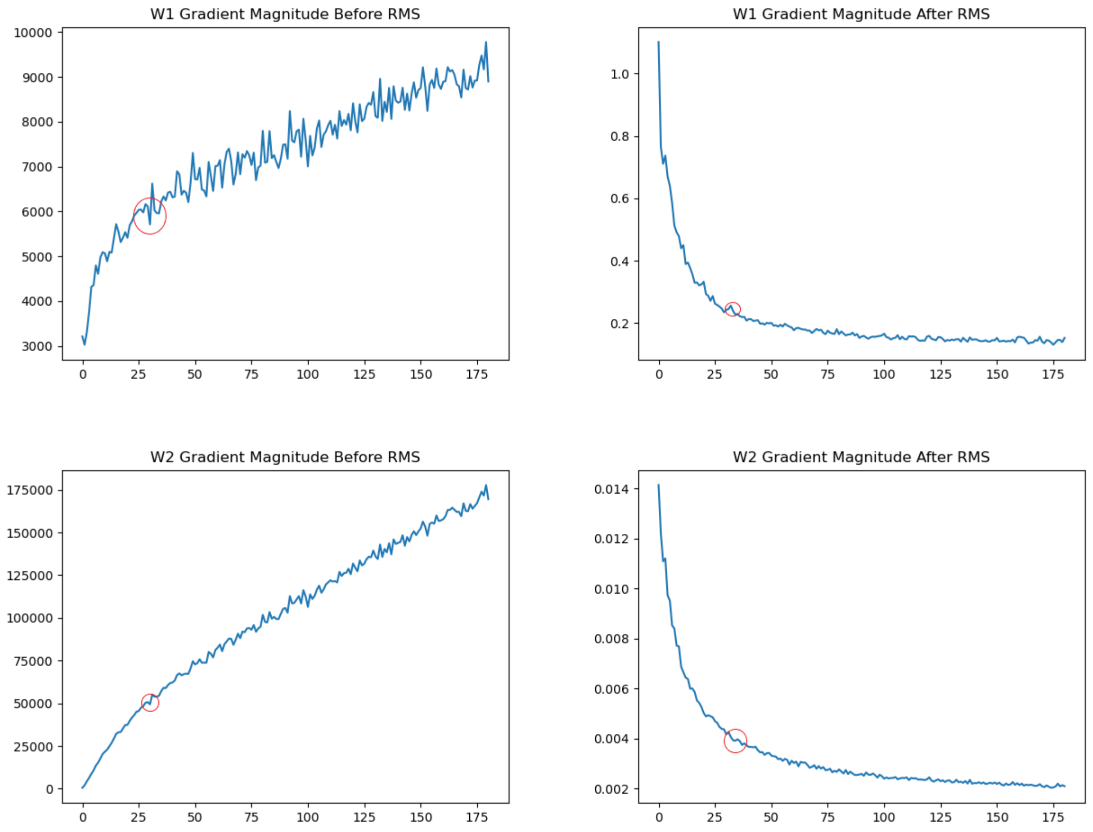


Figure 4.19: The magnitude of the gradient vector was again calculated after using L2 Normalization, and though the size of the magnitude decreased the value continued to increase with time. Each unit of the x-axis represents the number of batches of gradients were aggregated and applied using batch sizes of 200. These plots are for agent A8 before and after RMSprop.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5: Conclusions

Though capable of superhuman performance in many tasks, RL using neural networks can be very sensitive to implementation method and environment complexity. Our results show just how precise network design must be in order to comprehensively learn a successful policy for an environment.

A key challenge facing the RL and Artificial Intelligence (AI) communities is explaining how agents learn the policies or models that they do. At a high level, our work and results reflect the importance of explainability. In the rest of this chapter, we first draw several specific conclusions from the results and then comment on the limitations of our work and directions for future work.

5.1 Conclusions

- Hee's pathology was observed in every agent without exception. In analyzing the cause of the pathology in agents not utilizing human heuristics we discovered that over time the agents were learning to focus too far into the future when deciding what action to take at each step. Our evidence shows that though the heuristic may contribute to the onset of the pathology the topology itself is extremely sensitive to the environment it is being used in, and the network implementation has much more influence on collapse.
- The objective of generalizing Hee's pathology required the use of a similar network architecture on a similar environment. Though rebuilding an architecture is simple, analyzing a game environment for complexity similarity is not. To be certain that the environment is truly the cause of an issue, the researcher must design specific tests for all controllable components of the architecture, just as we did in later experimentation. There is no established method for determining if a network will have the required learning capacity to model an environment, and this is done in practice through "trial and error" tuning of hyper-parameters, weight initialization, activation and optimization.
- As has been previously mentioned, this research was hampered by the environment

chosen for use. There are no tools in existence that can compare two environments for complexity or similarity other than human intuition. Human intuition of environmental complexity is inherently flawed, and the lack of scientific methods to determine the best test environment for this research required extra experimentation to verify the suitability of the network. Through this experimentation we learned that the use of conventional optimizers such as RMSprop does not guarantee improved performance.

5.2 Source Code

All scripts used to conduct experiments as well as analysis of results can be found in a GitHub repository located at https://github.com/B-Atkinson/FlappyBird.git. All experimental data can be viewed inside of NPS's Hamming High Performance Computing cluster at /home/brian.atkinson/thesis/data.

5.3 Limitations

- Due to time constraints we were limited to studying only the FlappyBird game environment, and if time had permitted we would have attempted to generalize our results to others. As we began to observe a consistent result across our experiments we chose to design further experiments to help determine the root cause of the pathology before trying to generalize the pathology in a new environment.
- Given our finding that the gradients were becoming uncontrollable during training we would have liked to revisit Hee's experiments to see if the same condition was present. It would have been incredibly useful to understand if the gradient behaviors we observed were also present in Hee's work, as it would better explain why his human-assisted agents collapsed.
- A final limitation of this research was that there was no mathematical analysis of the gradient ascent being used by the network, once again due to time constraints. Because of this all the evidence in this body of work is empirical.

5.4 Future Work

Future research should be done to develop tools allowing researchers to compare environments, as this will not be the last time a behavior/result needs to be generalized. Ways this could be done are through comparison of degrees of freedom, number of available actions, how the environment changes with respect to the agent (does the agent move around it or is the agent reasonably static in the frame).

This research relied heavily on visual aids to understand the policy being learned, especially so with feature and saliency maps. The development of a better scoring function for both of those mapping tools using perturbation would also be beneficial. In this work, the feature maps were developed as a way of avoiding perturbation though the scoring function it uses potentially lacks granularity.

Future work should also be done to add more layers to the network architecture in order to generalize Hee's pathology by getting a functioning baseline agent. This environment maybe more complex but has the potential to yield meaningful results, and merits more network parameters. Our data shows similar behavior to that of Hee's in early stages of learning, and by adding more layers the baseline agents will likely be able to perform as desired. Once the baseline agents behave as they should by not exhibiting performance collapse, attention can then be turned to generalization of the pathology.

Future implementations should consider using CNNs as a way of efficiently extracting information from the frames and memory efficiency as compared to fully-connected networks. Many successful game-playing agents use deep CNNs to accomplish fantastic results, and if the pathology is indeed a limitation of shallow neural networks, progression to CNNs would be a logical next step.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, Jan. 2016. [Online]. Available: https://www.nature.com/articles/nature16961

[2] W. Bradley Knox and P. Stone, "TAMER: Training an agent manually via evaluative reinforcement"," in *2008 7th IEEE International Conference on Development and Learning*. IEEE, Aug 2008, pp. 292–297. [Online]. Available: http://ieeexplore.ieee.org/document/4640845/

[3] R. Hee, Brandon, "Understanding the adverse effects of accelerating reinforcement learning with human trainers," master's thesis, Dept. Computer Science, Naval Postgraduate School, Sep 2020. [Online]. Available: https://calhoun.nps.edu/handle/10945/66081

[4] W. B. Knox and P. Stone, "Combining manual feedback with subsequent MDP reward signals for reinforcement learning," AAMAS '10: The Ninth International Conference on Autonomous Agents and Multiagent Systems. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/1838206.1838208

[5] G. Warnell, N. Waytowich, V. Lawhern, and P. Stone, "Deep TAMER: Interactive agent shaping in high-dimensional state spaces," *arXiv:1709.10163 [cs.AI]*, 2018.

[6] M. Walton, Garret, "An evaluation of using deterministic heuristics to accelerate reinforcement learning," master's thesis, 2018. [Online]. Available: https://calhoun.nps.edu/handle/10945/61297

[7] V. Yordanova, "Markov Decision Process (MDP) framework," ResearchGate. [Online]. Available: https://www.researchgate.net/figure/Markov-Decision-Process-MDP-framework_fig10_324483497

[8] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch18.html

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv:1509.02971 [cs, stat]*, July 2019. [Online]. Available: http://arxiv.org/abs/1509.02971

[10] A. Karpathy, “Deep Reinforcement Learning: Pong from Pixels.” [Online]. Available: http://karpathy.github.io/2016/05/31/rl/

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous Methods for Deep Reinforcement Learning,” *arXiv:1602.01783 [cs]*, June 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[12] K. D. Harris and G. M. G. Shepherd, “The neocortical circuit: themes and variations,” *Nature Neuroscience*, vol. 18. [Online]. Available: http://www.nature.com/articles/nn.3917

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, Feb. 2015. [Online]. Available: https://www.nature.com/articles/nature14236

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” *arXiv:1312.5602 [cs]*, Dec. 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[15] S. Griffith, K. Subramanian, J. Scholz, C. L. Isbell, and A. Thomaz, “Policy shaping: integrating human feedback with reinforcement learning,” in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS’13. Red Hook, NY, USA: Curran Associates Inc.

[16] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm,” *arXiv:1712.01815 [cs]*, Dec. 2017. [Online]. Available: http://arxiv.org/abs/1712.01815

[17] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, Mar. 2010, pp. 249–256, iSSN: 1938-7228.

[18] G. Hinton, N. Srivastava, and K. Swersky, “Overview of mini-batch gradient descent,” University of Toronto. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[19] K. Team, *Keras documentation: RMSprop*. [Online]. Available: https://keras.io/api/optimizers/rmsprop/

[20] S. Greydanus, A. Koul, J. Dodge, and A. Fern, “Visualizing and Understanding Atari Agents,” *arXiv:1711.00138 [cs]*, 2018. [Online]. Available: http://arxiv.org/abs/1711.00138

[21] K. Simonyan, A. Vedaldi, and A. Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps,” 2014, arXiv: 1312.6034. [Online]. Available: http://arxiv.org/abs/1312.6034

[22] Li and Karpathy, “Visualizing and understanding convolutional neural networks,” 2016. [Online]. Available: http://cs231n.stanford.edu/slides/2016/winter1516_lecture9.pdf

[23] L. Lu, “Dying relu and initialization: Theory and numerical examples,” *Communications in Computational Physics*, vol. 28, no. 5, p. 1671–1706, Jun 2020. [Online]. Available: http://dx.doi.org/10.4208/cicp.OA-2020-0165

[24] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” Feb. 2015, arXiv: 1502.01852. [Online]. Available: http://arxiv.org/abs/1502.01852

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California