The MS Committee for Elise McEllhiney certifies
that this is the approved version of the following project report:

Self-Training Autonomous Driving System Using An Advantage-Actor-Critic Model

October 26, 2023

Committee:

_____

Chairperson

_____

Committee Member

_____

Committee Member

_____

Date Approved

Abstract

We describe an autonomous driving system that uses reinforcement learning to train a car to drive without the need for collecting training input from human drivers. We achieve this by using the Advantage Actor Critic reinforcement system that trains the car based on continuously adapting the model to minimize the penalty received by the car. This penalty is determined if the car intersected the borders of the track on which it is driving. We show the resilience of the proposed autonomously trained system to noisy sensor inputs and variations in the shape of the track.

Contents

List of Figures

Chapter 1

Introduction and Motivation

Neural Networks can be used in autonomous driving systems for vehicles. In this paper we present an autonomous driving system utilizing an Advantage Actor Critic Model [1].

Neural networks are useful in their ability to identify and utilize features of inputs that are efficient to calculate and provide accurate predictions, but their inner workings are not easily interpreted by human designers. This leads to a level of obfuscation in neural network models that makes it difficult to ascertain whether or not the network is successful in providing desired and useful predictions consistently enough to be comparable with human driving capabilities. A drawback to neural networks is that they are only as accurate as the training data that they are provided. In this work we explore the training and limitations of Neural Networks in simulated autonomous driving systems.

Simulated driving environments are useful in their ability to provide a self contained, consistent environment which we can leverage to test individual elements of a system at a time. For simulated driving environments it is simple to retrieve feedback metrics such as how far a simulated car is from the center of a lane or

if the car has collided with a simulated object. Using a simulated driving environment the interactions between simple sensors and autonomous driving systems are studied here.

A strategy for training neural networks in real world settings is to have them mimic human behavior. This is done by recording sensor inputs while a person is directly controlling the car. The sensor data is recorded in real time alongside the human controls and used to train a supervised neural network. That is, with the sensor data being used as the input to the neural network and the human action associated with those inputs being the ideal action. The model is trained in order to find a mapping that outputs the most likely human action given the sensor inputs.

This approach allows neural networks to perform similarly to a human user so long as it has sufficient training data over a wide enough variety of possible driving scenarios. This also eliminates the need for the system to determine whether or not it is in a good state since it assumes that the actions it takes should be the same actions that a human would have taken. The base assumption here is that human action is the ideal action in a given situation. This modeling strategy also assumes that the vehicle's sensors will never be outside of the range of inputs covered by its human generated training data. Additionally, this approach offloads the responsibility of finding the optimal path to a human who must train the network with a large number of good driving samples. This approach limits a neural networks best possible actions to the capability of the human that trained the network.

Driving does not have a well defined ideal behavior. Determining when neural network is in an optimum state relative to all the possible decisions that could

be made is more difficult than determining situations in which the car is in a bad state. A car that has crashed or is too close to objects, has clearly entered a bad state and should seek to avoid that state in the future if possible.

This paper proposes an extension of neural network models that can be trained on inputs and associated human actions and extend it to a reinforcement style of learning. This enables the system to train continuously and more readily adapt to new inputs that the network has not previously encountered.

Reinforcement learning allows the network to train with only a reward reflecting whether or not the system is in a good state. We describe a system that uses reinforcement learning with negative feedback and train itself to drive without ever requiring a human to drive the car.

Chapter 2

Background and Related Work

There are numerous way to tackle the problem of learning using neural networks. These include a variety of supervised learning techniques that involve taking known sensor input data paired with associated actions. They also include reinforcement learning techniques. Reinforcement learning takes sensor inputs and chooses actions to learn the resulting rewards. The model then uses feedback from the results of the previous actions to weight the model so that future actions better optimize the rewards. The goal of this paper is to develop a system using a continuously training neural network capable of autonomous driving.

One challenge with the task of autonomous driving using a sensor input is that the car is usually operating within a continuous space. Sequential inputs are correlated due to the time dependent nature of driving a car. Another consideration for driving models is that rewards are potentially time delayed. In other words, an action now may not have any immediate reward or loss, but may contribute to a good or bad state in the future. Driving is also a task where there are many acceptable sequences of actions and it is the culmination of a series of independent choices that leads to either a final result of success or failure. Any sequence of

states in which the car has not crashed is equally acceptable to any other non-crashed state. Additionally, measurements from sensors on cars are corrupted by noise and can be missing, so the model needs to be robust against noisy or lost sensor data.

2.1 Progression of the Autonomous Driving Algorithm

In our development of an autonomous driving system we started with the idea of creating a software model that was capable of driving well with minimal human involvement. To achieve this we defined driving well as being able to continuously move forward without crashing into barriers. For minimizing human intervention we started with a structure where all the sensor interpretation logic and all the driving logic was interpreted by a human and coded manually. This consisted of coding the interpretation of sensor inputs and creating mappings to desired outputs. From this starting point we moved forward by trying to reduce the amount of human intervention required to create a model. To this goal, we eventually chose to use an advantage actor critic model that could take in the interpreted readings from sensors and, through training, back-propagate the results of driving attempts to refine the models ability to output steering decisions in a way that doesn't result in the vehicle crashing. By the end, the only human intervention was the definition of the model's shape and it's update function. The shape and update functions of the models are explained below.

2.1.1 Manual Modeling

The first and most labor intensive way of creating an autonomous driving system is to manually code all the logic of the system. This includes writing logic

that addresses noisy sensors. This sensor data is then formatted into something usable by manually coded logic that takes those inputs and selects driving actions such as moving forward, turning left or turning right. We fully implemented a simple system using an IR sensor and a small physical robot nicknamed "roachbot" as mentioned in section 3.1.

There were major drawbacks to this method of creating an autonomous driving system. The model is static and only handles what has been considered by whoever has coded the model. Any considerations towards sensor noise must be coded manually. Driving logic is also limited by the amount of work the developer had put into developing the model. All edge cases must be considered and addressed. Any new ability in the model increases the complexity of the software. Our implementation showed us that there was always another minor change to add to the model to address additional edge cases. During indoor testing, the reflectivity of the wall and lighting conditions needed to be addressed by the sensor logic. While all sensor data was noisy, measuring the distance to brick walls was particularly difficult because of the large measurement variance. A large variance required more processing and refinement than was covered by the basic averaging methods we were using to smooth the noise inherent in the sensor data. Instead of measurements of distances to the nearest obstacle, we were getting what appeared to be random numbers. This predictably did not work well with the logic that chose which direction to turn based on these inputs. The result was that the car drove erratically. We also found that different floor textures or the amount of charge in the cars battery had to be addressed by the driving logic. The texture of the floor altered how much of the momentum of the vehicle affected the cars ability to turn and how fast the car was able to accelerate. The charge of the

battery also changed how fast the car accelerated and how forcefully decisions were carried out. These factors together made it difficult to determine how the car would execute a given instruction. This lack of predictability made both the robot and sensor an unappealing option for implementing more complex models.

2.1.2 Supervised Learning Using Expected Actions

We wanted a method to address the variables associated with driving without having to manually code all the logic pertaining to every situation that the car might encounter. The primary variables we wanted to focus on were the positions a car might find itself on a track and how the car should react to any given position. We also hoped that a more sophisticated model might be able to handle some of the unpredictability inherent in physical hardware and its interactions with the environment such as varying rates of acceleration. Our next attempt was to use recordings of human driving to train a model as mentioned in section 3.2. This led us to trying to create a car that used supervised learning models.

An autonomous vehicle "ALVINN" [2] used supervised learning, and mimicked human driving by recording state-action pairs. Some of the earliest successful systems using machine learning for autonomous driving were supervised systems that learned to mimic human driving by recording state-action pairs where the state is a record of the current sensor data and the action is the action a human took for that state.

With current advancements in technology, small embedded systems are capable of this style of autonomous driving. This style of autonomous driving using machine learning can be effective, it is entirely dependent on the quality of the training data provided during the training phase. A negative quality of these

systems is that each frame of the camera input is treated entirely independently from any previous or sequential frames. It does not learn to adapt to transitions from one frame to the next. It takes an individual input frame and determines an ideal output frame, regardless of any previous or future conditions of the systems. Supervised learning produces a static model and is not usually capable of continuously improving the driving model unless provided additional labeled training data.

We found that we still had to have a human intervention for every potential edge case in the form of someone having to generate training data for every possible state in which the car might find itself.

2.1.3 Random Actions To Discover Resultant Rewards

In order to reduce the amount of human intervention required to train the model, we turned to reinforcement learning techniques that would allow the car to learn by driving such as those explored in "Epsilon-first policies for budget-limited multi-armed bandits" [3].

A simple strategy to transition from a supervised learning environment to basic reinforcement learning using neural networks is to take in inputs, output a random action for each input, and record the reward it receives. These are recorded in groupings called episodes and the network picks the episodes that result in the greatest cumulative rewards. The episodes with the highest cumulative rewards are treated as if they are the ideal actions. The neural network model is then updated with those episodes. This system is highly dependent on the chance that a random sequence of actions will eventually see some kind of successful results since the algorithm must have some level of success using nothing but random

actions. A drawback is that rewards are calculated at the end of episodes. This requires that random actions eventually land on a successful sequence of events. This method does not lend itself well towards accounting for increasing numbers of variables.

The concept of supervised learning using human inputs as training data is very similar to this reinforcement tactic. Human actions on given inputs are considered the ideal actions for those inputs and the network is trained using those input/action pairs. This strategy simply takes actions that it determines to be most similar to the human actions that it observed within its training data.

Further updates to the network using backpropogation are implemented by following the same strategy, but instead of truly random inputs, the network uses an epsilon-greedy strategy to pick either the best action according to its existing model. "Playing Atari with Deep Reinforcement Learning" [4] gives an overview of this strategy and its drawbacks. The episodes with the highest cumulative rewards are yet again picked out and used as training data for another update.

The drawbacks to this strategy are substantial, however. Mainly, this style of learning has trouble dealing with delayed rewards since a future reward may be outside of the episode duration. It also takes a long time for it to accumulate enough experience to be able to meaningfully react to all situations it may encounter.

Due to the drawbacks of this strategy, we opted not to create a driving model using this method. We did not expect to be able to execute enough simulations to generate a feasible driving model.

2.2 Epsilon-greedy Action Selection

What we needed was a better method for selecting actions that the car would take while it was learning to drive. Learning to drive requires that we take actions in order to determine their associated rewards. We need to explore new actions to determine if they are better than actions that we have already tried. We also need to exploit known actions so that we can drive far enough along the track to discover new situations. The delayed nature of rewards and penalties in the sequential actions required for driving makes action selection while training a necessity of any reinforcement model that we create.

Learning to drive can be considered a multi-armed bandit problem as first proposed by Robbins in 1952 [5]. The multi-armed bandit problem considers the distribution of unknown rewards or different actions and balances exploration of actions with unknown results to discover if unknown actions offer increased rewards as compared to known actions and the exploitation of actions it has previously explored. The goal of the multi-armed bandit problem solutions is to maximize the total rewards over the entire sequence of actions it selects. This can be achieved in a variety of ways [6] but simple heuristics can be shown to be effective for practical applications [7].

If we simplify actions that a car can take as being forward, left, and right, we can simplify each state to a three-armed bandit problem in which it must take the existing knowledge of penalties and reward and select one of the three actions. The current model gives the predictions for expected values of the potential actions and any action taken is fed back into the model. Since each state can be considered a completely separate bandit problem, we do not try to fully optimize the exploration versus exploitation trade-off, but instead we choose to

value simplicity of the method while maintaining some level of balance between the two.

Epsilon-greedy approaches are some of the simplest approaches to this balance. The epsilon greedy approach takes the best known action with a probability of $1 - \epsilon$ and it selects a non-optimal action with probability ϵ. We can select the value of ϵ to increase or decrease the amount of exploration while training the model.

2.2.1 Q-Learning and Deep Q Learning

While epsilon-greedy action selection addresses our concerns about exploration and exploitation, we still have yet to establish how we know how good an action is compared to another or how to update our model so that it learns from experience. This is where the Q function comes into play. We use the Q function to establish a model that can learn from experience.

Q learning is an advancement in reinforcement learning that allowed for the model to have a way to learn sequential steps. This allows future rewards to impact the systems decisions [8].

The basic concept of Q learning is to pick the action of a set of actions that leads to the highest cumulative reward by the end of a episode. An episode is defined as the states and actions taken before a terminating condition. This cumulative reward is called a Q value. If you have a function that can take in a state and action and output the Q value, then it becomes very simple to select actions that will lead to the highest reward. By selecting actions with the highest Q values, you achieve the optimal set of actions.

$$\pi(s) = argmax_a Q(s, a)$$

$$\pi : predicted\ reward$$

$$s : state$$

$$a : action$$

In other words, this Q function can be used iterate through every action in a discrete set of possible actions from the current state and choose the action with the highest Q value. This action will be the action that maximizes the reward received.

However, this is easier said than done since finding the maximum possible reward for actions would require that you know the entire state map of an environment. If there are a discrete set of states and a discrete set of actions per state, then it is possible to use the basic Q learning update.

$$Q(s, a) = r + \gamma argmax_{a'} Q(s', a')$$

$$\gamma : discountfactor$$

$$a' : nextaction$$

$$s' : nextstate$$

Where $Q(s, a) = r$ if s is a terminal state and γ is the discount factor. A terminal state is a state from which no more actions are taken. The discount factor γ is used so the network values immediate rewards above future rewards. While not strictly necessary, it is generally favored to have a network select the

shortest path to a reward over a longer path to an equal reward.

If $\gamma = 1$ then the Q value of an action is equal to the maximum cumulative reward, if $\gamma = 0$ then the Q value is equal to the reward directly associated with that action and no consideration is given to any future state.

Q learning is a substantial improvement over randomly generating training data since it is able to act on delayed rewards. This allows it to be able to execute sequential actions maximizing immediate and future rewards. Q learning still has some inconvenient limitations, however.

The first issue is that it requires a complete state mapping of the environment. This inherently requires that there be a discrete set of states. It also requires a discrete set of potential actions for each state. As such, continuous states or actions are beyond the capabilities of basic Q learning.

Algorithm 1 Deep Q Learning Algorithm

```
procedure DEEP Q LEARNING UPDATE
    initialize network with random weights
    s ← initial state input
    while e < max episodes do
        if ε < random number between 0 and 1 then
            a ← rand(action)
        else
            a ← argmax_a Q(s, a)
        t ← Q(s, action) for any action not selected
        execute action a
        r ← reward received for action
        s' ← new state
        t ← r + γ argmax_a' Q(s', a')
        use (t − Q(s', a'))^2 as loss to update network
        s ← s'
        e ← e + 1
```

Methods used in and with deep Q learning as shown in "Playing Atari with Deep Reinforcement Learning" [4] help reduce the need to create a complete state

mapping of an environment. Instead of creating an entire state map of the environment, it updates a neural network to input a state and output predicted Q values of each action. The Deep Q Learning Algorithm (see Algorithm 1) shows the update cycle of a Deep Q Learning network. The network is updated using the target value that gets back-propagated in time as the algorithm explores new options.

Both Q and Deep Q learning suffer from the fact that future rewards can only back-propagate one time-step at a time. This style of learning also has a tendency to settle into local maxima since it does take so long for future rewards to back-propagate through the network.

The tendency to settle into a local maximum can be combated using experience replay as shown in its success at playing Atari games [4]. Experience replay consists of recording frames of $[s, a, r, s']$ groupings where $s = state$, $a = action$, $r = reward$, $s' = nextState$. These small sequences are used for updating the network using a set from this record to update the network. Experience replay is useful since it reduces the impact of time dependent and closely sequences of actions such as sequential frames in Atari games. However, it requires a large amount of memory to store these frames and a larger number of frames has to be processed per update for this strategy to work.

2.2.2 Advantage Actor Critic

One of the most recent advancements in machine learning has been the Actor-Critic model and much research has been done to verify its usefulness. Most machine learning models can be categorize into either actor models or critic models as shown in "Actor-critic Algorithms" [1].

We still use epsilon-greedy for action selection, and Q learning for our update function. Advantage actor critic algorithms address how to interpret rewards. Instead of using Q learning exclusively to establish a state-mapping of the scenario, we use a model that predicts both the best action and the anticipated reward of that action. We create two models that update via the Q function and use one to predict the best action and the other to predict the anticipated rewards of that action. We use the concept of advantages to allow judgment of actions to be relative to other actions instead of an absolute predicted reward.

Actor-only models use input frames paired with known good values to train the model to perform actions on input frames in a way that mimics the known good values as closely as possible. Actions are directly estimated from the input state. These models do not account for sequential actions and estimate each frame independently. These models also cannot "learn" based on a reward as feedback. The above-listed supervised learning model is an example of an actor-only model.

Q-learning on the other hand, is a critic-only model. Critic-only models create a value function that takes an input state and a possible action and translates those two parameters into an estimated reward value. These models are likely to find at least a local maxima for the reward calculation it is operating on. However, convergence of the model may be inconsistent due to its tendency to get caught in local maxima. Critic-only models do have the advantage that they can learn based on a reward value as feedback.

Actor-Critic models incorporate advantages from both actor models and critic models. The critic uses the state of the system to learn to predict values and the actor uses to update its policy.

As shown in the "Advantage Actor Critic Algorithm in Asynchronous Methods

Algorithm 2 Advantage Actor Critic Algorithm

```
1: procedure ACTOR CRITIC LEARNING UPDATE
2:     initialize network with random weights
3:     s ← initial state input
4:     a ← actor.predict(s)
5:     execute a
6:     while e < max episodes do
7:         r ← reward
8:         s' ← new state
9:         v ← critic.predict(s)
10:        v' ← critic.predict(s')
11:        if terminal state then
12:            advantages[action] ← r − v
13:            target ← reward
14:        else
15:            advantages[action] ← r + γv' − v
16:            target ← reward + γv'
17:        fit actor to advantages
18:        fit critic to target
19:        a ← stochastically select π(a)
20:        execute a
21:        s ← s'
```

for Deep Reinforcement Learning" [9], the actor-critic updates the actor network based on the value predicted by the critic network. The critic network then updates to improve the accuracy of its value function to reflect the real world rewards. This style of network has been shown to have favorable convergence properties. It is also capable of producing a performant network with small updates every frame and only two frames worth of memory in storage.

This model and the logic outlined in figure 3.1, given its minimal memory requirements, capability of learning sequential steps, and its ability to act on continuous inputs could prove to be an effective model to implement on small embedded systems for driving. In this paper, we will be evaluating the efficacy of the Advantage-Actor-Critic algorithm (see Algorithm 2) in a simulated driving

environment.

Chapter 3

Analysis of Autonomous Driving Models

3.1 Manual Model on a Physical Car

Our research began by exploring what would be required to implement an autonomous driving system on a minimal embedded system. This intent led to a few exploratory implementations on embedded systems and small physical car implementations. This sequence of experiments provided the background for the difficulties associated with the driving problem and how a variety of models and environments could be used to establish and test systems for autonomous driving.

Our first and simplest implementation of an autonomous car was comprised of a two wheel robot car frame, an Intel Up [10], and an Intel RealSense R200 IR camera. The R200 RealSense camera [11] was used for the depth-map produced by the IR camera. This camera detects depth by emitting infra-red signals that then bounce off of an object and the reflected signal is processed into a matrix of distance measurements called a depth-map.

This autonomous car implementation made use of a manually designed model that made decisions based on the depth-map sensor input. The first attempt at this manual model scanned across all the points in the depth-map. If a point's measured distance was under a set threshold, then the car was programmed to turn to the right until it no longer detected points that measured under the set threshold. If it did not detect any points that were under the threshold, it drove forward.

This naïve implementation perpetually turned right and never moved forward. Upon inspection we discovered that the depth-map we were generated via the IR camera was unexpectedly noisy. A large number points within the depth-map were erroneously minimum or maximum values regardless of the actual distance of objects from the sensor.

To mitigate this impairment, the depth-map was interpreted as three even columns and the depth of the column was averaged. We then implemented a very simple ruleset based on only the resultant three average depths. If all the average distances were sufficiently far, it would drive forward. If all the average distances were too close, then it would turn right. If the furthest average distance was the center column, the car would drive forward. If the furthest average distance was on the right, the car would turn left and vice versa for the left being the column detected to have the furthest average distance.

This algorithm was more effective than the first implementation since it accounted for sensor noise, but it still had some drawbacks. Primarily the material of the objects and the lighting of the room notably impacted the ability of the IR sensor to create an accurate depth-map. Even reducing the depth-map to three average distances was not enough to overcome this impairment. Brick and glass

were some of the most difficult materials to handle while painted walls caused the least impairment. Another issue was the algorithms difficulty in handling concave shapes and corners since the algorithm would detect the edge of the corner as the furthest average distance and get stuck in a limbo of turning back and forth into a corner it was facing.

The difficulties in programming around edge cases such as convex shapes and differences in the sensors ability to detect different materials justified the need to use machine learning algorithms to try to generate a system general enough to account for the numerous potential edge cases.

3.2 Supervised Model on a Physical Car

The next experimental car system that we built included a supervised model used to control a physical car based on the Donkeycar implementation [12]. There were some changes made between the first experimental car and this new implementation. The IR camera depth map was noisy and difficult to work with in the first implementation. In response, this new implementation used a camera input to explore another sensor option. Having changed the sensor input, the Intel Up board was no longer required. The Intel Up was selected for its USB 3.0 OTG port and its compatibility with the R200 IR camera. The Raspberry Pi 3 and the Raspberry Pi 3 camera were used in this implementation. This simplified the system since resources and information about the Raspberry Pi 3 and its associated camera are readily available. For steering, the car a servo was used that set a steering angle rather than varying the speed of the wheels. This allowed the car to move a consistent speed forward.

The autonomous driving model was created using a supervised learning train-

ing strategy on a convolutional neural network. The test track used in the experiment was delineated by lines of masking tape. The test track was laid out so that all turns of the track were no sharper than the cars turning radius. The color of the tape was selected by determining which color a masking tape provided the maximum contrast with respect to the floor color. The training data was collected by recording video and driving actions frame by frame as a human drove around a track. A video frame at a fixed rate of frames per second as determined by the camera was recorded and stored paired with the action that the human driver took on that frame.

The convolutional neural network was then trained with the frame of the video as the sensor input and the action taken by the human driver as the desired output.

Training sessions were first done on a short track approximately four meters long with a single turn. The training frames were recorded by having a human driver start at the beginning of the track and driving the car up to speed. The recording was then started and the car was driven along the entirety of the track. The recording was stopped as the car was driven to the end of the short track. This was repeated approximately twenty times to acquire additional training frames. If the car continued to drive erratically, more training data was acquired using the same method.

Trained in this fashion, the car had a tendency to overfit to the training data. Since the human driver would start in approximately same place every time, drive down the middle of the track until the track ran out, the training data was nearly all from the center of the track. When the model was trained on this set of data, it would be very sensitive to starting position. Any deviations from the center of the track quickly resulted in the car driving erratically.

To address this, training data that consisted of the human driver driving back and forth between the two sides of the track while also driving down the track was recorded and used in training of new models. The increase in training data allowed for the model to include the ability to recover from frames that were not straight down the middle of the track. This resulted in a much more consistent driving model that could drive much further without straying irrecoverably off-course.

The over-fitting problem was also approached by creating a larger looping course approximately forty meters long so the training data could include a larger variety of frames and actions. The longer track required much more training data since the track was much larger with many more potential vehicle positions and sensor inputs that could be encountered. We combined training data that was recorded both by driving straight down the middle of the track and data that recorded sessions of driving in a back and forth pattern. The training data also included training sessions that went both ways around the track and that started from different locations.

The longer, more complex track revealed additional complexities. The longer training sessions and the fact that gathering training data may span multiple days increased the number of variables that needed to be observed. The level of charge in the cars batteries noticeably changed the models ability to drive autonomously on a given model. We determined that this was due to the DC motors turning slower due to the difference in available power. The training data for a given model and the testing of the model had to be done with the same level of charge in the battery. The amount of light in the room was also a variable that had to be addressed. A model's ability to drive consistently was reduced when it ran in light conditions that did not closely match the light conditions of the training

data that was used to train the model. It was becoming more difficult to train the model across all the possible variations of conditions that would effect the model. Gathering training data and testing models was then dependent on inconvenient factors that were difficult to control.

To achieve a model that could train itself to account for and adapt the ever increasing numbers of variables without requiring a human to manage and track edge cases, we started to move towards the next implementation and experiment with reinforcement learning. We did some simple testing on the raspberry pi 3 that involved training the supervised network on the raspberry pi 3 instead of the more powerful computer. We found that a single online training frame of the convolutional neural network we were using for image processing took approximately ten seconds. This meant that training even a single model using reinforcement learning on a raspberry pi 3 would be a significant time investment. We sought both a simpler model to decrease computation requirements of online training and a method to establish how consistently a model could be trained.

3.3 Reinforcement Model in Simulation

Although we were making progress towards implementing a model on a physical car, we learned from our previous experiments that isolating variables such as lighting, obstacle positioning, or battery charge for experimentation was becoming an unreasonable task. The physical cars we were working with had many variables that we could account for but could not control well enough to allow isolation of variables. Training and testing of the model had to be done in the same lighting conditions, which changed by the time of day. The car battery had to be equally charged for both training and testing. The battery lasted for approximately fif-

teen minutes before there was enough power loss to effect the speed of the car and significantly alter the results of testing. The car also used DC motors, which have a delay in increasing and decreasing speed. This results in the acceleration and deceleration portions of driving to be unusable for training inputs.

Training a reinforcement model requires that the model be able to take an action and record the result. The large variance in the distance that the DC motors were moving the car would have necessitated building another car with more control over driving distances per frame. A system to return feedback for determining the reward or penalty would have to be another system that would have to be created and developed. This additional system increased the number of moving parts in our model development that could be error prone. Another major concern with training a reinforcement on a physical car was that the Raspberry Pi 3 that we were using for computation power was not capable of inline training of the model while it was driving. Each backpropogation was taking approximately ten seconds.

The increasing complexity drove the desire to create a model that could generate its own training data for changing conditions instead of continuing to find and track ever-increasing numbers of variables. To test our ability to train the model in this system, we chose to move to train the model in simulation instead of building another car. This allowed for greater control and the ability to focus on single variables at any given time. In the following analysis, we consider experiments run in a simulated environment that shows the actor-critic reinforcement models ability to generate its own training data and learn to drive.

3.4 Simulation Architecture

The simulation was created as a combination of features from the two physical cars. The attributes were chosen for their simplicity and ability to be isolated.

In the simulation model five distance sensors were considered that were positioned on the car and aimed in a radial pattern outward. The central sensor was aimed directly forward relative to the car. The furthest left and right sensors were ninety degrees left and right from the central sensor. The other two sensors were aimed at forty-five degree angles between the central sensor and the side sensors. These five distance sensor inputs are the only information used in the simulation to establish the state of the car. These inputs mimic commercially available sensors like IR, sonar, radar, and LIDAR sensors.

The actions that the car is allowed to take were also chosen for simplicity in the simulation. The car is set to travel forward at a constant speed. The system can select to either turn right, turn left, or go straight. The turning is a simulation of the wheels being angled like the second physical car instead of the in-place turning of the first physical car. This limitation on actions removed the complexity of selecting from a continuous steering angle. It also simplified the design of the simulation algorithm since the car must drive forward at all times and cannot simply rotate in place for long duration of time. These actions can be taken on a frame by frame basis.

The simulated track is represented as the loop used to test the second physical car. It is a simple square loop that is too narrow for the car to be able to make a full one-hundred-eighty degree turn within a single lane. If the car is moving forward within an episode, it can be concluded that the car is successfully navigating the track. Since the track is too narrow for a continuous turn outside of completing

the full loop, the car would intersect one of the walls before fully turning within the width of the track.

3.5 Metrics

To successfully have the car to train itself to drive, objective metrics had to be established and all subjective metrics had to be removed from the judgment of how well the car was driving. The physical models were being judged by humans watching the testing and was subjectively comparing the automated driving against human driving while the training frames were being recorded. This kind of judgment was not going to work using a system simulation for two reasons. First was that the simulation must have well defined success metrics. The other reason is that having the neural network train itself meant that there was no section of successful human driving to use for comparison.

Considering the subjective nature of the judgment of good driving, we choose to instead determine a single simple metric for when the car has driven badly. If the car intersects with any of the borders of the track, it is determined that the car has crashed. We define successful driving to occurs when the car completes a defined minimum amount of time without having crashed.

In the specific simulation experiments conducted in this paper, we split each driving session into episodes. An episode begins at a starting point on the track chosen by how the episode is currently being reset after the car has intersected with the track or if the car successfully drives one thousand time steps without crashing. Each time step accounts for one set of simulated sensor inputs and one car action that moves a set distance. We define the model to have successfully learned to drive if it has completed ten sequential one thousand step episodes

without crashing. This requirement was chosen simply to define a quantifiable value to success and to ensure that the success was repeatable. Once this occurs, the model is considered to have successfully converged on a autonomous driving model and no more training episodes are run. Being in a simulated environment, the car does not change position outside of these defined incremental steps so processing time does not impact the ability to train the model. It also results in time steps not being an amount of time, but a set of inputs and outputs.

3.6 Model

The model for the simulation utilizing the advantage actor critic network we use is composed of the sensor input for state information. The simulated sensors do not include random noise, the difficulties associated with random noise in sensors are variables beyond the scope of this experiment and much of the reason for using a simulated system to establish the viability of this system. This state information is fed into two neural networks. One neural network is used as the actor, which selects which actions to take based on the input state. The other neural network is the critic which is used to predict the rewards that will result from a given state. A diagram of the model used in to drive our simulation is shown in figure 3.1.

Both the actor and the critic neural networks have a single input layer with five nodes. One node for each distance sensor. They then have two internal dense layers composed of twenty-four nodes each. The actor network ends in an output layer of three nodes, one for each possible action that the car can take. Each action node outputs the probability that the associated action is the optimum action. This is trained into the actor network by using the critic network the

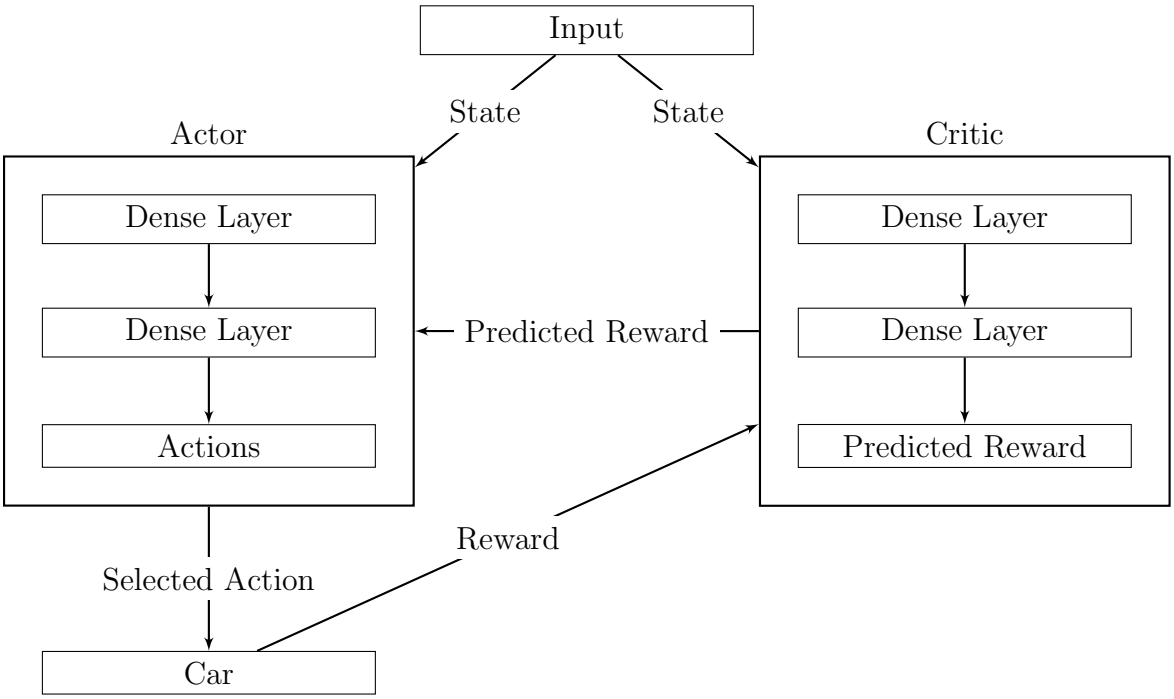


Figure (3.1) Diagram of Advantage Actor Critic Model

predict rewards. This uses two time-steps as outlined in the actor critic algorithm 2.

The actor outputs the advantage values of each possible action that the car could take. These advantages are decimal values that sum to one and we use a pseudo-random number generator to pick which of the actions to take, but the selection is weighted by the advantage value of the associated action. An action with an advantage value of 0.1 would be selected ten percent of the time, where an action with an advantage value of 0.9 would be selected ninety percent of the time. This can also be described as the action being chosen stochastically based on the advantages. The critic network has a single output node that outputs the predicted reward for the inputted state. These predicted rewards can be converted into the relative advantage values of each action. The actor is fitted to these advantages and the critic is fitted to the target. The reason this is useful

for a driving model is this allows the critic to give low score values for states that it has learned eventually lead to collisions. In turn, the critic's ability to calculate states that lead to crashes allows the actor good feedback to allow it to take actions that are less likely to lead to crashes.

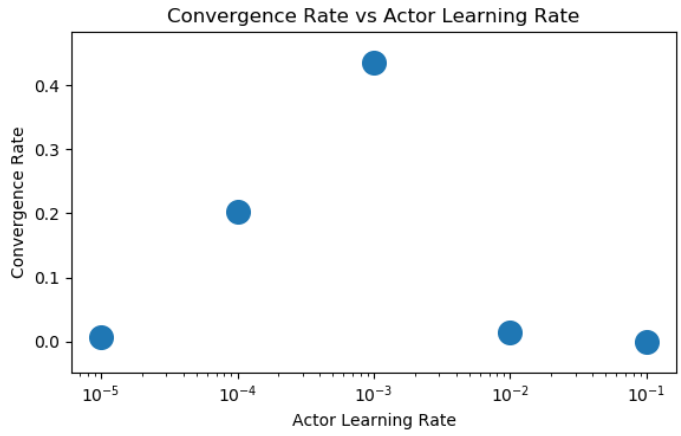
3.7 Hyperparameters

The trainability of the model changes with the selection of the hyperparameters. Hyperparameters are parameters within the machine learning model that change how the model is manipulated during training. The parameters are the values within the nodes of the neural network whereas hyperparameters determine how the parameters are changed during the training process. Hyperparameters are set by the models' creator to guide the training process of the neural network. Due to the possibility of hyperparameters being non-linearly related, all combinations of the tested hyperparameters were tried. The hyperparameters we tested were actor learning rate, critic learning rate, discount factor, and two additional hyperparameters that controlled how the car reset between training episodes. The results of these training sessions are shown in figures 3.3, 3.4, 3.5, and 3.6. These hyperparameters and their results are reviewed individually in the following sections.

3.7.1 Learning Rate

Actor and critic learning rates must be selected for the update function of the actor-critic model. The learning rate is a decimal value used to determine how much each individual training step changes the existing model. These learning rates altered how rapidly the each network would update given new information.

(a) Convergence by Actor Learning Rate



(b) Convergence by Critic Learning Rate

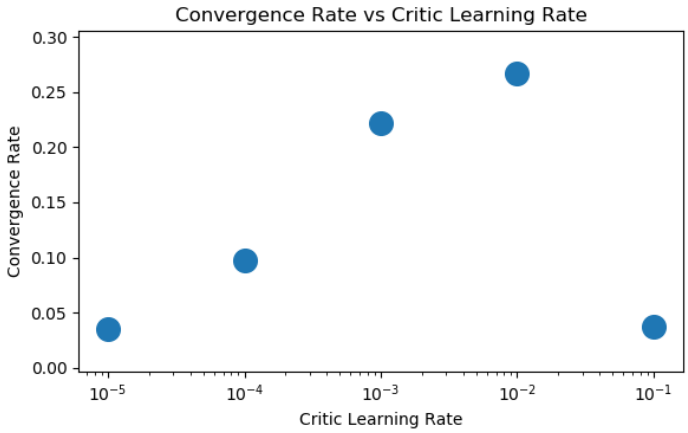
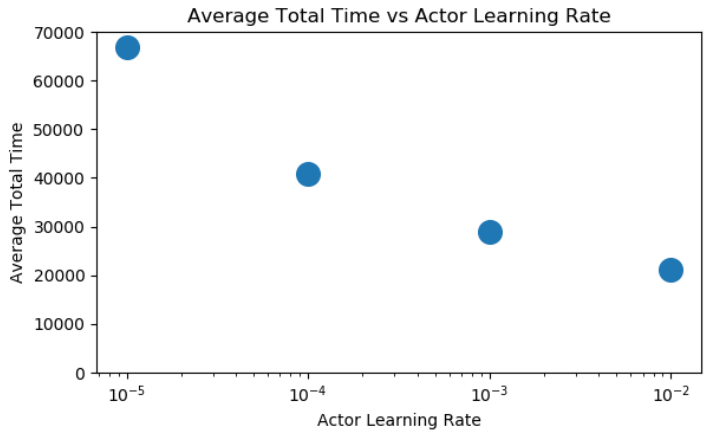


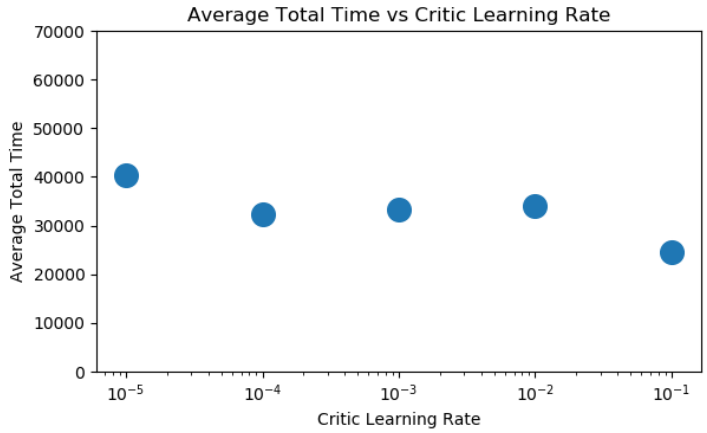
Figure (3.2) Average Training Time and Convergence Rate by Actor and Critic Learning Rates

(a) Average Training Time by Actor Learning Rate



Converged Models Only

(b) Average Training Time by Critic Learning Rate



Converged Models Only

Figure (3.3) Average Training Time and Convergence Rate by Actor and Critic Learning Rates

Both actor and critic learning rates were tested independently to discover how the learning rates interacted with each other in the training of the model.

If the model had too low of a learning rate, it would take excessively long to train. If it had too high of a learning rate, it would set too high of a value on new information and not learn enough from the historical experience already in the model. Either option prevents the model from efficiently converging on a viable autonomous driving solution.

We see the learning rates compared against the average time that a model took to train and the convergence rate of the model in Figure 3.3. Convergence is defined as the model completing ten sequential episodes with no collisions within a maximum of 500 episodes.

In figure 3.2a we see that there is a relatively narrow band of increased convergence rate centered at 10^{-3}. This seems to be the ideal learning rate for the actor given a maximum training duration of 500 episodes. We can see from 3.3a that the average number of training frames that it takes for the model to learn to drive ten-thousand sequential frames without crashing decreases as learning rate increased. This metric is taken exclusively for models that did converge, so we observe when comparing Figure 3.2a against 3.3a that this speed comes at the cost of the model being less likely to converge. We can also observe that too low of a learning rate took significantly more frames on average to converge for both the actor and the critic networks. This shows that it is likely that most of the models that did not converge were because they exceeded the time limit.

We can see that setting the learning rate for the actor will require a careful balance between having a small enough value that the model is capable of learning from experience and a large enough value that the model trains in a reasonable

amount of time. It is beyond the scope of this paper, but it would be interesting to test actor learning rates decrease over time and how that impacts training speed.

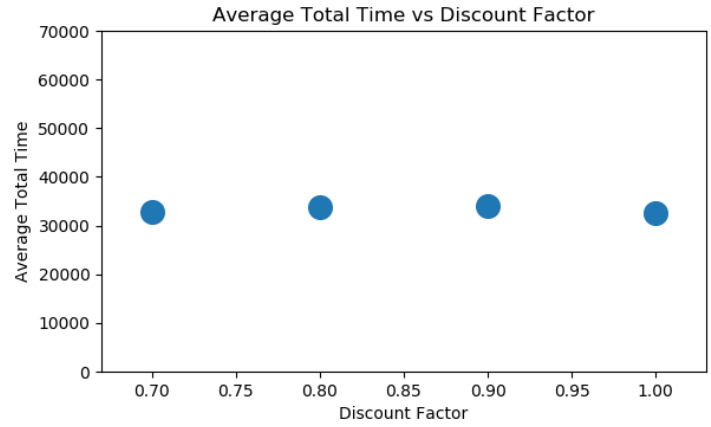
Figure 3.2b shows that the range of learning rate values with increased convergence rates for critic learning rates is wider. The critic learning rate also seems to have very little impact on the average time it takes for a model to train. Since the actor network chooses the actions that lead to motion or crashes, it makes sense that models that do manage to converge will not show a pattern in average time relative to critic learning rate. The critic network does not directly impact the driving of the car, but instead is used to calculate the advantages that the actor uses to train. If these values become accurate enough in a short enough period of time that the actor can viably use them for training, then the actor learning rate is the determining factor on whether or not the model learns to drive.

3.7.2 Discount Factor

Discount factor is also an integral part of the actor-critic model and needs to be established for the update function so that the model could account for the time dependence of driving. The lower the value of the discount, the more the model weights its decisions towards immediate rewards and the less the model values future rewards.

A value of zero would mean that the model would look only at the current frame and its rewards. This is not likely to converge in the driving situation because the feedback value is always the same so every action that has not already crashed is determined to be equally valuable. Looking at only one frame without any consideration for future rewards does not allow the model to recognize actions that would immediately or eventually lead to a crash as negative. Only the final

(a) Average Training Time by Discount Factor



Converged Models Only

(b) Convergence by Discount Factor

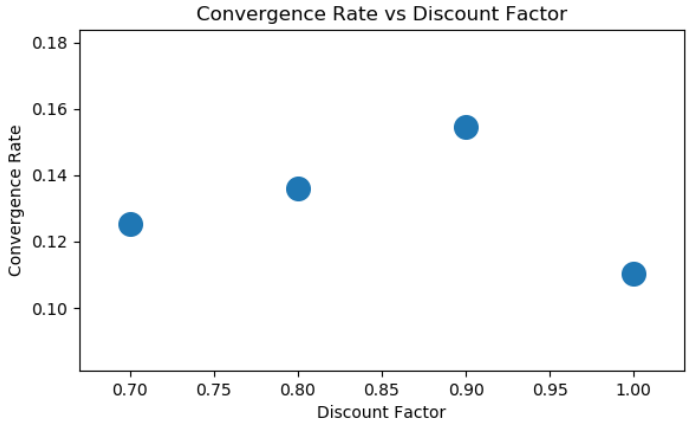


Figure (3.4) Average Training Time and Convergence Rate by Discount Factor

action before the crash would be recognized as bad. By that time, the car has no other action than to crash since it is limited by turning radius.

A value of one would mean that the model would put equal weight on the current frame and the estimated future frames. A crash that can happen in the distant future is less concerning than an immediate crash since it can still be avoided.

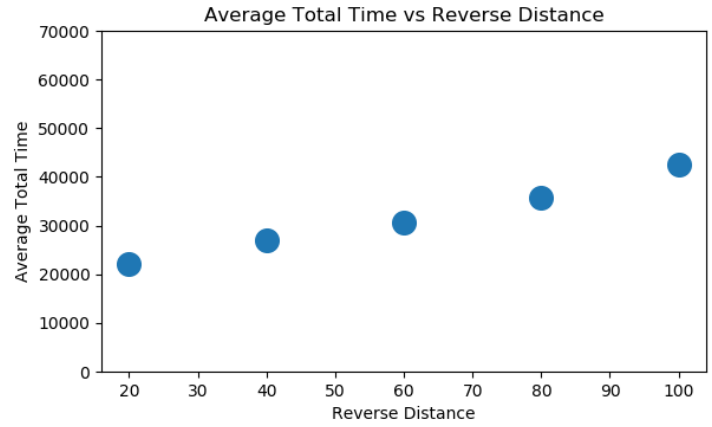
This is reflected in figure 3.4 since a discount factor of 0.90 shows the highest rate of convergence of the values that were tested. That peak suggests that 0.90 is the value for which both immediate and future rewards are balanced in a way that allows the model to best account for time dependence. We can also see that discount factor has minimal impact on the average training time required for convergence as shown in 3.4. This suggests that discount factor needs to be set within an acceptable range for the model to converge and values outside of that range will not converge regardless of how long they are allowed to train.

3.7.3 Reverse Distance

In the autonomous driving system that used supervised learning described in Section 2 there were ongoing concerns stemming from the fact that resetting the car between training sessions was manual and that it was not possible to reset the car into exactly the same starting position each time. This notably impacted the training frames collected on each run and the ability for the model to be tested. The simulation approach used provided a consistent environment and allow for the car to be reset identically for each episode. This cannot be repeated in a real-world implementation.

The dynamics of a car driving system always driving forward and having a

(a) Average Training Time by Reverse Distance



Converged Models Only

(b) Convergence by Reverse Distance

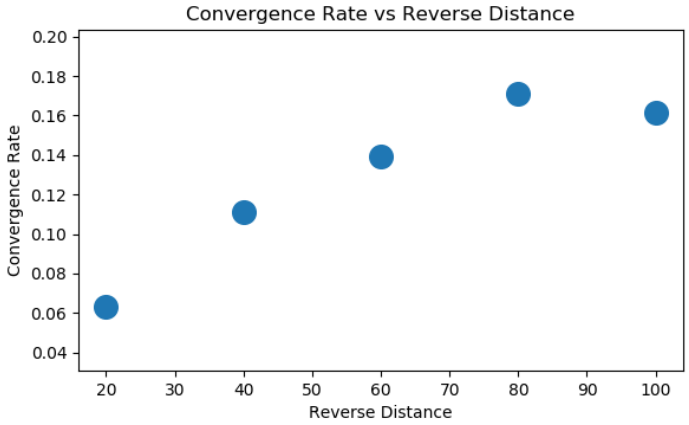


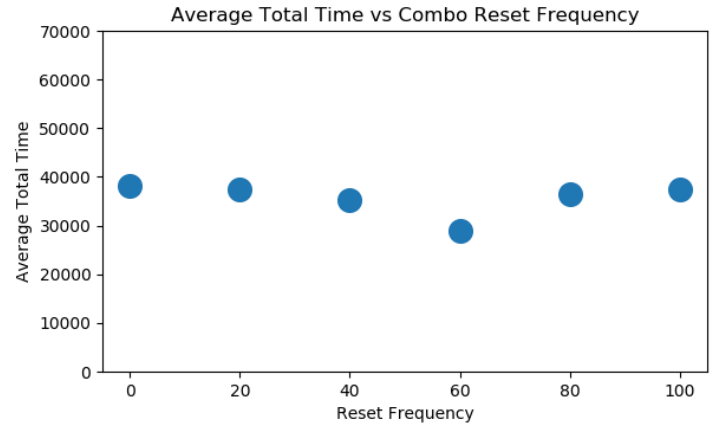
Figure (3.5) Average Training Time and Convergence Rate by Reverse Distance

minimum turn radius allows us to take advantage of the cars physical limitations and simply move the car backwards to return the car back into a pre-crash condition. We wanted to test the viability of this as a reset policy since it could allow for the driving model to be implemented and trained on a physical car without requiring human intervention for resets. Episodes are often very short due to the fact that the car approaches the situations it previously failed quickly and either learns or crashes faster, so allowing the car to train in this fashion makes training a real car much more viable.

In Figure 3.5, we can see that resetting the car via backing it up instead of resetting to the beginning of the track has convergence rates that are similar to fully resetting to the same start condition each time. We also see that there is an additional benefit of shorter reverse distances compared to longer reverse distances in reducing the average amount of time that converging models took to train.

Longer reverse distances converge at a greater rate because the car backs up far enough for the driving model to react in time for car not to crash again. The shorter distances benefit from the ability to immediately repeat events that the model previously failed on. However, the shorter reverse distances had the drawback of getting temporarily stuck in a state that did not allow the car to escape for several episodes since the car is always limited by its turning radius. Given that the car would still drive the furthest possible distance it could achieve before crashing, it would eventually escape this condition. This finding shows that imposed limitations on the simulation algorithm can be used to seek solutions to real-world implementations on a car and design both the model and actions taken to train the model with significantly less overhead than training on a physical car.

(a) Average Training Time by Reset Rate



(b) Convergence by Reset Rate

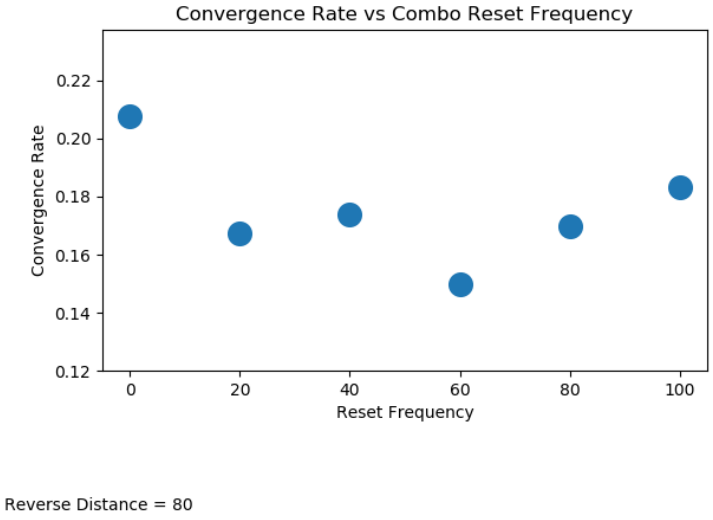


Figure (3.6) Average Training Time and Convergence Rate by Reset Rate

3.7.4 Periodic Full Resets

We also wanted to test the impact of collecting training data from multiple sessions, each of which would be set up by a person and have the car start in a position that the human set it in. In Figure 3.6 we see a combination of resetting the car by reversing a set distance and of the simulation emulating multiple training sessions by restarting the vehicle from the original placement.

We see that fully resetting every episode results in the highest rate of convergence. If we choose to use reversing the car as a reset method, that highest convergence rate occurs when we allow the highest number of training sessions before resetting to the original placement. We can see that allowing long training sessions between times a person would reset the vehicle to the original position both reduces the amount of human intervention to train a model and is more likely to converge.

Chapter 4

Conclusions and Future Work

In this paper, we used simulations to develop a simple neural network model and tested how that model would perform if implemented on a physical car. We showed that we could simulate distance sensors as inputs and use a simulated track to gather training data that would train an actor-critic network.

Implementing and testing a physical car with this model is beyond the scope of this paper. Additional variables and hyperparameters would have to be tested and developed for before fully being able to implement this neural network model in a real-world setting. These would include developing a system that could identify bad states from sensor inputs. It would also include testing and refining the model for the limited computation capacity of embedded systems.

We showed that we can utilize a simple simulation system to refine hyperparameters and reduce the amount of time consuming real world testing required to develop a new neural network model. We found that we could find a range of values for learning rates and discount factors that would be likely to result in the driving model that would converge to an autonomous driving solution. We also established a strategy for resetting the car that would allow the car to train

continuously without requiring a person to reset the car to the beginning location each episode. We demonstrated that this style of training would be practical across multiple training sessions. We also showed that this strategy converged at a rate that was comparable to moving the car after each episode.

In summary, we used a simulation system to successfully develop a neural network model and showed that it was capable of learning to drive with minimal human intervention.

References

[1] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[2] Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 305–313. Morgan-Kaufmann, 1989.

[3] Long Tran-Thanh, Archie Chapman, Enrique Munoz de Cote, Alex Rogers, and Nicholas R Jennings. Epsilon–first policies for budget–limited multi-armed bandits. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[5] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[6] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[7] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[8] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[9] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[10] AAEON Technology Inc. *UP Datasheet*, 2016.

[11] Intel. *Intel RealSense Camera R200*, 6 2016. Revision 001.

[12] Donkeycar. `http://www.donkeycar.com/`. Accessed: 2018-05-18.