Research in Industrial Projects for Students

Sponsor

Advanced Micro Devices

Final Report

Exploration of Reinforcement Learning in Computer Games

Student Members

Jiajing Guan (Project Manager), *George Mason University*

jiajingguan@gmail.com

Patrik Gerber, *University of Oxford*

Elvis Nunez, *Brown University*

Kaman Phamdo, *University of Maryland*

Academic Mentor

Tonmoy Monsoor, mtonmoy@g.ucla.edu

Sponsoring Mentors

Nicholas Malaya, nicholas.malaya@amd.com

Abhinav Vishnu, abhinav.vishnu@amd.com

Alan Lee, alan.lee@amd.com

Date: August 22, 2018

This project was jointly supported by Advanced Micro Devices and NSF Grant DMS-0931852.

Abstract

Reinforcement learning (RL) is an area of machine learning where algorithms learn from experience while dynamically interacting with an environment. RL approaches have recently achieved superhuman performance in games such as Go, Chess, and classic video games. However, these algorithms are sensitive to perturbations in the environment and often struggle or catastrophically fail when applied to environments that differ from where they were trained. RL methods must become more robust if they are to become certified and deployed in the real-world. This project investigated the performance of the Deep Q-Network (DQN) algorithm on an unsolved OpenAI car racing game challenge. The report begins with an overview of reinforcement learning and the methods used for the car racing game. This is followed by a description of a tuned DQN algorithm that solves this particular challenge. The algorithms sensitivity to changes in feedback and input image characteristics are discussed. This report concludes with a discussion of the generalizability of these algorithms, techniques for improving their robustness, and future work needed to achieve autonomous RL algorithms.

Acknowledgments

We would like to acknowledge our academic mentor, Tonmoy Monsoor, for guiding us through the project. We would also like to thank our industrial mentors, Nicholas Malaya, Abhinav Vishnu and Alan Lee for sponsoring this project and providing support along the way. We are grateful for the help we have received from the RIPS program director, administrators, and colleagues. This project is supported through the National Science Foundation grant DMS-0931852.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Sponsor

Advanced Micro Devices, Inc. (AMD) is an American multinational semiconductor company based in Santa Clara, California, that develops computer processors and related technologies for business and consumer markets. AMD's main products include microprocessors, motherboard chipsets, embedded processors and graphics processors for servers, workstations and personal computers, and embedded systems applications. AMD is interested in developing powerful hardware devices and software for machine learning tasks. The results from this project will provide insights for improving hardware efficiency needed for deep reinforcement learning algorithms.

1.2 Motivation

Supervised machine learning approaches operate in distinct modes of training and inference. During training, an algorithm is shown labeled data and learns a representation of the underlying distribution of the data. After training, the model can be deployed to infer predictions based on the information it learns during training. However, supervised machine learning approaches are limited because they require labeled data, and labeled data is not always available.

Reinforcement learning is a goal-oriented approach that is fundamentally different from supervised machine learning. The objective of reinforcement learning is to provide a means for an agent to continually learn while dynamically interacting with an environment. As the agent performs actions within the environment, it receives feedback through rewards and learns new behavior. The goal of the algorithm is to find the optimal actions at each given state. This is distinct from typical approaches in artificial intelligence, where an algorithm must first learn by viewing examples of a task being performed before it is capable of making predictions for this same task.

However, there are many challenges to overcome in the field of reinforcement learning. First, reinforcement learning fails in area where it requires long-term strategy. Second, reinforcement learning requires a huge amount of training time and experience. Third, reinforcement learning is not robust to adjust to perturbations in the environment. This impediment is the main focus of this project. The goal is to investigate the sensitivity of reinforcement learning algorithms and try to develop a robust algorithm that can adapt to changes in the environment.

1.3 Problem Statement

This project focused on improving the flexibility of reinforcement learning algorithms, allowing them to adapt to perturbations in the environment. We explored common frameworks and algorithms in reinforcement learning and investigated techniques to reduce the power consumption and latency of the deployed model without compromising the accuracy of the system. We investigated the effects of enriching the feedback received by the agent to guide it towards more effective learning. Moreover, analysis of model efficiency revolved around characterizing the influence of hyperparameters and sensitivity to hardware devices.

1.4 Approach

Computer games were used as a sandbox to explore the sensitivity of reinforcement learning algorithms. There are several advantages with using computer games: first, computer games have well-defined rules; second, they provide an efficient avenue for collecting data; third, the environment can be easily changed in order to induce perturbations. In particular, our investigations focused on an unsolved car racing challenge provided by the OpenAI Gym [2]. The environment in the car racing game is interesting due to its application to autonomous cars.

First, we implemented basic reinforcement learning algorithm with success in simple classic control games. Next, we implemented reinforcement learning algorithm incorporating deep learning, which produced promising results on a few classic control games and the Atari Breakout game. We then implemented deep Q-learning on the car racing game and explored the effect of reward clipping and regularization on performance. Lastly, we explored data compression in inputs to investigate how to reduce the storage required by the algorithm.

1.5 Overview

In this report, we will first explain the mathematical foundations of reinforcement learning in Chapter 2. Then we will focus on Q-learning and its implementation in Chapter 3. We will present some of our successful results on some of the simple classic control and Atari games in Chapter 4. We will discuss the results and findings while attempting to solve a car racing game in Chapter 5 and 6. Lastly, in Chapter 8, we provide ideas for future works needed to be done to make the algorithm more robust.

Chapter 2

Reinforcement Learning

In this chapter, we introduce the fundamental concepts of reinforcement learning algorithms. We start with a broad overview of reinforcement learning and dive into the basic principles of Markov Decision Processes, Value Functions, Bellman Optimality Equations and Temporal Difference Learning.

2.1 Reinforcement Learning Overview

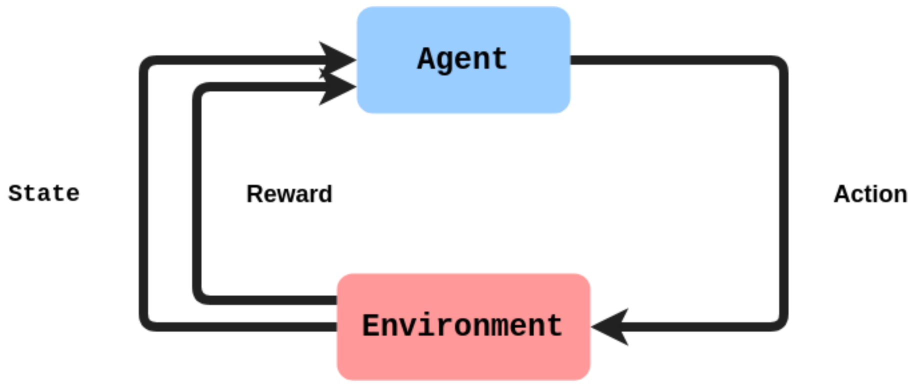


Figure 2.1: Interaction of Agent and Environment

Following *Reinforcement Learning* by Sutton and Barto [11], we define Reinforcement Learning as the field dedicated to the study of the Reinforcement Learning Problem (RLP). The Reinforcement Learning Problem can be defined broadly as the problem of learning from interaction to achieve a goal. The decision maker is called the *agent*, while the world it interacts with is called the *environment*. These two interact continually: the agent acts and the environment responds accordingly, presenting new situations to the agent. The environment also provides feedback to the agent via a scalar *reward* signal. The goal of reinforcement learning is to learn a strategy, called *policy*, that maximizes the expected cumulative reward the agent can receive. This process is illustrated in Figure 2.1. In the following sections, we base our treatment of the material on [12].

2.2 Markov Decision Processes

The mathematical framework used to study the RLP, is the theory of Markov Decision Processes (MDPs), which allows for the modeling of sequential decision-making processes, subject to Markovian conditional independence assumptions. Here we restrict ourselves to countable MDPs, however there exist extensions to continuous state-action MDPs too.

Definition. *A Markov decision process is a tuple* $(\mathcal{S}, \mathcal{A}, \mathbb{P}, \gamma)$ *where*

1. $\mathcal{S}$ *is a finite set of states*
2. $\mathcal{A}$ *is a finite set of actions*
3. $\mathbb{P} := \mathbb{P}\left(S_{t+1} = \cdot, R_{t+1} = \cdot \mid S_t = \cdot,\ A_t = \cdot\right) : \mathcal{S} \times \mathbb{R} \times \mathcal{S} \times \mathcal{A} \to [0,1]$ *is the state transition probability matrix*
4. $\gamma \in [0,1]$ *is the discount factor*

Given an MDP $\mathcal{M}$, the interaction between the agent and its environment happens as follows: Let $t \in \mathbb{N}$ denote the current time, and let S_t and A_t denote the state of the environment and the action selected by the decisionmaker, respectively. Upon receiving the action chosen by the agent, the environment makes a transition according to the dynamics of $\mathbb{P}$:

$$(S_{t+1}, R_{t+1}) \sim \mathbb{P}\left(S_{t+1} = \cdot, R_{t+1} = \cdot \mid S_t, A_t\right) \tag{2.1}$$

Definition. *A policy* $\pi(\cdot \mid \cdot) : \mathcal{A} \times \mathcal{S} \to [0,1]$ *is a conditional probability distribution over actions, given the state.*

Note that the policy fully defines the behavior of the agent. The last notion we need to define the problem is the *return.*

Definition. *The return underlying a policy* π *is defined as the total discounted sum of the rewards incurred:*

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t R_{t+1} \tag{2.2}$$

The goal of the agent is to learn a policy that performs well. Our measure of performance is the *expected return*–a rather natural metric.

2.3 Value Functions

Given the expected return characterization of performance, we can compare two policies and decide which one leads to better rewards. But how do we find good policies? The optimal value $V^*(s)$ of $s \in \mathcal{S}$ is the highest achievable expected return when the process is started from state s. We call $V^* : \mathcal{S} \to \mathbb{R}$ the optimal value function, and if a policy π_* achieves these optimal values in all states, it is called an *optimal policy.* We can extend the notion of value function to arbitrary policies:

Definition. *The value function $V^{\pi} : \mathcal{S} \to \mathbb{R}$ underlying the policy π is defined as*

$$V^{\pi}(s) = \mathbb{E}\left[\mathcal{R} \mid S_0 = s\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s\right] \tag{2.3}$$

where the values R_{t+1} are obtained following policy π.

A related object is the action-value function of a policy π.

Definition. *The action-value function $Q^{\pi} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ underlying the policy π is defined as*

$$Q^{\pi}(s,a) = \mathbb{E}\left[\mathcal{R} \mid S_0 = s, A_0 = a\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a\right] \tag{2.4}$$

where the values R_{t+1} are obtained following policy π.

Similar to the way we defined V^*, we can define the optimal action-value function Q^*, given to rise by the optimal policy π_*. Due to the Markov property, given V^* and $\mathbb{P}$, we can easily devise an optimal policy: in a given state, choose the action that leads to maximal average V^* value. With Q we can do even better: Given just Q^*, the policy $\pi(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a)$ is optimal. Note that in the MDPs considered here, an optimal policy always exists.

2.4 Bellman Optimality Equations

Denote $r(s,a) := \mathbb{E}\left[R_{t+1} \mid S_t = s,\ A_t = a\right]$. Then the optimal value function V^* and Q^* are connected by the following equations:

$$\begin{aligned} V^*(s) &= \sup_{a \in \mathcal{A}} Q^*(s,a) && (2.5)\\ Q^*(s,a) &= r(s,a) + \gamma\ \mathbb{E}\left[V^* \middle| S_t = s,\ A_t = a\right] && (2.6)\\ &= r(s,a) + \gamma \sum_{w \in \mathcal{S}} \mathbb{P}\left(S_{t+1} = w \mid S_t = s,\ A_t = a\right) V^*(w) && (2.7)\\ & && (2.8) \end{aligned}$$

These can be shown to hold by simple conditioning arguments. Intuitively they can be thought of as saying the following: Suppose we are following an optimal policy π_*. By optimality, in state s, we must choose an action with maximal state-action value Q^*. But then it must be that V^* is equal to this maximal value, and Equation (2.5) follows. Equation (2.6) follows by a similar argument. Combining these two equations gives rise to the Bellman Optimality Equations:

$$\begin{aligned} V^*(s) &= \sup_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{w \in \mathcal{S}} \mathbb{P}\left(S_{t+1} = w \mid S_t = s,\ A_t = a\right) V^*(w) \right\} && (2.9)\\ Q^*(s,a) &= r(s,a) + \gamma \sum_{w \in \mathcal{S}} \mathbb{P}\left(S_{t+1} = w \mid S_t = s,\ A_t = a\right) \sup_{a \in \mathcal{A}} Q^*(w,a) && (2.10) \end{aligned}$$

A short proof of Equation (2.10) is given in the next section. Introducing new notation, we can rewrite Equation (2.9) more succinctly, in a way that suggests a solution:

Definition. *Define the Bellman Optimality operator $T^* : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$, by*

$$(T^*V)(s) = \sup_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{w \in \mathcal{S}} \mathbb{P}\left(S_{t+1} = w \mid S_t = s,\ A_t = a\right) V(w) \right\} \tag{2.11}$$

Then the Bellman Optimality Equation for V can be rewritten as

$$T^*V^* = V^*$$

An analogous operator exists for Q^* that we don't cover here. Under certain conditions, it can be shown that both (nonlinear) Bellman Optimality operators are in fact contractions. Thus, by a fixed point argument, they have unique solutions that can be iteratively approximated. Such iterative approximations have been the bases of many of the early reinforcement learning algorithms.

2.5 Temporal Difference Learning

Instead of pursuing these fixed point approximation methods, we turn to the more sophisticated Temporal Difference (TD) Learning approach, one of the most significant ideas in reinforcement learning. Recall the optimality equation for Q^*:

$$\begin{aligned}
Q^*(s_t, a_t) &= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{i=t}^{\infty} \gamma^{i-t} R_{i+1} \middle| S = s_t, A = a_t \right] && (2.12) \\
&= \mathbb{E}\left[R_{t+1} + \mathbb{E}\left[\sum_{i=t+1}^{\infty} \gamma^{i-t} R_{i+1} \middle| s_t, a_t, S_{t+1} \right] \middle| S = s_t, A = a_t \right] \\
&\overset{*}{=} \mathbb{E}\left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} \mathbb{E}\left[\sum_{i=t+1}^{\infty} \gamma^{i-(t+1)} R_{i+2} \middle| S_{t+1}, A_{t+1} = a' \right] \middle| S = s_t, A = a_t \right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q^*(S_{t+1}, a') \middle| S = s_t, A = a_t \right] && (2.13)
\end{aligned}$$

where the starred equality follows by the Markov property and the fact that we follow the optimal policy π_*. This splits Q^* into two quantities: the immediate reward and the discounted future rewards. More importantly, Equation (2.13) reveals structure that immediately yields a suitable iterative update shown in Equation (2.14).

$$Q(s_t, a_t)_{j+1} = Q(s_t, a_t)_j + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right) \tag{2.14}$$

where $\alpha \in [0, 1]$. As α approaches 1, the Q value function is constructed to weight the future rewards more heavily.

This update is in the standard iterative update format $Q(s_t, a_t)_{j+1} = Q(s_t, a_t)_j + \alpha \times \text{error}$ where the error captures the difference between what the value $Q(s_t, a_t)$ *should* be at the current iteration: $R_{t+1} + \gamma \max\limits_{a' \in \mathcal{A}} Q(s_{t+1}, a')$ and what it is at the current iteration: $Q(s_t, a_t)$.

In the next section we expand on this idea, and introduce the Q-learning algorithm, which will form the basis of our investigations.

Chapter 3

Q-Learning

The previous chapter presented the backbone of Q-learning. In this chapter, we explain the details for implementing Q-learning algorithms. In section 3.1, we demonstrate a tabular approach with the Cart Pole game as an example. The detailed results will appear in Chapter 4. In section 3.2, we explain the theoretical background for deep Q-learning. We introduce regularization in section 3.4, which is the technique we used to improve our results in Chapter 6.

3.1 Tabular Q-Learning

Having established the theoretical aspects of Q, it is necessary to find a suitable representation that can be found computationally. Two such representations will be discussed; here, a tabular representation is described.

In the tabular setting, Q will be represented by a look-up table where every row corresponds to a state of the game and the respective columns represent actions. Due to finite computational memory, storing the value of Q for every possible state-action pair is infeasible when state spaces are continuous. Consequently, continuous parameters from state spaces must be discretized into disjoint bins. An example of such a discretization is given in Table 3.1 for the game of Cart Pole.

$$\begin{aligned} x_t \in [-2.4, 2.4] &\Rightarrow \{[-2.4, -0.8), [-0.8, 0.8), [0.8, 2.4]\} \\ v_t^x \in [-\infty, \infty] &\Rightarrow \{[-2, -1), [-1, 0), [0, 1), [1, 2]\} \\ \theta_t \in [-0.419, 0.419] &\Rightarrow \{[-0.419, -0.14), [-0.14, 0.14), [0.14, 0.419]\} \\ v_t^\theta \in [-\infty, \infty] &\Rightarrow \{[-2, -1), [-1, 0), [0, 1), [1, 2]\} \end{aligned}$$

Table 3.1: Since the state space is continuous, in order to represent the Q function as a table, it is critical to discretize the state space.

Once the state space has been discretized, each state can be mapped to its corresponding bin which can then be mapped to a row in the tabular representation of Q. An example of Q for Cart Pole at some iteration is given in Table 3.2.

	Left	Right
[1, 1, 1, 1]	0.8	0.3
[1, 1, 1, 2]		0.6
⋮	⋮	⋮
[2, 1, 2, 3]	0.15	0.11
⋮	⋮	⋮
[3, 4, 3, 4]	1.2	0.9

Table 3.2: This table is an example of how to represent Q function as a table. This table in particular stores the Q function for the game Cart Pole, which was solved by the team. The details about solving Cart Pole is in Chapter 4

3.1.1 Drawbacks

While the tabular representation of Q is favorable for pedagogical purposes and its implementation ease, it does not generalize well to games with large state spaces. For example, a gray-scale image with 1000 pixels has a state space size of 256^{1000}. Storing values for each state is noticeably infeasible. Rather than storing the value of Q for every possible state, we will use a function to approximate Q. In particular, we will use a neural network to approximate Q, as discussed in the next section.

3.2 Deep Q-Learning

3.2.1 Overview

In this section we outline the Deep Q-Learning (DQN) algorithm, which we took as the starting point of our investigation into Deep Reinforcement Learning. The basic idea of DQN is to represent the state-action value function Q as a convolutional neural network–recent work has shown deep neural networks to work well in this setting.

There are three simple modifications one has to make to the tabular Q-Learning algorithm to get what is now commonly called the DQN-algorithm [8]. We outline these modifications below.

3.2.2 Q as a neural network

The first step is to replace the function $Q : S \times A \to \mathbb{R}$ by a neural network $Q(\cdot, \cdot \mid \theta) : S \times A \to \mathbb{R}$, where θ denotes the weights of the model. The resulting network, called the Q-network, takes as its input a state and an action, and outputs the estimated Q-value of the pair. But the question arises: How do we find good parameters θ for this model? Suppose that at timestep i we have current estimates θ_i for the weights of the network. We take the loss function to be the mean squared TD-error, the same error term as used in the simple tabular Q-learning algorithm, derived from the Bellman optimality equation.

$$L_i(\theta_i) = \mathbb{E}\left[\left(\mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a' \mid \theta_{i-1})\middle| s_t, a_t\right] - Q(s_t, a_t \mid \theta_i)\right)^2\right] \tag{3.1}$$

Because the dynamics of the environment the agent is operating in is unknown, the expectation above cannot be evaluated; we get around this by using Stochastic Gradient Descent (SGD). Using this method, we get the following update rule for the weights θ :

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}\left[\left(R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a' \mid \theta_{i-1}) - Q(s_t, a_t \mid \theta_i)\right) \nabla_{\theta_i} Q(s_t, a_t \mid \theta_i)\right] \tag{3.2}$$

$$\theta_{i+1} = \theta_i - \alpha \, \nabla_{\theta_i} L_i(\theta_i)|_{\theta_i} \tag{3.3}$$

3.2.3 Experience Replay and Fixed Q-Targets

In theory, SGD only works under some independence assumptions on the data we feed it. We need to ask ourselves whether these assumptions are reasonable in this setting. As we are playing the game, and updating our weights at each timestep, the sequence of transitions visited by the agent is highly correlated: We cannot reasonably expect the game's state to be independent of its immediate past. Therefore, we use the method of experience replay to decorrelate the data and to achieve higher data efficiency. Moreover, this helps reduce the risk of getting stuck in unwanted feedback loops.

At each timestep, we store the tuple $(s_t, a_t, R_{t+1}, s_{t+1})$ in the replay memory denoted by $\mathcal{D}$. Instead of updating θ based on the most recent transition, we instead sample uniformly from $\mathcal{D}$, and use these past experiences to perform the gradient descent step. In practice, we sample a minibatch of transitions $(s_j, a_j, R_{j+1}, s_{j+1})_{j \in J}$ where the minibatch size $|J|$ is usually chosen to be a multiple of 2 in the range of 16 to 64. This way we reuse all experiences multiple times during training. Another parameter that arises from experience replay is the replay memory capacity - the maximum length of the replay memory - which is chosen in the range of 10^3 to 10^6.

The final modification that needs to be made in order for DQN to work is the method of fixed Q-targets. In the loss function, in the absence of supervision, one must bootstrap from the current weights θ_i in order to arrive at the TD-error, an estimate of Q's accuracy. However, instead of using the most recent estimates θ_i, one can use a separate, periodically updated set of weights, usually denoted by θ^-. After the modification, the loss function becomes

$$L_i(\theta) = \mathbb{E}\left[\left(\mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a' \mid \theta^-)\middle| s_t, a_t\right] - Q(s_t, a_t \mid \theta)\right)^2\right] \tag{3.4}$$

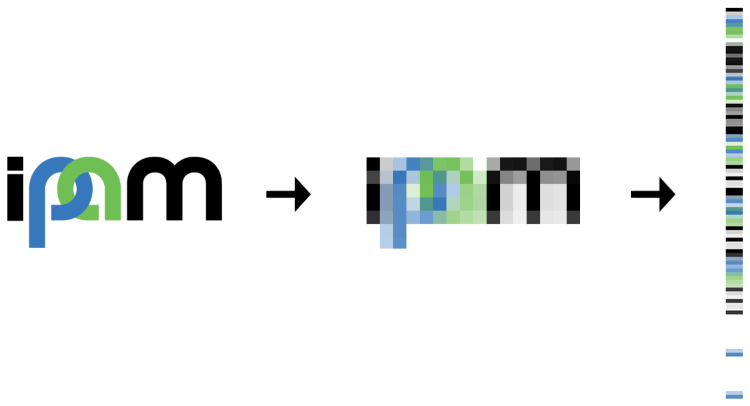


Figure 3.1: This is an illustration of the pre-processing method commonly employed in RL algorithms.

3.3 Q-learning in practice

We now address the question of how the data used to train the agent is generated. In practice, Q is initialized randomly, thus immediately taking the actions suggested by Q as in Equation (2.14) will introduce bias towards the initialization of Q. As such, the agent is expected to explore the state space and determine which actions are more suitable for certain states. Consequently, a parameter $\epsilon \in [0, 1]$ will be used to determine which action to take at each iteration: with probability ϵ, a random action will be taken; with probability $1 - \epsilon$ the action suggested by Q will be taken. In more complex computer games, the only observation the agent is given by its environment is an RGB image of the screen for each frame of the game. We process these images, in order to be able to use them as input to the Q-network. We do so following the methods used by Mnih et al. in *Playing Atari with Deep Reinforcement Learning* [8]: at each timestep, we take the most recent four frames, convert them to grayscale, downsample to a size of 84×84, and then stack them to produce the input to the network. A toy visualization of this process is given in Figure 3.1.

3.4 Regularization

A common problem among machine learning algorithms is that the algorithm often overfits to the training data and struggles to generalize to new inputs during testing. Many strategies have been developed to address this, possibly at the expense of increasing the training error. These strategies are collectively called regularization.

In deep learning, most regularization strategies are based on regularizing estimators, such as parameter norm penalties where parameters are penalized based on its norm. In other words, the larger the effect a parameter has on the outcome, the more heavily the parameter will be penalized, forcing the algorithm to not overfit to certain features during training.

In our project, we chose **dropout** as our regularization method as it provides a computationally inexpensive method for regularizing the models. The next section expands on the theoretical aspects of dropout.

3.4.1 Dropout

Dropout was first introduced by Srivastava et al. in 2014 [10] as a regularization method. Dropout drops units (along with their connections) from the neural network during training, preventing estimators from co-adapting to the training samples. Alternatively, dropout can be understood in the context of bagging. In particular, dropout separates a large neural network into an ensemble of of many sub-networks, which is similar to bagging. Bagging involves training multiple models and evaluating multiple models for each test case. However, bagging becomes impractical when it comes to large neural networks due to the huge amount of training time and computational power required. Dropout provides a way to implement bagging on large neural networks since it essentially trains sub-networks at once and infers the results from the collective knowledge of all sub-networks.

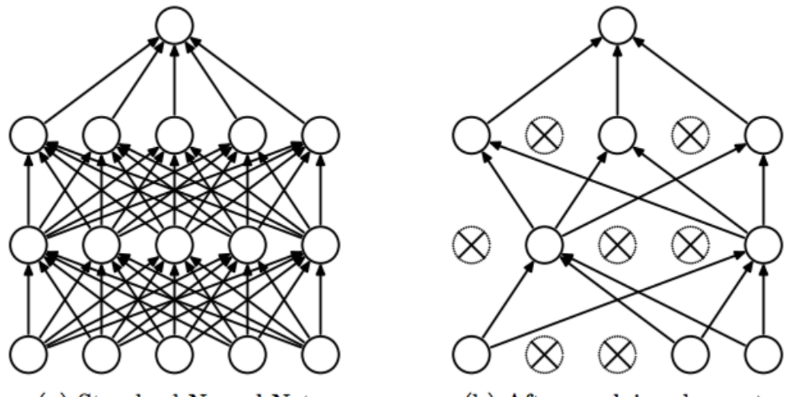


Figure 3.2: This figure shows how dropout decomposes a connect neural network into sub-networks. The cross marks on neurons indicate that the neurons are zeroed out and not included in the sub-network.

Dropout seeks to divide a large neural network into sub-networks by 'dropping out' some of the neurons in the network. During training, a mask containing zeros and ones is randomly generated based on a hyperparameter for determining the percentage of neurons to keep in the sub-networks. This mask is then applied to the neural network, zeroing the weights of some neurons. By performing this operation, a large neural network is decomposed into sub-networks. During inference, the result is calculated based on the geometric mean of the results from all sub-models given by

$$\tilde{p}_{ensemble}(y|\boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y|\boldsymbol{x}, \boldsymbol{\mu})} \tag{3.5}$$

where $\boldsymbol{\mu}$ is the randomly generated mask, d is the number of neurons that may be dropped, $\boldsymbol{x}$ is the input and y is the predicted output.

After taking the geometric mean, it is necessary to normalize the ensemble:

$$p_{ensemble}(y|\boldsymbol{x}) = \frac{\tilde{p}_{ensemble}(y|\boldsymbol{x})}{\sum_{y'} \tilde{p}_{ensemble}(y'|\boldsymbol{x})} \tag{3.6}$$

Dropout is a commonly used regularization method in deep neural networks. It is commonly believed that dropout is not necessary for deep RL algorithms because overfitting does not occur often in the RL realm. Due to the large state space, it is not likely for the model to overfit to the environment. However, we provide an counter example in Chapter 6 to demonstrate that the use of dropout could improve performance and save training time by generalizing model trained in a fixed environment.

Chapter 4

Preliminary Results

In this chapter, we briefly discuss the preliminary results we obtained on simpler games. In section 4.1, we present our successes with implementing tabular Q-learning on classic control games, Cart Pole and Mountain Car. We then show our results using deep Q-learning on a Atari game, Breakout, in section 4.2.

4.1 Tabular Q-Learning Results

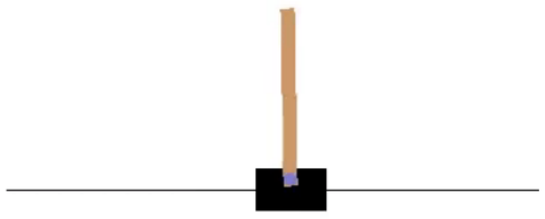


Figure 4.1: The game environment of a simple classic control game, Cart Pole

We initially investigated two simple games from the OpenAI Gym toolkit. The first game was Cart Pole, where the goal is to balance a pole in a cart for as long as possible. In this game, the state is represented as a four-dimensional vector containing the values for the position of the cart, the velocity of the cart, the angle of the pole, and the velocity of the pole. The game is considered solved when the agent achieves an average score ≥ 195 over 100 consecutive trials. Using the tabular Q-learning approach, the algorithm achieved a perfect score of 200 over 100 consecutive trials. This score was achieved with a bucket size of 1 for the position and velocity of the cart and a bucket size of 20 for the angle and velocity of the pole for state space discretization. This agent was trained for 500 episodes with a discount factor of 0.99, minimum learning rate of 0.05, and minimum explore rate of 0.1.

We also trained the tabular Q-learning algorithm for the Mountain Car game. In this game, the goal is for the car to reach the top of the mountain as quickly as possible. We initially faced challenges with this game due to the nature of the reward function. Since the reward function returns a negative reward for each time step, it is difficult to reinforce positive behavior before actually winning the game. As a result, the performance varied as a result

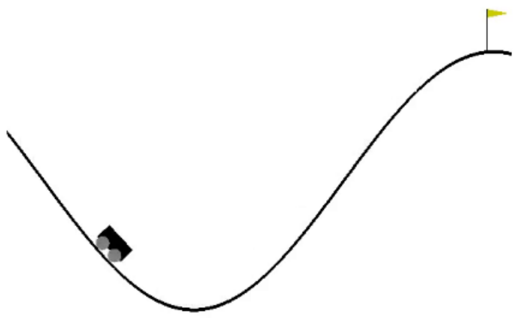


Figure 4.2: The game environment of a simple classic control game, Mountain Car

of slight changes in parameters. After experimentation, the algorithm performed best with a bucket size of 10 for position, bucket size 30 for velocity, discount factor of 0.99, fixed learning rate of 0.05 and fixed explore rate of 0.1. After training for 10,000 episodes, the algorithm achieved an average reward of -157.64 over 100 test episodes.

4.2 Deep Q-Learning Results

Next, we implemented the Deep Q-Learning algorithm on the Atari Breakout and Pong games using the convolutional architecture as defined in [8]. The first layer of the neural network was a convolutional layer with 32 kernels, 8x8 filter, and stride 4x4. The second was a convolutional layer with 16 kernels, 4x4 filter, and stride 2x2. The third layer was a dense layer with 256 neurons and a nonlinear rectifier. The final layer was a dense layer with 16 neurons and a linear rectifier. The performance was quantitatively analyzed using average Q-values. Assuming that S is a uniform sample of n fixed states, an estimate of the average Q-value was computed as follows.

$$Q_{avg} = \frac{1}{n}\sum_{i=1}^{n} max_a Q(s_i, a) \quad s_i \in S$$

Using this architecture, the algorithm was able to achieve promising results on both Breakout and Atari with the former shown in 4.3. The average Q-value appeared to increase over time and eventually converge, essentially recreating the results described in [8].

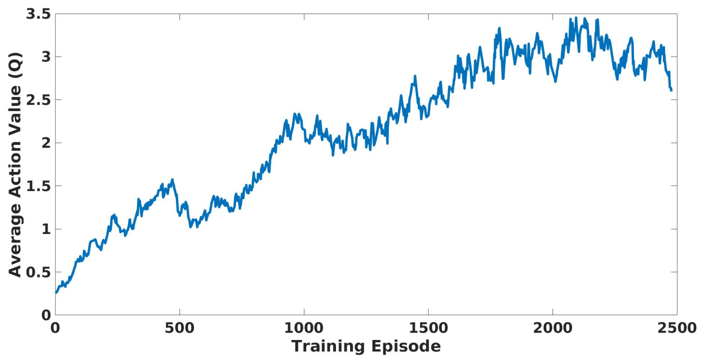


Figure 4.3: Average action value converges as expected

Chapter 5

Autonomous Driving

After verifying the correctness of our implementation by applying it to simpler environments such as Cart Pole and Breakout, we moved on to working on OpenAI's CarRacing environment, which is a 2-dimensional car racing game. In section 5.1, we go over the CarRacing environment in detail. We explore the effect of neural network structure, action space and reward feedback in the following sections.

5.1 The OpenAI CarRacing Environment

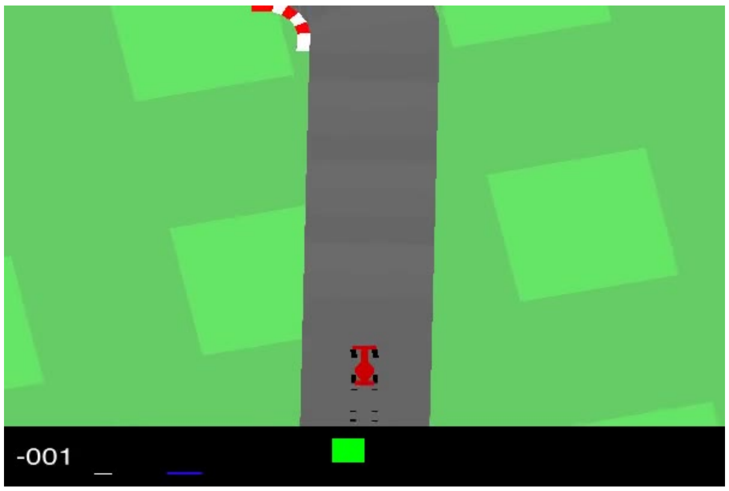


Figure 5.1: The game environment of the car racing game

In OpenAI's CarRacing environment - from here on referred to as the 'car racing game' - the emulator provides 96 by 96 RGB screenshots to the agent, which has control over a red racing car, as shown in Figure 5.1. Notice that the track is made up of distinct tiles. The reward signal is -0.1 for every frame passed and $\frac{1000}{N}$ for every tile visited, where N is the total number of tiles on the track. Thus the goal of the game is to finish the track as fast as possible, without missing any of the tiles. The actions come in the form of an array of three numbers. In the array, the first number indicates steer, the second number indicates acceleration and the third number indicates deceleration. The steer coordinate is continuous from -1 to 1 while the acceleration and the deceleration are continuous from 0 to

1. For example, if the agent finishes in 732 frames and never leaves the track doing so, the reward is 1000 - 0.1*732 = 926.8 points. The episode finishes when all tiles are visited or more than 1000 frames pass. As per OpenAI's guidelines, the environment is solved when an average score of 900 or more is achieved over 100 episodes.

We chose this environment for several reasons: first, the car racing game is similar to real life autonomous driving; second, the game environment can be easily altered, allowing for interesting exploration of robustness of Deep RL methods; third, according to the leaderboard on OpenAI's website, no one has successfully solved the game.

To be able to iterate faster, we created four different types of training environment of varying complexity: random short tracks, fixed one track, fixed three tracks and random tracks. In the random short track environment, there are 50 tiles for the agent to finish while in the other environments, there are approximately 300 tiles. Random short tracks and random tracks environment display randomly generated tracks for each episode while the other environments display the same one (fixed one track) or three tracks (fixed three tracks).

5.2 Network Architecture

When we first started working on the car racing environment, the 2 convolutional layer architecture used in Chapter 4 and [8] was used. However, considering the complexity of the game, a deeper three-convolutional architecture was tested.

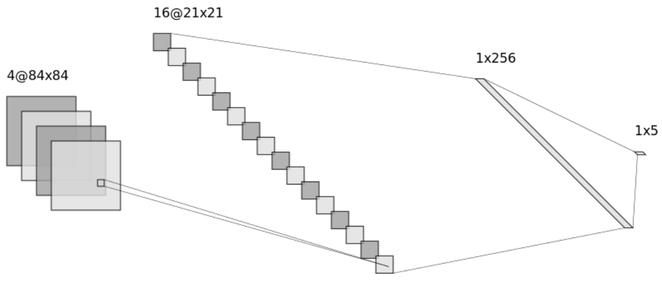


Figure 5.2: Neural Network with two convolutional layers. Specific parameters are in Table B.1.

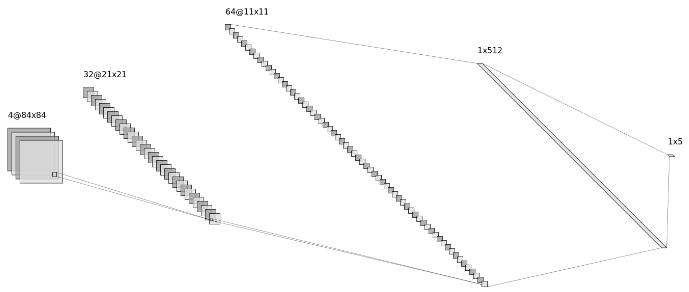


Figure 5.3: Neural Network with three convolutional layers. Specific parameters are in Table B.1.

We trained these two neural network structures in the random short tracks environment.

As shown in Figure 5.5, the model with two convolutional layers didn't perform as well as the one with three convolutional layers. The model with two convolutional layers also took

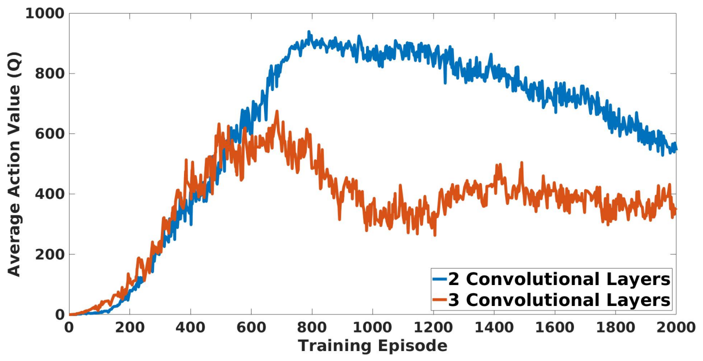


Figure 5.4: Average action values using two convolutional layers (orange) and three layers (blue).

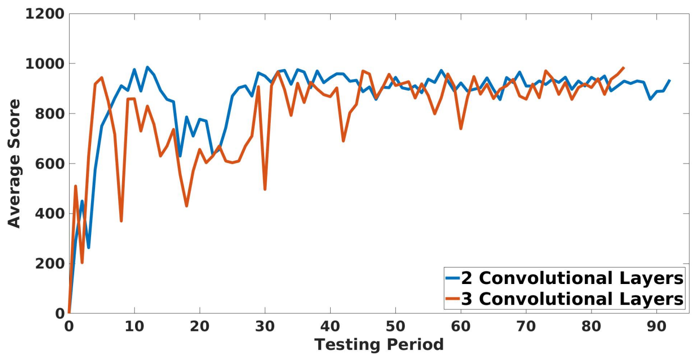


Figure 5.5: Average score over five tests using two convolutional layers (orange) and three layers (blue).

longer to converge, judging on the trend of average Q-values. Therefore, we believe that three convolutional layers are better at identifying important features in the input, thus we decided to continue our investigation using the deeper architecture.

5.3 Action Space

The action space in the car racing game is continuous. But when human plays the game, the only available actions are right turn, left turn, acceleration and deceleration. We started experiments with these four actions. However, we soon realized that human players can also choose not to input any actions (no operation). In order to compare the effect of having four with five actions, we conduct a test once again on the random short track environment.

As shown in Figure 5.7, the model with five actions achieves better average scores over five tests after 120 testing periods. Hence we resolved to use five actions for models.

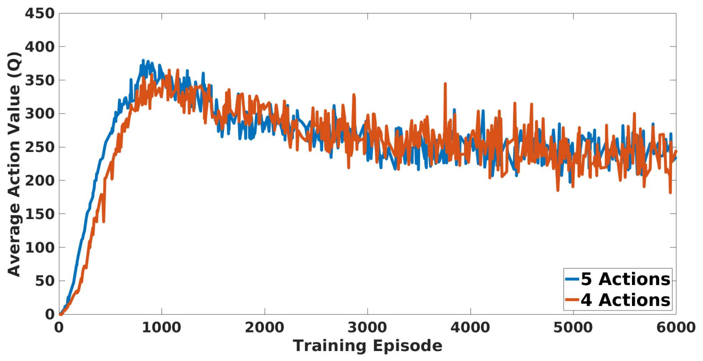


Figure 5.6: Average action values using four (orange) and five (blue) actions. Specific parameters are included in Table B.2

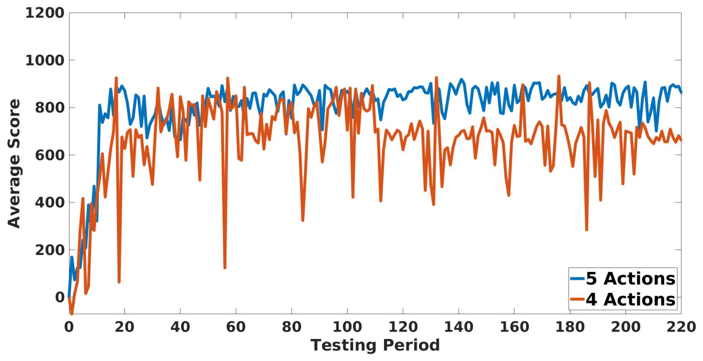


Figure 5.7: Average scores over five tests using four (orange) and five (blue) actions. Specific parameters are included in Table B.2

5.4 Reward

The nature of the reward signal from the environment greatly influences the convergence and performance of RL models. Therefore, we progressed to investigate the effect that rewards have on the performance in the car racing game. As mentioned in section 5.1, in the original game there are two different kinds of rewards the agent can receive. For every tile the car visits, the reward is 1000/N, where N is the total number of tiles on the track, while for every frame passed, the reward is -0.1.

Intuitively, if the reward for visiting tiles is magnified, the agent will tend to visit all tiles at the cost of possibly slowing down. If the negative reward for one frame passed is magnified, the agent will be encouraged to go faster while possibly missing some of the tiles. Therefore, we experimented with changing the ratio and magnitudes of positive and negative rewards.

We trained models with six different pairs of positive and negative rewards. We then tested the trained models on 100 random tracks and compared the average scores and standard deviations. As shown in Figure 5.8, when the positive reward for visiting one tile is fixed to 1, the performance worsens as the magnitude of the negative reward decreases. When the negative reward is fixed to -0.1, the performance varies when the positive reward changes from 20 to 1. We suspect there is a better reward pair, with positive reward between 1 and 10 and negative reward being 0.1.

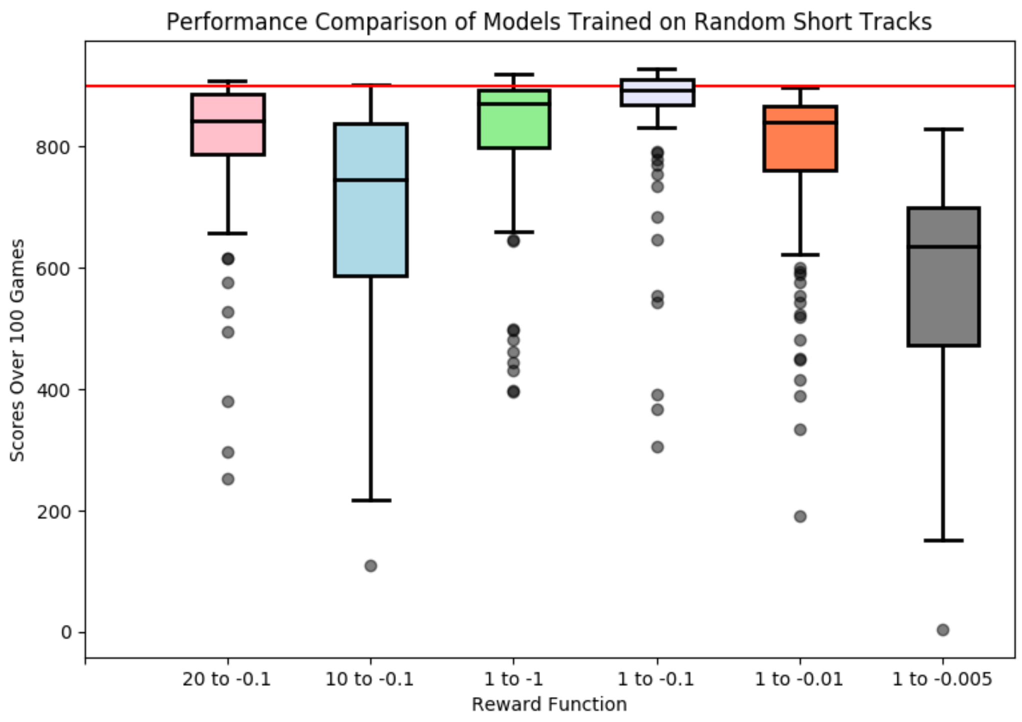


Figure 5.8: Performance of models with different reward ratios trained in the random short track environment.

Chapter 6

Performance Analysis

After tuning the neural network structure and action space, we began to analyze the performance of our models. In section 6.1, we explain the metrics we used for evaluating the model. We establish the baseline results in section 6.2. We compare the models with dropout to the baseline in the rest of this chapter. Note that in section 6.5, we present a model that solved the CarRacing game challenge.

6.1 Performance Metrics

We used two metrics to evaluate the performance of our models. First, we allowed the model to play 100 games and computed the average score. Although the racetracks in these games were random, we used the same 100 racetracks for all tests in order to compare performance between models. We also compute the standard deviation of these 100 scores. This performance metric is based on the OpenAI challenge to achieve an average score of 900 or more over 100 games.

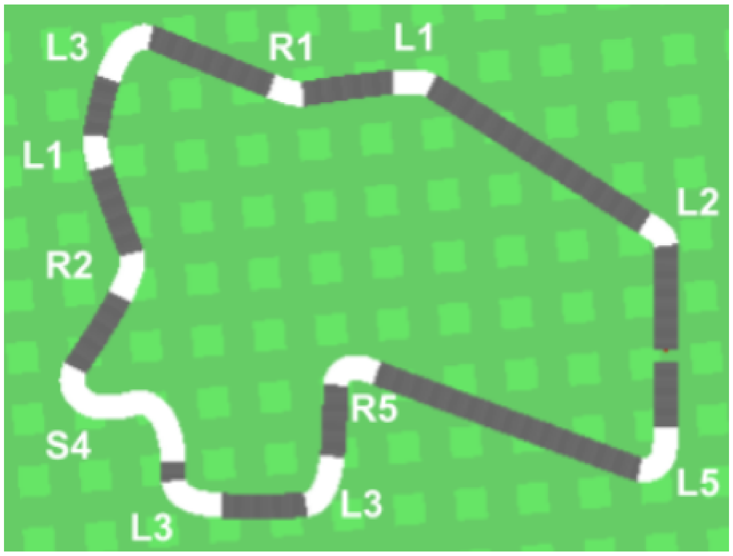


Figure 6.1: An example of curve characterization

Second, we analyzed how the models performed on different curves in a racetrack. We developed a simple curve classification algorithm, which is demonstrated in Figure 6.1. Each curve is characterized as either as either a Left, Right, or S-shaped curve. An S-shaped curve is either a left turn followed by a right turn or a right turn followed by a left turn. Then, the steepness of the curve is ranked on a scale from 1 to 5, where 1 represents a very shallow curve and 5 represents a very steep curve. For example, L1 is a slight left curve and S4 is a fairly steep S-shaped curve. We compute the proportion of tiles that the agent visits for each class of curves. This performance metric is useful for understanding the strengths or weaknesses of a model as well as its ability to generalize to different curves.

6.2 Baseline Results

We trained the algorithm on three environments and performed a baseline experiment. In the first environment, there was 1 predetermined racetrack that the agent interacts with during training. In the second environment, there were 3 predetermined racetracks. In the third environment, the racetracks were generated randomly. The average score and standard deviation of testing these models over 100 random racetracks is shown in Table 6.1.

Model Environment	Average Score	Standard Deviation
1 Track	849.99	78.72
3 Track	853.88	127.71
Random Tracks	854.83	107.14

Table 6.1: Baseline Performance over 100 random games

The standard deviation of these test scores is fairly high because our current models are unable to perform well consistently. While testing these models, we observed that the algorithm struggled to generalize to different racetracks. For example, the model trained on a single fixed track catastrophically failed when it encountered a steep right curve during testing. We believe that this failure due to the fact that the model does not see any examples of steep right curves during training. This is an indicator that our algorithm is overfitting.

6.3 Regularization with Dropout

In order to prevent overfitting, we added dropout as a regularization technique to our neural network architecture by dropping out units at a rate of 30% after the second convolutional layer of the neural network. We applied dropout to each of the three models tested in the baseline experiments in order to compare performance. The results of these experiments are shown in Table 6.2.

Model Environment	Average Score	Standard Deviation
1 Track	894.38	24.5
3 Tracks	875.58	43.58
Random Tracks	887.39	24.65

Table 6.2: Performance for Dropout Rate 30% over 100 random games

When adding dropout to our neural network architecture, the average score over 100 games increased and standard deviation decreased in all three environments. The model achieved higher scores in the game more consistently. Hence, applying a dropout rate of 30% improved the performance of our algorithm and came remarkably close to solving the OpenAI challenge of an average score of 900 over 100 random games.

6.4 Curve Characterization

We used the curve characterization method in order to determine whether applying dropout improved the ability of our algorithm to generalize to curves that were not seen during training. In particular, we analyzed the models that were trained on a single fixed track, because these models only saw a few curves durining training. Even though both models saw the same fixed track during training, the dropout model performed better on each class of steep curves. These results are demonstrated in Figure 6.2. These results indicate that using dropout allows the model to generalize better to curves it did not see during training. Thus, dropout has the potential to be an effective regularizer in deep reinforcement learning problems.

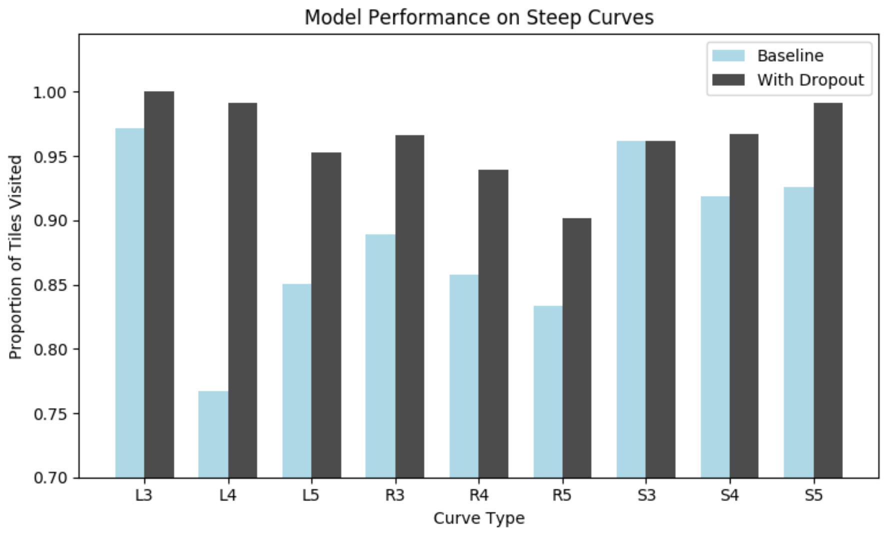


Figure 6.2: Curve Performance with and without Dropout

6.5 Solving Car Racing Game

We then trained the neural network with dropping out units at a rate of 50% to explore if higher dropout rate would improve the performance. The results of the experiments are shown in Table 6.3.

Model Environment	Average Score	Standard Deviation
3 Tracks	906.67	23.6
Random Tracks	892.62	41.48

Table 6.3: Performance for Dropout Rate of 50% over 100 random games

Increasing the dropout rate from 30% to 50% increases the average score in all three environments. Using this method, we are able to achieve a score over 900 in the 3 Track environment, which successfully solves the OpenAI challenge.

Chapter 7

Image Compression

GPUs have become increasingly common in machine learning due to their rapid matrix multiplication. While we had access to GPUs, the on-board memory was not sufficient to store the images in the experience replay memory. Consequently, host-to-device transfer must occur to properly store all images. In an effort to reduce the required memory, we converted the eight-bit grayscale representation of images to binary images, reducing the memory consumption by a factor of eight. The benefits of this compression are twofold: foremost, we reduce the memory required, and the agent learns to follow the road more closely. The latter is discussed further in the next few sections.

7.1 Edge Detection

Upon completing a racetrack, we found that the car would not continue to drive on the road. This suggested the agent learned to follow the tiles on the road; in their absence, the agent did not know to continue on the road to collect the missed tiles. While pursuing tiles is a reasonable strategy for maximizing rewards in the game, this behavior is at least detrimental in seeking missed tiles and at worst adverse in the context of real-life driving. To mend this, we altered the representation of the images fed into the network so that tiles were not visible to the agent. An example of the altered image is given in Figure 7.1.

With this compression method, the agent no longer sees tiles and must learn to stay on the road to receive rewards. The method in question is a form of edge detection known as the Canny edge detection algorithm [3]. The next few sections provide an overview of the Canny algorithm followed by a discussion of the performance of our algorithm with this updated representation.

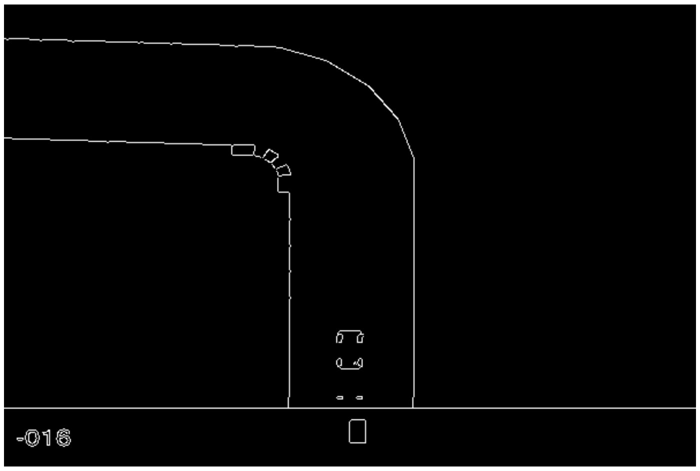


Figure 7.1: Edge detection compression

7.1.1 Canny Edge Detection

The Canny algorithm consists of five main steps:

1. Conversion from RGB to grayscale.
2. Noise suppression.
3. Gradient approximation.
4. Non-maximum edge suppression
5. Hysteresis thresholding

Each step is discussed in detail in the next few sections.

7.1.2 RGB to Grayscale

Several methods exist for converting a red-green-blue (RGB) color image to grayscale, though all reduce to taking a weighted average of each color channel. We used the conversion in Equation (7.1), where R, G, B represent the red, green, and blue color channels (respectively) of the RGB image.

$$Gray = .299R + .587G + .114B \tag{7.1}$$

An example of a grayscale conversion using Equation (7.1) is given in Figure 7.2.

7.1.3 Noise Suppression

Edge detection is typically applied to images captured by digital cameras and inherently exhibit some noise. For our purposes, 'noise' refers to regions in the image that are not

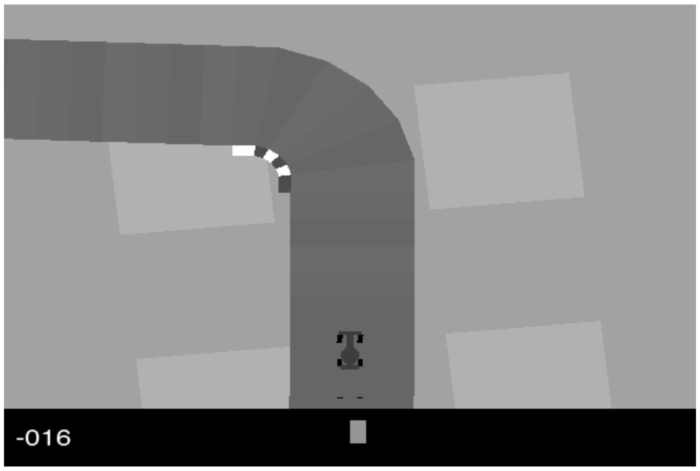


Figure 7.2: Grayscale image representation

crucial in the agent's learning how to navigate the tracks. Such regions are the tiles and grass squares visible in Figure 7.2.

A common technique used to suppress noise is image smoothing where pixel values are replaced by a weighted average of its neighbors. This weighted average computation is typically performed via convolution. Given two continuous functions f, g with bounded support $[0, \infty)$, the convolution of f and g gives a third function and is defined as

$$(f * g)(t) = \int_0^t f(\tau)g(t-\tau)d\tau. \tag{7.2}$$

For discrete functions, the convolution is given by

$$(f * g)(t) = \sum_{\tau=0}^{t} f(\tau)g(t-\tau). \tag{7.3}$$

To be more precise, let f above denote the grayscale image obtained using Equation (7.1). g will then be a two-dimensional matrix, called a kernel, that determines how much weight to give to neighboring pixels. We used the 3×3 Gaussian kernel given below

$$g = \begin{bmatrix} 0.0751 & 0.1238 & 0.0751 \\ 0.1238 & 0.2042 & 0.1238 \\ 0.0751 & 0.1238 & 0.0751 \end{bmatrix}$$

which can be found by taking the matrix below and dividing so that the matrix entries sum to 1; h here refers to the bivariate Gaussian density $\mathcal{N}\left([0,0], \begin{bmatrix} 1 & 00 & 1 \end{bmatrix}\right)$.

$$\begin{bmatrix} h(-1,-1) & h(0,-1) & h(1,-1) \\ h(-1,0) & h(0,0) & h(1,0) \\ h(-1,1) & h(0,1) & h(1,1) \end{bmatrix}.$$

40	40	200	240	100

Table 7.1: 1-dimensional signal

Let h now be the output of convolving the image f with kernel g, then $h(m,n)$ is given by

$$h(m,n) = \sum_{k=-1}^{1} \sum_{l=-1}^{1} f(m-k, n-l)g(k,l) \tag{7.4}$$

where g is indexed such that the top left entry is entry $(-1,-1)$ and its bottom right entry is $(1,1)$. h now represents the smoothed image; this operation is typically referred to as a Gaussian blur.

7.1.4 Gradient approximation

For our purposes, an edge refers to a change in pixel intensity. Change in functions is typically quantified by differentiation; the same applies to images. Having smoothed the image using Equation 7.4, edges are identified by computing this image's gradient. Letting f be a differentiable function, its derivative at x, $f'(x)$, is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{7.5}$$

For a small enough h, $f'(x)$ can be approximated using the forward difference approximation in Equation (7.6).

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{7.6}$$

For images, a more accurate derivative approximation is the central difference defined in Equation (7.7).

$$f'(x) \approx \frac{f(x+.5h) - f(x-.5h)}{h} \tag{7.7}$$

An image is two-dimensional and hence has derivatives along two axes, which we call x and y. Thus, given a signal such as that in Table 7.1–which can be interpreted as the row of an image–it can be differentiated using Equation (7.7).

For example, to estimate the derivative of the signal at the third pixel value of 200, we compute $\frac{240-40}{2} = 100$.

Padding the image with zeros allows us to perform the operation above to every pixel in the signal. Up to scaling by a constant, this operation is computationally equivalent to convolving the signal with the kernel $[-1, 0, 1]$. Thus to obtain the gradient along the

x direction of an image, we simply need to convolve it with the kernel $[-1, 0, 1]$. This operation outputs a grayscale image with edges most prominent along the original image's x direction; however, this also introduces 'false edges'–edges that may not necessarily be edges of interest. To remedy this, we blur the outputted image with a smoothing filter. In particular, we will convolve the gradient approximation with the kernel $[1, 2, 1]^T$. We smooth along the y-axis so that false edges parallel to the x-axis are averaged. Thus, to obtain the gradient along the x-direction of the smoothed image, h, given by Equation (7.4), we compute

$$I_x = h * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 121 \end{bmatrix}. \tag{7.8}$$

The convolution operation is associative, allowing us to first convolve the filters in Equation (7.8) to obtain

$$I_x = h * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{7.9}$$

where the filter on the right of Equation (7.9) is called the Sobel filter. In literature, the filter above may be presented as having the negative values on the rightmost column and the positive values on the leftmost column; we prefer our representation as it illustrates how the kernel arises. Moreover, the sign of the kernel is not imperative as we are interested in magnitudes.

We can similarly obtain the gradient along the y-axis by convolving with the transpose of the Sobel kernel given in Equation (7.9):

$$I_y = h * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}. \tag{7.10}$$

Having obtained the gradients along the x and y axes, the magnitude of the gradient, I, can be obtained by computing

$$I = \sqrt{I_x^2 + I_y^2}. \tag{7.11}$$

Similarly, the direction of the gradient, θ, can be obtained by computing

$$\theta = \arctan\left(\frac{I_y}{I_x}\right). \tag{7.12}$$

At this juncture, I in Equation (7.11) represents the edges of the image, though it contains false edges. The next two sections discuss methods to suppress these edges using θ and hysteresis thresholding.

7.1.5 Non-Maximum Edge Suppression

By smoothing the original grayscale image, we have changed the values of the pixels such that we will ultimately be able to remove them in the final step of the Canny algorithm. At this stage, it is possible that the unwanted edges are still present in I. Indeed, this is corroborated by visualizing I in Figure 7.3 where the tile and grass edges are clearly visible. These will be removed in the next step of the Canny algorithm, but this image reveals another potential issue: the edges are very thick. This step of the edge detection algorithm aims to thin these edges via a method known as non-maximum edge suppression.

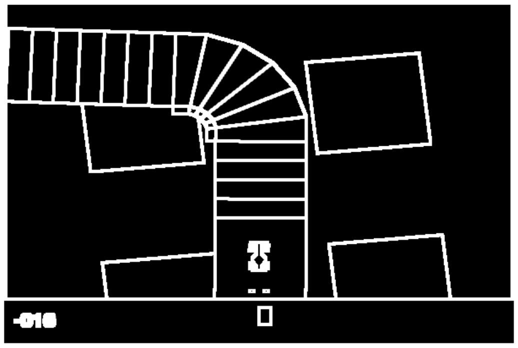
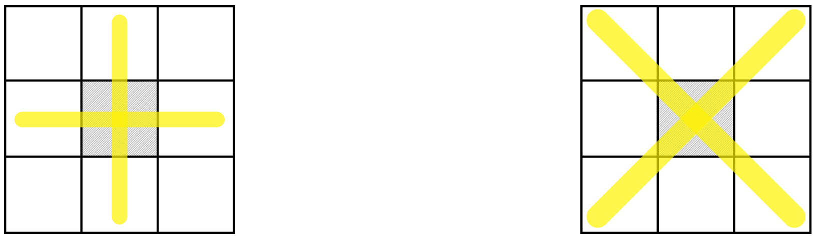


Figure 7.3: Thick edges

This technique goes through every pixel in I and compares the pixel value to the value of its neighbors; if the pixel value is larger than those of its neighbors, it is kept, otherwise it is set to 0. If (i, j) represents the pixel in question, its 'neighbors' are defined to be the pixels parallel to the direction of (i, j) where the direction is given by the gradient direction at (i, j)–that is, $\theta(i, j)$.

Images are discrete objects and each pixel is surrounded by eight potentially neighboring pixels. Thus, the direction of a pixel's gradient must be discretized into eight possible directions. As such, every entry in θ will be binned into eight possible directions: north to south, south to north, east to west, west to east, southwest to northeast, northeast to southwest, northwest to southeast, and southeast to northwest. A diagram illustrating these directions is given in Figure 7.4.

This diagram depicts a clear redundancy: directions can be discretized into four bins. As such, gradient directions will be classified as one of: north to south, west to east, northwest



(a) N to S, W to E (b) NW to SE, SW to NE

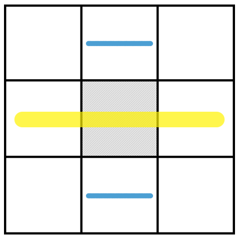
Figure 7.4: Discretized gradient directions

Angle	Bin
(0, 22.5) ∪ (157.5, 202.5) ∪ (337.5, 360)	W to E
(22.5, 67.5) ∪ (202.5, 247.5)	SW to NE
(67.5, 112.5) ∪ (247.5, 292.5)	N to S
(112.5, 157.5) ∪ (292.5, 337.5)	NW to SE

Table 7.2: Gradient direction binning

to southeast, and southwest to northeast. The discretization is performed as shown in Table 7.2, where the left column is the value of θ in degrees and the right column is the corresponding bin; the notation (a, b) denotes the set of numbers in the range from a to b.

Once the gradient directions have been discretized, every pixel in I is compared to its neighbors. The neighbors of a pixel are determined by the direction of its gradient. The neighbors for every gradient direction is given in Figures 7.5 and 7.6.


(a) W to E neighbors

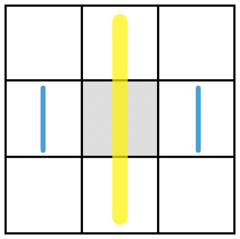
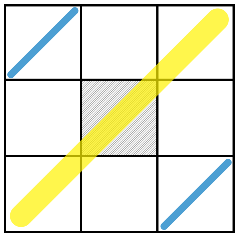

(b) N to S neighbors

Figure 7.5: Discretized gradient directions


(a) SW to NE neighbors

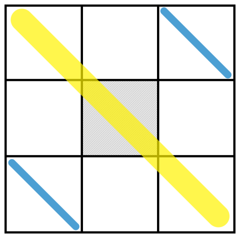

(b) NW to SE neighbors

Figure 7.6: Discretized gradient directions

For example, if pixel (i, j) has a gradient direction of SW to NE as in Figure 7.6a, then the neighbors of (i, j) are pixels $(i-1, j-1)$ and $(i+1, j+1)$. We then compare the value of pixel (i, j) to its neighbors and set it to zero if its value is less than those of its neighbors, otherwise its value remains the same.

After performing the operation outlined above, we obtain an image similar to that in Figure 7.3, though now the edges have been thinned as shown in Figure 7.7.

7.1.6 Hysteresis Thresholding

After performing non-maximum edge suppression via the procedure outlined above, the final step of the Canny edge detection algorithm is to remove edges that are deemed insignificant. More precisely, we will introduce two parameters $a, b \in \{0, 1, 2, \ldots, 255\}$ $a > b$ known as

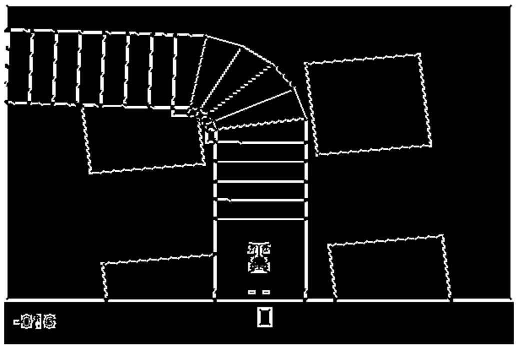


Figure 7.7: Thin edges

thresholds. Then, every pixel in the thinned image will be compared to these thresholds: values less than b will be discarded (set to zero), and values greater than a will be kept (set to one). Values between a and b will be kept only if the pixel is connected to an edge with a pixel value greater than a. More precisely, if the value of pixel (i, j) is between a and b, we will look at its neighbors (as determined by the gradient direction) and set the value of pixel (i, j) to one if one of its neighbors has a value greater than a, otherwise it will be set to zero.

For our algorithm we used the thresholds $a = 150$ and $b = 250$. Applying hysteresis thresholding to Figure 7.7 yields the image in Figure 7.1.

7.1.7 Edge Detection Performance

In writing, feeding the output of the compressed edge detected image offers several benefits: the agent learns to stay on the road and the required memory decreases by a factor of eight. However, it is important to verify these claims. The orange curve in Figure 7.8 plots the average Q-function of our algorithm when the images fed are those passed through the edge detection algorithm; the blue line depicts the average Q-function when the images are simply grayscale images. In particular, both models were trained on three fixed tracks and tested on random ones. Both methods converge to similar values, suggesting the algorithm retains its generalizability. Moreover, the orange curve in Figure 7.9 depicts the average training score for the edge detection algorithm as compared to the grayscale algorithm (in blue); there is no evidence suggesting the performance of the algorithm is compromised.

In short, by first passing the images output by the environment through an edge detection compression, we are able to reduce the amount of memory required for training, and retain the performance obtained without compressing the image.

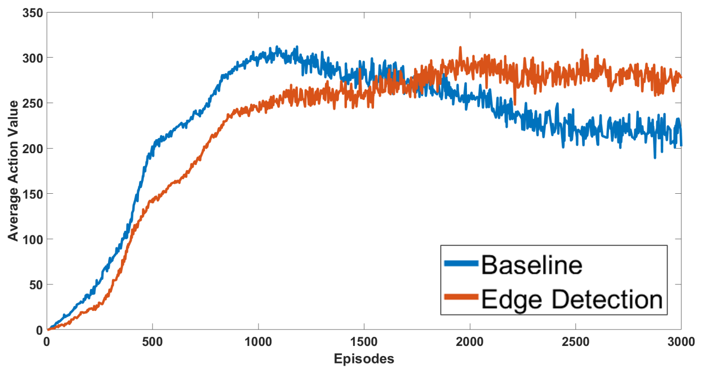


Figure 7.8: Average Q-value for edge detection

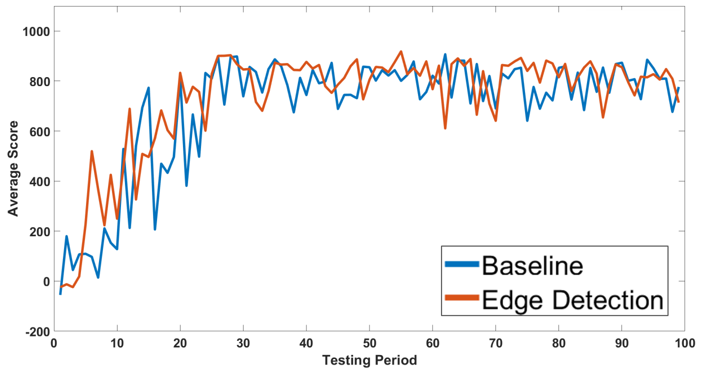


Figure 7.9: Average training score for edge detection

7.2 Grass Detection

A common behavior exhibited by the agent is cutting corners as shown in Figure 7.10.

Moreover, as commonly seen in professional race car driving, the car lingers along the right edge of the track to easily make left turns. Consequently, when presented with a right turn, the agent has trouble making it and often misses tiles. Both of these issues can be resolved if the agent learns to stay out of the grass. While it is possible that over time the agent will learn to stay out of the grass, the long training time is undesirable. As such, we implemented a form of grass detection so that the agent receives a negative reward while training to learn to avoid the grass. The grass detection implementation is very simple: we simply crop a 13×11 rectangle around the car and count the number of green pixels. If the number of green pixels is larger than 44, then we say the car is in the grass.

As shown in Figure 7.11, the average Q-value for edge detection and edge with grass detection converge to the same values. More importantly, as shown in Figure 7.12, the training score when using grass detection increases as compared to using edge detection without grass detection. While both methods are able to exceed score of 900, adding grass detection allows the agent to exceed the score more often and reduces the number of times the car cuts corners.

Our grass detection algorithm simply alters the reward function while training: it returns a score of -1 every time the car enters the grass. In the context of real-world autonomous driving, grass detection can be interpreted as the auxiliary devices of the car. For example, the reward given to the car while training can vary based on the distance between the car and the sidewalk calculated with a radar. Ultimately, incorporating grass detection into our

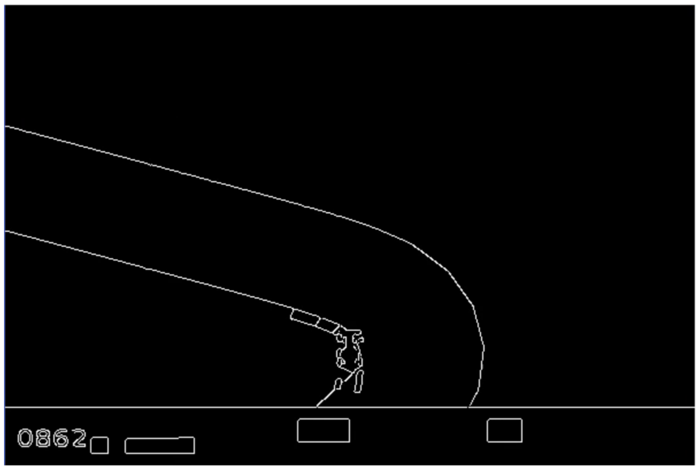


Figure 7.10: Agent cuts corners

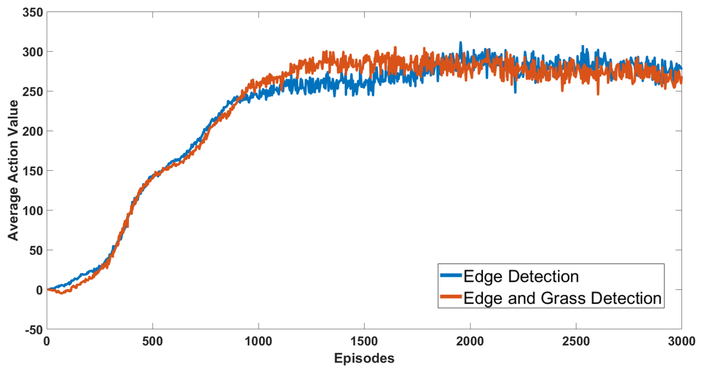


Figure 7.11: Average Q-value for grass detection

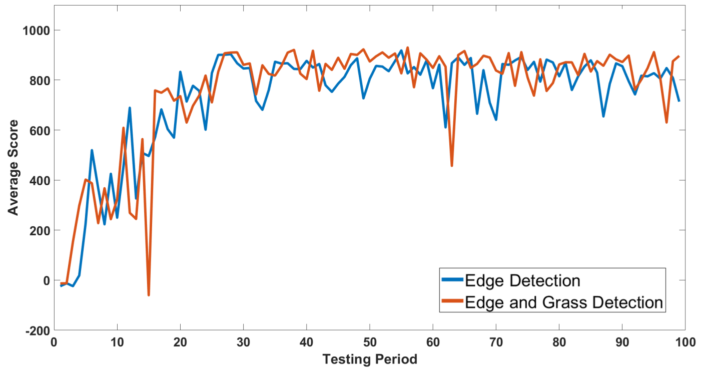


Figure 7.12: Average training score for grass detection

algorithm while training, and observing its performance on random tracks, suggests that such auxiliary data does not compromise the generalizability of our algorithm and improves its performance.

Chapter 8

Discussion

Our work has shown that the DQN algorithm does generalize from limited training environments to all tracks in the OpenAi CarRacing environment, but not perfectly. The algorithm fails occasionally during testing, such as cutting corners and swirling out of the road. There is still much work to do to uncover the reasons for these problems. We have compiled a list of ideas that could improve the performance on this specific task, with some of them also applicable to other reinforcement learning problems:

- Explore hyperparameters more thoroughly, most importantly replay memory size, batch size and update frequency for target Q-network
- Explore multiple regularization methods and their hyperparameters
- Try Deeper architectures
- Introduce perturbations into the testing environment and analyze the effect

Nonetheless, we have surpassed the current score for the car racing game on the OpenAI leaderboard and come very close to solving the game. Our results involving dropout also shone some light on the positive effect regularization can have on Deep Reinforcement Learning algorithms.

Appendix A

Algorithms Outline

A.1 Basic Q-Learning Algorithm Outline

```
Initialize $Q(s, a)$ for all $s \in S, \quad a \in A$
for each episode do
    Get initial state $s_t$
    while episode is not over do
        Sample $U \sim Unif(0, 1)$
        if $U < \epsilon$ then
            $a_t$ = random action
        else
            $a_t = \arg\max_{a'} Q(s_t, a')$
        end
        Take action $a_t$, get reward $R_{t+1}$ and new state $s_{t+1}$
        Set $Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left( R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$
        $s_t = s_{t+1}$
    end
end
```

Algorithm 1: Basic Q-learning

A.2 Deep Q-Learning Algorithm Outline

```
Initialise $Q$ with random weights $\theta$
Set $\theta^- = \theta$.
for each episode do
    Get initial state $s_0$
    for $t$ = 1, 2, 3 ... $T$ do
        Sample $U \sim Unif(0,1)$
        Set $a_t = \begin{cases} \text{random action sampled uniformly} & \text{if U} < \epsilon \\ \arg\max_{a'} Q(s_t, a') & \text{otherwise} \end{cases}$
        Take action $a_t$, get reward $R_{t+1}$ and new state $s_{t+1}$
        Store $(s_t, a_t, R_{t+1}, s_{t+1})$ in the replay memory $\mathcal{D}$
        Sample a random minibatch of transitions $(s_j, a_j, R_{j+1}, s_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} R_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-) & \text{if } s_{j+1} \text{ is not terminal} \\ R_{j+1}, & \text{otherwise} \end{cases}$
        Perform a gradient descent step on the loss $(y_j - Q(s_j, a_j; \theta))^2$
        Every C steps set $\theta^- = \theta$.
    end
end
```

Algorithm 2: Deep Q-learning

Appendix B

Parameters

B.1 Neural Network Structures

Layer Number	Two Convolutional Layers	Three Convolutional Layers
Layer 1	Convolutional, 16 8x8 kernels with stride 4, ReLU activiation	Convolutional, 32 8x8 kernels with stride 4, ReLU activiation
Layer 2	Convolutional, 32 4x4 kernels with stride 2, ReLU activiation	Convolutional, 64 4x4 kernels with stride 2, ReLU activiation
Layer 3	Dense, 256 kernels, ReLU activiation	Convolutional, 64 3x3 kernels with stride 1, ReLU activiation
Layer 4	Dense, 5 kernels, linear activation	Dense, 512 kernels, ReLU activation
Layer 5		Dense, 5 kernels, linear activation

Table B.1: Neural Network Structures

B.2 Actions

Four Actions	Five Actions
Left Turn [-1, 0, 0]	Left Turn [-1, 0, 0]
Right Turn [1, 0, 0]	Right Turn [1, 0, 0]
Acceleration [0, 1, 0]	Acceleration [0, 1, 0]
Deceleration [0, 0, 0.8]	Deceleration [0, 0, 0.8]
	No Operation [0, 0, 0]

Table B.2: Action Space Comparison

B.3 Hyperparameters

Name	Value
Architecture	Three convolutional layers
Explore Rate	Decrease from 1 to 0.1 linearly for the first 250000 frames, 0.1 thereafter
Learning Rate	0.00025
Discount Rate	0.99
Batch Size	32
Memory Capacity	100000

Table B.3: Hyperparameters

Appendix C

Glossary

Agent:	Decision maker
Environment:	World the agent interacts with
State:	Snapshot of the environment e.g. a screenshot of the game (denoted as S)
Action:	Actions the agent can take, e.g. accelerate or turn right (denoted as A)
Reward:	Scalar feedback from environment to agent (denoted as R)
Return:	Sum of all rewards the agent receives (denoted as $\mathcal{R}$)
Q-Learning:	A reinforcement learning technique used in machine learning.
Dropout:	A common regularization technique.

Appendix D

Abbreviations

AI. Artificial Intelligence.

DDQN. Double Deep Q-Network.

DQN. Deep Q-Network.

IPAM. Institute for Pure and Applied Mathematics. An institute of the National Science Foundation, located at UCLA.

MDP. Markov Decision Process.

RGB (image). Red, Green and Blue (image).

RIPS. Research in Industrial Projects for Students. A regular summer program at IPAM, in which teams of undergraduate (or fresh graduate) students participate in sponsored team research projects.

RL. Reinforcement Learning.

RLP. Reinforcement Learning Problem.

SGD. Stochastic Gradient Descent.

TD. Temporal Difference.

UCLA. The University of California at Los Angeles.

References

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, *A brief survey of deep reinforcement learning*, arXiv preprint arXiv:1708.05866, (2017).

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, arXiv preprint arXiv:1606.01540, (2016).

[3] J. Canny, *A computational approach to edge detection*, IEEE, PAMI-8 no. 6 (1986), pp. 679 – 698.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, vol. 1, MIT press Cambridge, 2016.

[5] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, arXiv preprint arXiv:1502.03167, (2015).

[6] G. Kahn, A. Villaflor, B. Ding, P. Abbeel, and S. Levine, *Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation*, arXiv preprint arXiv:1709.10489, (2017).

[7] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman, *Teacher-student curriculum learning*, arXiv preprint arXiv:1707.00183, (2017).

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, arXiv preprint arXiv:1312.5602, (2013).

[9] 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, 529 (2016), pp. 484–503.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting*, Journal of Machine Learning Research, 15 (2014), pp. 1929–1958.

[11] R. S. Sutton, A. G. Barto, et al., *Reinforcement learning: An introduction*, MIT press, 1998.

[12] C. Szepesvári, *Algorithms for reinforcement learning*, Synthesis lectures on artificial intelligence and machine learning, 4 (2010), pp. 1–103.

[13] Y. Tian, Q. Gong, W. Shang, Y. Wu, and C. L. Zitnick, *Elf: An extensive, lightweight and flexible research platform for real-time strategy games*, in Advances in Neural Information Processing Systems, 2017, pp. 2656–2666.