University of Nevada, Reno

Optimal Strategy Imitation Learning from Differential Games

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

by

Niki Silveria

Dr. Richard Kelley/Thesis Advisor

December, 2017

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

NIKI T. SILVERIA

Entitled

Optimal Strategy Imitation Learning from Differential Games

be accepted in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE

Richard Kelley, Ph.D., Advisor

Kostas Alexis, Ph.D., Committee Member

Thomas Quint, Ph.D., Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

December, 2017

Abstract

The ability of a vehicle to navigate safely through any environment relies on its driver having an accurate sense of the future positions and goals of other vehicles on the road. A driver does not navigate around where an agent is, but where it is going to be. To avoid collisions, autonomous vehicles should be equipped with the ability to to derive appropriate controls using future estimations for other vehicles, pedestrians, or otherwise intentionally moving agents in a manner similar to or better than human drivers. Differential game theory provides one approach to generate a control strategy by modeling two players with opposing goals. Environments faced by autonomous vehicles, such as merging onto a freeway, are complex, but they can be modeled and solved as a differential game using discrete approximations; these games yield an optimal control policy for both players and can be used to model adversarial driving scenarios rather than average ones, so that autonomous vehicles will be safer on the road in more situations. Further, discrete approximations of solutions to complex games that are computationally tractable and provably asymptotically optimal have been developed, but may not produce usable results in an online fashion. To retrieve an efficient, continuous control policy, we use deep imitation learning to model the discrete approximation of a differential game solution. We successfully learn the policy generated for two games of different complexity, a fence escape and merging game, and show that the imitated policy generates control inputs faster than the differential game generated policy.

Table of Contents

List of Tables

List of Figures

1. Introduction

For autonomous vehicles, precise navigation through dynamic environments is necessary for the safety of passengers, surrounding vehicles, and pedestrians. However, driving on a freeway, navigating city streets, or pulling into a neighborhood -- any real life driving scenarios -- are situations that add the additional difficulty of avoiding collisions in an environment with other intentional agents. An intentional agent is any object in an environment that moves according to its own attitudes, goals, and beliefs. Most importantly, an intentional agents' choice of action is affected by the actions of other agents. All cars and pedestrians may be considered intentional agents, independent of whether they are autonomous or fully under human control. As humans, we are able to make assumptions about other agent's intentions and, based on this knowledge, how these agents will move in reaction to our actions. The understanding of where others will be in reaction to ourselves allows us to manipulate the environment in our favor. To avoid collisions effectively, an autonomous vehicle must also be able to emulate this estimation of future positions and agent reactions within a continuous state and control space.

Although it seems the straightforward choice, future positions of pedestrians or other cars on the road are not accurately estimated using solely object tracking algorithms. Object tracking assumes that the only intentional agent in the environment is the tracker. Other agents' movements are assumed not to be affected by any actions taken, and it contains no structure to account for competitive actions taken in response some driver maneuver. It is possible that one agent may attempt to prevent the actions of another in order to achieve its own goals faster, more easily, or to prevent the other from reaching its goal. In the case of highway driving, merging and changing lanes are both actions that require the cooperation of at least one other driver. It is possible that other drivers may merge from different lanes and reduce the merging space or speed up to make more room. A vehicle must have some kind of intent recognition in order to plan around anticipated reactions of other agents, especially when those agents may be competing against it.

Differential game theory is turned towards instead to model actions and reactions in a given situation. Game theory provides a method of generating optimal controls in a multi-agent, competitive environment. With the publishing of their book *Theory of Games and Economic Behavior*, John von Neumann and Oskar Morgenstern solidified the concept of mathematically determining an optimal way for either competing or cooperative players to behave in games of choice (Neumann and Morgenstern). Players create a strategy, or series of decisions, that results in some return of value to each player at the end of the game. The strategy that is superior to any other is considered the optimal strategy. A greedy policy, chosen with respect to this optimal strategy, is guaranteed to be the optimal optimal policy.

The original formulation of game theory by Von Neumann and Morgenstern is not applicable to autonomous vehicles as the physical dynamics of vehicles are not accounted for. It is more appropriate to apply differential game theory (Isaacs). Representing two player games with ordinary differential equations leads to strategies that are expressed as a series of choices in continuous time, rather than as a sequence of steps. Several authors have previously presented papers utilizing the optimal controls generated by differential games relating to vehicle behavior, such as naval collision avoidance, but the optimal controls generated by these algorithms are practically implementable for more complex problems (Ho, Yu-Chi, et al.). The solutions presented by these algorithms require large amounts of calculation to obtain in continuous space. They can, however, be discretely approximated, but then produce large tables of state and action values that take up large amounts of memory for complex problems. While the obtained strategies are guaranteed to be optimal, a more efficient solution needs to be developed for realtime, limited space situations.

Another method of learning an optimal way to avoid collisions is through machine learning. Deep neural networks are currently a popular choice in research to perform many functions in the autonomous car industry. Variations of deep neural networks are being explored as collision detection, lane navigation, and full end-to-end navigation control, and large amounts of data for these problems can be collected by human drivers or simulated to train complex models (Xu, Huazhe, et al.; Chen and Huang; Wang, Pin, and Chan). More recently, improvements in imitation learning, learning an end-to-end policy, have enabled successful models to learn sequences of actions better than supervised learning alone. However, the data to train models is often collected for ideal situations, where autonomous vehicles seek to avoid objects in their environment and abide by the rules of the road while assuming that other vehicles are cooperating. A method of determining what to do in an adversarial situation is necessary while there are still humans sharing the road with autonomous vehicles.

In order to take advantage of the optimal control strategies offered by game theory in adversarial conditions without requiring large amounts of real time processing, and data storage, we propose modeling the results of game theory with deep imitation learning. Strategies generated by differential game theory consist of a series of states and actions which can be used as data to train a neural network in a supervised fashion. A test if this combination would produce reasonably accurate results was created, and two differential pursuit-evasion games were created for an application of game theory to generate optimal controls and train an imitation learning model.

A simple fence escape game and a more dynamically complex vehicle merging game were defined to test the architecture of our solution and to see if it would generalize to more complex game environments. Optimal actions for states in each game are calculated using a variation of the iterative iGame algorithm developed by Mueller and Zhu. Once collected, these state-action pairs are used to train a deep neural network model using an implementation of DAgger. Additionally, a reinforcement learning algorithm was applied to the results of the fence escape game to further optimize its control policy.

The model learned an accurate representation of the game theory generated policy, and was able to provide controls twice as quickly. The success of the model indicates that collecting optimal action training data from game theory could be a good method of training neural networks on cars to navigate situations where ideal behavior cannot be expected from the environment around them.

2. Background and Related Works

The combination of differential game theory and machine learning is proposed given two functions; the ability of differential game theory to provide the optimal control policy for players in a given game environment, and the ability of a neural network to learn a policy provided by an expert. In order to provide a foundation for understanding this work, both of these concepts are delved into individually before introducing how they work together. An introduction to the relevant game and differential game theories and current solutions are presented first, followed by a short tutorial on neural networks, deep learning, and imitation learning. Then, recent works related to game theory and machine learning for autonomous cars specifically are explored. The contributions of our work in the form of the connection between game theory and machine learning is discussed with relation to these references.

2.1 Differential Game Theory

Game theory is a mathematical method for calculating the best actions a player can take while playing a game. In this case, a game is any situation involving two or more players that make choices where he results of all players choices result in some value return to the players. In many games, players' actions result in a benefit to themselves and a cost to their opponent. One such famous game is the prisoner's dilemma. Two prisoners that have been arrested for some crime have the option to either betray the other’s involvement, or remain silent. Their prison sentences are decided based on the combination of their responses, but they cannot communicate with each other. For example, if both prisoners remain silent, they both serve a year in prison, but if prisoner A betrays prisoner B and prisoner B remains silent, the betrayer will not serve any prison time while the silent prisoner will have to serve three years (Selten). A diagram of the typical rewards for a prisoner’s dilemma is included in table 1.

Table 1: A formulation of the prisoner’s dilemma. Prisoners have two choices of action that determine their number of years in prison. The number of years in prison is represented as a negative reward value.

Prisoner A \ Prisoner B	Betray	Stay Silent
Betray	-2 \ -2	0 \ -3
Stay Silent	-3 \ 0	-1 \ -1

The goal of a prisoner in the prisoner’s dilemma is to earn the least amount of prison time, so they wish to discover which action is the best action to take to achieve that goal. A player’s action choice is referred to as that player’s strategy, and the strategy that results in a greater reward than any other strategy is the dominant strategy or *optimal strategy* (Issacs). The goal of game theory is to calculate an optimal strategy for one or all players in the game.

The solution to a game can be found according to multiple criteria. On one hand, a game could be said to be solved when the player’s strategies reach an *equilibrium*. An equilibrium is reached when a strategy is found for both players that cannot be improved, however the definition of improved changes equilibrium points. Nash Equilibrium for example, defines lack of improvement as when neither player can increase their reward by changing their strategy while the other players' remain constant. Alternatively, the trembling hand perfect equilibrium takes into account when players may make unintentional choices, or tremble (Stelton). Alternatively, the solution of a game can be determined with a minimax function. In this way, different solution concepts can lead to different strategies for players in a game, but would still result in a solved game. The prisoner's dilemma, however, is a very simple game and changing solution criteria does not lead to a different solution.

Prisoner's dilemma can also be played iteratively: each prisoner makes the choice to betray or stay silent more than once and attempts to maximize their rewards over time rather than just for a single game instance. The strategy that a player forms to choose actions based on the state of the game over time can also be referred to as a *policy*. Different policies result in different rewards, so, similar to strategies, the policy that earns a player more reward than any other policy is the *optimal policy.* If optimal policies could be generated for games of vehicles interacting on the road, an autonomous vehicle could maximize a reward related to avoiding collisions. However, general game theory does not have any knowledge of the kinematics that players are bound by. Solutions to these games cannot be applied to autonomous vehicles as they map actions directly to reward values. In order to consider the dynamics for the players, the game can be formed as a differential game instead.

In a differential game, players are described according to their possible states, kinematic equations, and control inputs. Unlike in traditional game theory where players choose actions and receive some reward, players instead choose their control inputs as their strategies and change their state in the game according to their kinematic equations while attempting to reach some defined goal. Goals do not have to be to move towards a static location, they can include intercepting or avoiding a moving target, and players can have cooperative or adversarial goals. An advantage to this formulation of a game is that when a differential game is solved, both players will have optimal control policies that describe optimal paths for them to take towards their goals.

Isaacs originally developed and solved differential games to apply them to military applications (Isaacs). He provided several examples of the use of pursuit-evasion games, where the goal of one player, the pursuer, is to capture or cut off the evader player, whose goal is usually to avoid the pursuer. A common pursuit-evasion game is called the homicidal chauffeur. The pursuer is fashioned after a vehicle that has a high top speed but large turning radius and the evader is a pedestrian with low top speed but small turning radius. The solution to the game depends on whether the game is posed as a *game of kind* or *game of degree.* Games of degree have a continuum of outcomes, often solutions provide the controls of a player or the time until capture, whereas solutions to a game of kind report a finite outcome of a game, such as whether a player will win or lose based on starting positions. A sample diagram for the homicidal chauffeur game is found in fig. 1.

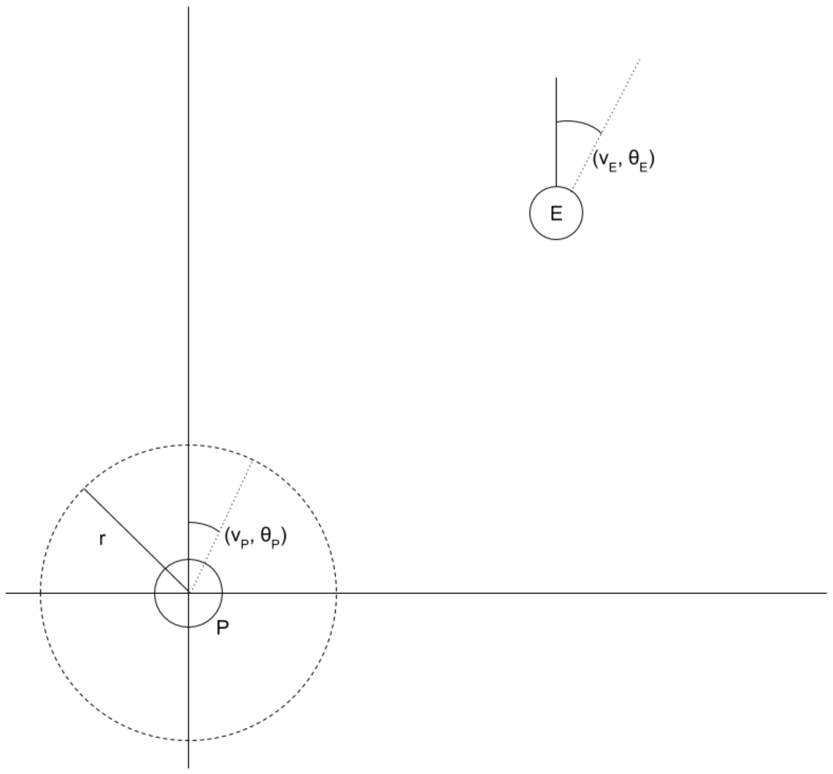


Figure 1: An illustration of the homicidal chauffeur game in a reduced state space. The pursuer P has to get within distance r of E to win the game. Players control their velocity and turning radius.

2.2 Differential Game Solutions

Optimal interception strategies for collision avoidance and capture can be modeled and solved as a pursuit-evasion game. The solution to a differential game can take the form of the optimal value function, optimal control policy, or optimal path, and any one of these solution formulations can be used to derive the others (Isaacs). Solving a differential game, rather than a traditional game, as a game of degree will provide an autonomous vehicle with a useable control policy for collision avoidance in various situations, however, the solutions to a differential game are not as easily calculated.

To derive a solution to a differential game in the continuous space that it is defined in, Isaacs defined the *Main Equation*. A generally nonlinear, first-order partial differential equation, the *Main Equation* is a function of the kinematics and controls equations of all the players in a game. When solved it provides a true solution to the game. It is, however, difficult or impossible to solve computationally for complex games. In all of the previously mentioned games, solutions relied on a great deal of pre-calculating equations and attempts to simplify the kinematic equations (Exarchos; Pachter and Yavin; Lewin and Olsder). Directly solving the *Main Equation* becomes impossible for more complex games. In such a vein, games designed for an autonomous vehicle, even simplified, are likely to yield a *Main Equation* that is too complex to solve computationally.

Fortunately, differential games' solutions can be obtained numerically. Using higher order derivations of the value function, to reduce the complexity of the *Main Equation,* and the development of the "multiple shooting method" have provided more computationally viable solutions (Lachner; Morrison et al.). However, some of the most computationally successful numerical solutions involve sampling based or discrete approximations.

Several sampling based methods can be used to solve games, but exhibit varying levels of accuracy and refinement. Algorithms like Rapidly-exploring Random Trees, or RRT*, are built upon randomly sampling states from the available state space. RRT* is used to generate optimal motion planning. It is important to note, however, that RRT* creates an open-loop strategy rather than the closed-loop strategy that is usually generated by pursuit-evasion games (Karaman, Sertac, and Frazzoli). Viability theory has also been applied to approximate a value function as a sampling based method (Cardaliaguet, Pierre, et al.). Multi-grid, successive approximation approaches, such as the one used alongside viability theory by Cardaliaguet, are successful at discretely approximating the value of games, but are limited in accuracy by a predefined and static grid resolution.

To leverage incremental sampling as well as viability theory in order to improve on multi-grid approximation methods, iGame*, was developed (Mueller, Erich, et al). iGame* operates similarly to Cardaliaguet's method, but refines the grid resolution each iteration to the smallest grid that all previously sampled points fit on, allowing the accuracy of the algorithm to increase the longer it runs. iGame* maintains this grid in order to create the value function for a game. This value function is represented as each sampled state having its own value.

iGame* as presented is faster than the previous multi-grid methods, and its asymptotic convergence is formally insured for all states, meaning that as time approaches infinity, iGame* will produce the optimal value function. From the approximated value function, iGame* can also generate the control policy that is greedy with respect to the value, and if that value function is optimal then the resulting policy will also be optimal (Sutton and Barto). For these reasons, iGame* is chosen to generate solutions to the proposed autonomous vehicle differential games.

iGame* generates solutions well; however, its discrete solution needs to be modeled in the continuous space, or have some form of policy compression applied to the results in order to avoid comparing states from a table of thousands of points in a real-time collision avoidance environment. Using deep learning to learn the policy with imitation learning accomplishes both of these objectives.

2.3 Deep Learning

Deep learning is one way to accomplish machine learning, or getting an algorithm to learn from experience. A deep learning algorithm is built to fulfill the definition of machine learning as a connection between several neural networks.

Broadly, machine learning is the ability for an algorithm to improve its performance on a task T, with respect to some performance measure P, given exposure to some experience E (Mitchell). As far as the task T is concerned, machine learning can be applied to problems such as classification, regression, translation, or anomaly detection, among other fields. The network built to learn iGame*'s policy is a classifier, identifying which is the proper control input to take given an input state. Formally, this classifier is tasked with producing a function f:Rn -> {y1, y2, …, yk} such that when given a state x the model will produce an appropriate y from the set of available controls (Goodfellow, Ian, et al.). Classification is a subset of supervised learning. In supervised learning, every element of a training dataset is accompanied by a label and provides the experience, E, part of machine learning. Specifically, a set of elements {x1:y1, x2:y2, …, xi:yi}, where xi is a feature observation and yi is its label, is used to train a network.

One of the simplest neural networks is one that performs linear regression. Although classification is the goal of the network created to model iGame*, the construction of a linear regressor is similar and provides a good introduction to the machine learning concepts that are necessary to extend into deep learning. Following is a condensed explanation of a simple network, based primarily on the work in Goodfellow, Bengio, and Courville's Deep Learning book, which can be referenced for more in depth information about different types of neural networks.

The task, T, of a linear regressor is to predict the value of a scalar $y \in R$ from an input vector $x \in R^n$. The model should output ŷ, an estimation of y, following the equation:

$$\hat{y} = w^T x + b,$$

where *w* is a vector of *weights,* $w \in R^n$, that are applied to x to determine the estimated ŷ, and *b* is a *bias* applied to the formula. Weights decide how much emphasis a feature in x has on the final prediction; changing the values of individual weights affects whichever feature it is multiplied with. Values can be: (1) positively valued, which increases ŷ, (2) negatively valued, decreasing ŷ, or (3) zero, causing no effect to the final value. If the input expands in feature size, *w* can also be a matrix.

The regressor also needs some way to measure performance, P. A common performance measure for linear regressors is mean squared error. Given the prediction ŷ and true label y, vectors of size m, the mean squared error (MSE) is defined as:

$$MSE = \frac{1}{m}\sum_{i}(\hat{y} - y)_i^2 .$$

Since the MSE is larger when ŷ and y are farther apart, and smaller when they are similar, the regressor can be said to perform better when the output of MSE is smaller. With a performance measure defined, the regressor needs some way to change the weights *w* and bias *b* after observing labeled data, the experience E, to minimize the MSE.

Gradient descent is the most commonly used method of updating w. Taking the derivative of the cost function yields its slope, or gradient, and given that information the values in the w vector can be updated to "descend" the slope of the cost function towards the minimum value. Effectively, after evaluating each data point in the training dataset, gradient descent can be run to update w before the next evaluation. Updates to *w* and *b* are scaled with a learning rate parameter to control how much the weights are changed each iteration. Note that on large datasets, gradient descent can be very slow, as it runs in O(n) time, so *stochastic gradient descent* is usually run on mini-batches of examples selected uniformly from the dataset instead of updating after every evaluation.

Simple machine learning solutions, like the linear regressor, can be powerful tools, but cannot be generalized to solve many complex problems. Deep learning was created to overcome problems like the *curse of dimensionality*, the number of distinct configurations of a set of inputs can be much larger than the number of available training examples, and the *local consistency prior,* that learned functions are assumed to change very little within a small region, along with other problems that hinder generalization (Goodfellow, Ian, et al.).

Deep neural networks can be thought of as a series of functions, like the linear regressor, that are chained together. Each function makes up a layer of the network, and the output from one function is passed on to the next, until the last layer's function produces an output. For example, three functions in a chain would form, $f(x) = f_3(f_2(f_1(x)))$. The number of layers in a network is the network's depth. Figure 2 is a sample diagram of a 3 layer *deep feedforward network.* In a feedforward network, the input, x, is only passed through each layer once in a straight line, as visible in the diagram. Layers that are not first or last, f_2 in the figure, are called *hidden layers*; during training, these layers are used to find the best approximation of *f(x)*.

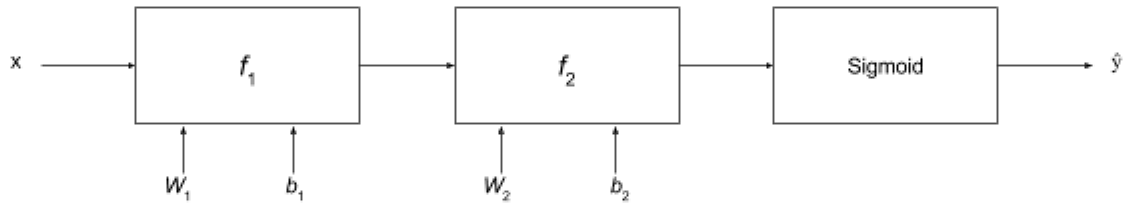


Figure 2: An illustration of a feedforward deep neural network. This network has three layers, two linear functions and a sigmoid function. The inclusion of a sigmoid function as the final layer means that this could function as a binary classifier.

Like in the linear regressor, deep networks are also updated using gradient descent. The chain rule is used to propagate changes to all weights and biases in the network working backwards from the last layer to the first in a process that is aptly known as *back propagation.* Networks can be made as large or as small as needed to learn a dataset, and functions do not have to remain linear. More complex networks can learn vast amount of associations.

2.4 Previous Deep Learning for Autonomous Vehicles

Variations of supervised learning are used in many neural networks intended for autonomous applications. Variations of deep neural networks are being explored as collision detection, lane navigation, and full end-to-end navigation control, as large amounts of data to train complex models for these problems can be collected by human drivers or simulated (Wang, Pin and Chan) (Xu, Huazhe, et al.) (Bojarski, Mariusz, et al.). The Next Generation Airborne Collision Avoidance System (ACAS-X) family of systems, for example, is a deep neural net explored by a Stanford team as a policy compression algorithm to reduce the memory requirements for air traffic collision avoidance tables. the purposes of policy compression, with particular success seen by the Next Generation Airborne Collision Avoidance System (ACAS X) family of machine learning solutions (Kochenderfer, Mykel, Holland, and Chryssanthacopoulos). However, pure supervised learning is not the most accurate way to train a deep learning network. There is an assumption that each new state is independent of the last. In truth, the next state relies on the previous state estimation and chosen controls. In practice, a model that makes a mistake can result in experiencing a state that it was never trained on. The model would be forced to guess at the best controls to make from the new state, most likely inaccurately, leading to compounding errors. To avoid models "getting off track", the DAgger architecture was created to train a better imitation learner (Ross, Stephane, et al.).

DAgger progressively limits the student learner's policies and action selection until it can no longer create off track policies.

Imitation learning can be used to learn a policy accurately, but the choice of policy to learn is also important. Most current applications of machine learning to control autonomous vehicles train on human collected data, either generating a simulation based on an aggregate of controls or training directly from human provided input to go with state or image data, but the data does not reflect adversarial situations. Most adversarial research aims to control for attempts at deceiving sensor input rather than avoid collisions with a potentially adversarial driver (Szegedy, Christian, et al.; Huang, Sandy, et al.; Gu, Shixiang, and Rigazio.). Our method seeks to generate optimal control strategies for worst case scenarios rather than average driving ones, so that autonomous vehicles will be safer on the road.

3. Approach

To create a control policy for an autonomous vehicles that considers intentional agents, a situation is modeled as a differential game, iGame* is used to approximate the value function and corresponding greedy policy, then an implementation of DAgger trains a student to imitate iGame*'s policy. The setup of this system is detailed from start to final output considering the connections between each section. The definition of a differential game in a way that is usable by iGame* is covered first, followed by a description of the iGame* and DAgger algorithms and their implementations, as well as the final output of the system.

3.1 Differential Game Setup

A pursuit-evasion game can be used to model many of the situations faced by an autonomous vehicle, but it must be defined in such a way that it can be solved in iGame*. Recall, a differential game is defined with a set of ordinary differential equations that describe the dynamics of the game environment. These equations are functions of the current state and controls of the system, at a time t, for each agent in the environment. Each agent is governed by its own equation and controls, but the state for the entire game is based on the state information of every player at the current time. Several subsets of the state space must be defined for use in iGame*.

The set of all possible states in the state space are defined as X, where $X \in \mathbb{R}^N$. The specific state that is represented at time t is x(t). For two-player games, we consider a pursuer player and an evader player which have different goals. The pursuer wishes to move the system from the initial state, x(0), into a set of states where the evader loses the game X_{bad}. The evader, meanwhile, wishes to move to a winning set of states, X_{goal}, while remaining free from X_{bad}. X_{free} is considered to be a constraint set where the game is neither in a winning or losing state, $X_{free} \doteq \text{closure}(X - X_{bad})$.

The controls space of the differential game must also be defined. Both players control the system through their choice of strategies in order to achieve their stated goals. Let sets U and W represent the control strategies of an evader and pursuer, respectively, and be defined as:

$$U \triangleq \{ u(.) : [0, +\infty) \to U, \text{measurable}\}$$

$$W \triangleq \{ w(.) : [0, +\infty) \rightarrow W, \text{measurable}\}$$

where $U \subset \mathbb{R}^{ma}$ and $W \subset \mathbb{R}^{md}$.

Finally, the following equation would define the dynamical system:

$$x'(t) = f(x(t), u(t), w(t)),$$

where $x'(t)$ denotes the next state and $u(t) \in U$ and $w(t) \in W$. The set of kinematic equations that move the state according to the control values are defined individually for each game considering the players and environment. With the dynamics of the system defined, computing the value function of the resulting game is the job of iGame*.

After defining the differential game, the controls and kinematic equations can be used in iGame* to approximate the optimal value function and control policy.

Give a vague overview of iGame*, dispersion, the values each state carries around...iGame*, that runs even faster by utilizing cascade update rules. Instead of updating every sampled point when a new one is generated, iGame* maintains a set of directed trees where the new point is inserted as a child of the nearest, lowest value neighbor within a defined ball distance. Children are decided with an updated VI function VI*. Updates are only triggered when a node's child changes their estimates or is added, or if an update has not been done for a defined amount of consecutive iterations.

As input, iGame* requires: I, the number of iterations to run, M, the, l, the Lipschitz constant of the dynamical system equation, alpha, any positive scalar to be used in the calculation of the time discretization, D, the maximum number of iterations allowed before a state must have its value updated, and (Beta), a constant used in the generation of the dispersion of the state space. The algorithm starts by generating an initial set of random samples in the state space X, S0. Each of these initial states are assigned a random value between 0 and 1 and their update Flag is set to 0. The estimated values of each state also are assigned 0.

On each step of the remainder of the n iterations, a new sample is uniformly randomly generated within X and added to the existing set of states S. Based on the current iteration and input constant (Beta) a new dispersion value, d_n, is calculated. d_n is considered the resolution of the finite grid that the elements of S_n lie on, and can be defined as: for any x (in) X_{free} (exists) x' such that distance between x and x' is less than d_n. The authors provided a lemma defining an upper and lower bound for the value of d_n, and chose to calculate d_n as the lower bound as it could be calculated offline; the math is reflected in line 8 of the algorithm. From d_n and the input values of M and l, values for the time discretization, h_n, and dilation size, α_n are also calculated that are used for determining how long a timestep is for the purposes of calculating controls and determining a search radius for neighbors around a state, respectively.

After calculating all the necessary variables, iGame* moves into several loops that modify the values of states in S depending on various factors. First, existing states are checked if they need to be updated. For each state x in S_{n-1}, newest sampled state excluded, if x has reached the maximum number of iterations since updating, or the child of x has just been updated, add x to the set of states that require updates, Kn. Next, for the states in Kn that are *not* within X_{goal} values are updated according to the VI* algorithm, which solves a single step game. The algorithm of VI* is detailed in fig. 3.

Within VI*, the single step game at the current point is solved. The solution to this game is to discover which state in S, within a reachable distance, is of best value to move to considering both the controls of the evader and the pursuer. To do this, a maximin function searches for a saddle point that minimizes the estimated value with respect to the evader's available controls while maximizing it with respect to the pursuer's controls. Two separate sets of controls choices are available to the evader and pursuer. It does not matter whether the maximin function makes a list of the evader's or the pursuer's controls choices first, as either way the same saddle point will be discovered (Isaacs). In the original VI* algorithm, a new possible control value is added to the controls set for the evader every iteration, but to reduce the computation time a subset of the control space is sampled once and this discretization is used for the remainder of the algorithm.

```
Algorithm 2 VI*($S_n, \tilde{v}_{n-1}$)
1: $U_n \leftarrow U_{n-1} \cup \mathrm{Sample}(U, 1)$;
2: $v_n(x) \leftarrow 1 - e^{-\kappa_n} + e^{-\kappa_n} \max_{w \in W} \min_{u \in U} \min_{y \in B(x + h_n f(x,u,w), \alpha_n)} \tilde{v}_{n-1}(y)$;
3: $(w_n(x), u_n(x)) \leftarrow$ the solution to $(w, u)$ in the above step;
4: $\mathrm{Child}(x) \leftarrow \mathrm{argmin}_{y \in B(x + h_n f(x, u_n(x), w_n(x)), \alpha_n) \cap S_n} \tilde{v}_{n-1}(y)$;
```

Figure 3: The VI* algorithm. The algorithm solves a maximin function for a single point in a game's space, leading to an update to the assigned value and controls of that state (Mueller, Erich, et al. "Anytime Computation Algorithms for Approach-Evasion Differential Games").

After calculating a solution to the single-step games at each of the state points that needed their value updated, iGame* continues to update the values of states based on two additional criteria

The next loop updates the states in S that are not in Kn or Xgoal. Each of these states have a child state assigned to them such that the state's child is the state within the defined ball distance with the lowest value. The new value of the chosen state will match the previous value of the child. Note that in this loop no controls are calculated to navigate between this state and its new child. Whatever controls are calculated for the state in VI* remain the same. The iterations until update flag is increased, and iGame* moves on to its final loop.

Finally, iGame* considers the last subset of states in S, those that are within a single step of Xgoal. These states follow a simple update rule; their next value is equal to the previous estimated value. The iGame* pseudo code is outlined in fig. 4.

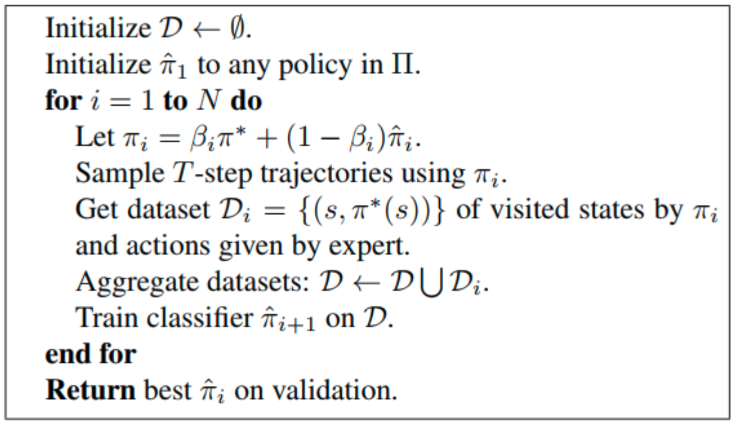
```
Algorithm 1 iGame* $(I, M, \ell, \alpha, D)$
for $x \in S_0 \subset X$ do
    $v0(x) =$Sample$([0,1])$;
    Flag$(x) = 0$;
$n \leftarrow 1$;
while $n < I$ do
    $y_n \leftarrow$Sample$(X, 1)$;
    $S_n \leftarrow S_{n-1} \cup y_n$;
    $d_n \leftarrow \beta * 1/n^2$;
    $h_n \leftarrow d_n^{1/1+\alpha}$;
    $\alpha_n \leftarrow 2d_n + \ell h_n d_n + M\ell h_n^2$;
    for $x \in S_{n-1}$ do
        if Flag$(x) == D$ then
            $K_n \leftarrow K_n \cup x$;
            Continue;
        for Flag(Child$(x)) == 0$ do
            $K_n \leftarrow K_n \cup x$;
    for $x \in K_n \backslash B(X_{goal}, Mh_n + d_n)$ do
        $(v_n(x), u_n(x)$,Child$(x)) \leftarrow VI^*(S_n, \tilde{v}_{n-1}(y))$;
        Flag$(x) \leftarrow 0$;
    for $x \in S_n \backslash (K_n \cup B(X_{goal}, Mh_n + d_n))$ do
        $z \leftarrow$argmin$_{y \in B(x, \alpha_{n-1}) \cap S_{n-1}} v_{n-1}$;
        $V_n(x) \leftarrow v_{n-1}(z)$;
        Child$(x) \leftarrow z$;
        Flag$(x) \leftarrow$Flag$(x) + 1$;
    for $x \in S_n \cap B(X_{goal}, Mh_n + d_n)$ do
        $v_n(x) \leftarrow \tilde{v}_{n-1}(x)$;
        Flag$(x) \leftarrow$Flag$(x) + 1$;
```

Figure 4: The algorithm for iGame*, a iterative sampling discrete approximator that solves differential games (Mueller, Erich, et al. "Anytime Computation Algorithms for Approach-Evasion Differential Games").

When iGame* has finished running, it produces a set of states with attached information. Each sampled state has an associated value and calculated control value that is saved in a python dictionary and written to file. The values of each state can be used to visualize the value function of the game and verify its correctness, however, it is the saved control values that are used to train the imitation learner to produce a similar policy.

3.3 Deep Imitation Learner Setup

Once iGame* has generated its set of sampled state and control inputs, an imitation learner can be trained to model the discrete policy created by iGame* in continuous space. As mentioned previously, DAgger is the framework that was chosen to train a model on the policy. Its pseudo code is included in fig. 5 but the algorithm is also explained briefly. To train a policy that will perform sequence prediction without dooming a model to being unable to recover from a mistake, DAgger starts by initializing a dataset of collected trajectories $\mathcal{D}$. This set is initially empty but is filled with trajectories generated from the first provided policy, $\hat{\pi}_1$, the expert's. In this case, iGame*'s policy will provide the expert policy. The next policy, $\hat{\pi}_2$, is produced by a student classifier trained to mimic the trajectories generated by the expert. This new policy is used to generate more trajectories to add to what already exists in $\mathcal{D}$. For each iteration n DAgger repeats this process, using $\hat{\pi}_n$ to generate more trajectories for $\mathcal{D}$ then training the student to create the next policy, $\hat{\pi}_{n+1}$, to mimic the whole dataset $\mathcal{D}$. When all iterations are completed, the final policy has been trained on the aggregate dataset of all previously generated trajectories.



```
Initialize $\mathcal{D} \leftarrow \emptyset$.
Initialize $\hat{\pi}_1$ to any policy in $\Pi$.
**for** $i = 1$ **to** $N$ **do**
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
    Sample $T$-step trajectories using $\pi_i$.
    Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by $\pi_i$
    and actions given by expert.
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
    Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
**end for**
**Return** best $\hat{\pi}_i$ on validation.
```

Figure 5: Pseudo code for the dataset agregator algorithm, DAgger. DAgger trains a policy on iterations of generated trajectories (Ross, Stephane, et al.).

The classifier mentioned in the pseudo code is the student attempting to learn the policy, and any choice of student for a classifier would suffice. A feedforward, deep neural network, with three fully connected linear layers is used as the student for both of the test cases presented. It is implemented using the pytorch machine learning library in Python. The network uses MSE as its loss function and the Adam optimizer provided by pytorch, and for both of the test cases trains for 2000 epochs with a minibatch size of 64. The final output of DAgger is a model of the policy generated by iGame*. The model accepts as input a state in the game space and outputs its learned control inputs for the evader and pursuer from that state.

3.4 Performance Guarantees

The combination of iGame* and DAgger algorithms results in a system that generates and learns policies for differential games; these learned policies will be near optimal due to the performance guarantees of both algorithms. While iGame* is proven asymptotically optimal in running time, it is also shown to globally converge from any initialized values of the initial random sampling of states. From two assumptions about the input and setup of iGame*, assumptions 2.1 and 3.1, the authors propose their theorem of convergence (Mueller, Erich, et al.). The following properties are said to hold for 2.1:

1. the sets of positions and controls, X, U, and W, are compact

2. the function f that defines the dynamical system is Lipschitz continuous in x for any (u,w) ∈ U × W

3. for any pair of x ∈ X and u ∈ U, F(x,u) is convex where the set-valued map $F(x,u) \triangleq \bigcup_{w} \in W\ f(x,u,w)$

Assumption 3.1 requires the definition of an integer D, D ≥ 0, such that each state sample invokes VI* at least once every D+1 iterations and for any iteration n ≥ 1, the set of sampled states $S_n \subseteq \bigcup_{s=0}^{D} K_{n+s}$. Given these two assumptions, it holds that the sequence of values v_n converges to the optimal value function v* for any x ∈ X: $v^*(x) = \lim_{n \to +\infty} \frac{min}{y \in B(x,d_n) \cap S_n} v_n(x)$. As the number of iterations approaches the limit, the difference between v* and v_n goes to 0. As such, when run for a significant amount of iterations, iGame* will produce a near optimal value function and control policy.

DAgger has its own performance guarantees which bound the loss and regret, or the variation of distance between states observed by the expert and learned policies in hindsight. Lemma 4.1 in the DAgger publication defines an inequality that bounds the distance between state distributions d over T steps, given that π will execute $\hat{\pi}$ over T steps with probability $(1-\beta_i)^T$: $\left\|d_{\pi i} - d_{\hat{\pi} i}\right\|_i \leq 2T\beta_i$. This lemma is used in their proof of theorem 4.1; there exists a policy $\hat{\pi}$ in DAgger,

$$\hat{\pi} \in \hat{\pi}_{1:N} \ \ s.t \ \ E_{s \sim d_{\hat{\pi}}}[\ell(s,\hat{\pi})] \leq \varepsilon_N + \gamma_N + \frac{2\ell_{max}}{N}[n_\beta + T\Sigma_{i=n_{\beta+1}}^{N}\beta_i] \ ,$$

where $E\,[\ell(s,\hat{\pi})]$ is the loss, ε_N is the loss of the best policy in hindsight, γ_N is the average regret of all generated policies, ℓ_{max} is the upper bound on loss, and n_β is the largest iteration such that $\beta_n > \frac{1}{T}$ (Ross, Stephane, et al.). For any input distribution, some policy achieves a surrogate loss of ε under its own state distribution in the limit. The authors also address a finite sample case wherein a similar bound of the loss is calculated. With iGame* able to produce a near optimal policy, a DAgger taught imitation learner will be able to imitate the same policy with minimal regret.

4. Test Cases

4.1 Fence Escape Game

Our first attempt to imitate iGame*'s policy was with a pursuit-evasion fence escape game. This game was selected first as its state space is simple and the solution is easy to observe without involving any pre-calculations. Two players start the game at random points in a plane along a fence of some defined length. The evader wants to reach the end of the fence and escape, but the pursuer can prevent its escape by remaining within a set capture distance from the evader. Figure 6 is an illustration of the setup of the fence escape game setup.

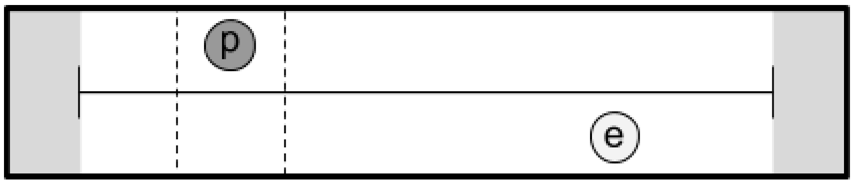


Figure 6: The fence escape game has two players, a pursuer p and evader e. The evader's goal is to get around the fence to the grey shaded X_{goal} without being within the dashed capture zone of the pursuer.

The pursuer and evader system is defined by the following differential equations:

x_p'(t) = x_p(t)*w(t)

x_e'(t) = x_e(t)*u(t),

where x_p and x_e are the positions of the pursuer and evader and w and u are their respective controls. A state of the game at time t is represented as the pair of position points of x_p and x_e. Each player directly controls their velocities,w = vel_p and u = vel_e. Both players move only along the x axis in either the positive or the negative direction; the controls values for velocity are constrained between [-1,1].

This implementation of the fence escape game used a fence of length 10 where the fence starts at position 0 and ends at 10. The players' states are confined between -1 and 11. iGame* variables. DAgger is configured to run trajectory length, how many initial trajectories and rounds to run, and how many rounds and trajectories per round are run after the initial trajectories.

4.2 Merging Cars Game

A pursuit-evasion game with more complex states was our second test of policy imitation learning. This merging game is a model of two vehicles in an everyday situation, the evader is merging onto a single-lane highway that already has the pursuing vehicle on it. In this situation, the evader wants to reach the gray X_{goal} without colliding with the pursuer after exiting the on-ramp. The pursuer is attempting to accomplish the worst case scenario of trying to run into the evader. Figure 7 illustrates the merging game.

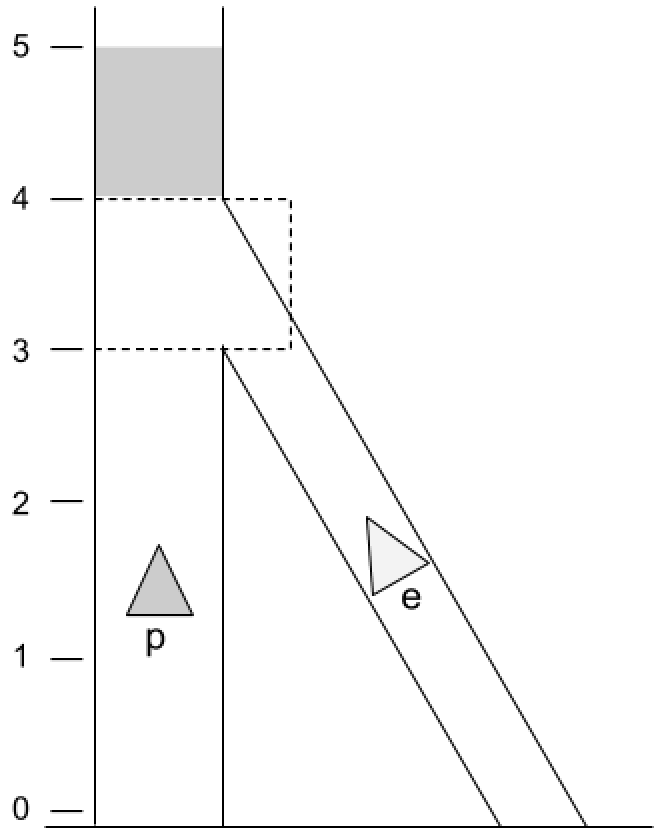


Figure 7: The merging game models an interaction between two cars, a pursuer p, already on the highway, and a merging evader, e. The evader wishes to merge

successfully by proceeding to the grey shaded X_{goal}, located after position 4, without colliding with the pursuer between positions 3 and 4.

The dynamical system of the merging game is defined as:

x_p'(t) = x_p(t)*v_p(t)

x_e'(t) = x_e(t)*v_e(t)

v_p'(t) = v_p(t)*w(t)

v_e'(t) = v_e(t)*u(t).

States for the game are made up of sets of 4 variables, the positions and velocities of the evader and pursuer [x_p, v_p, x_e, v_e]. Both evader and pursuer control their acceleration, w = accel_p and u = accel_e, constrained between values of [-0.2, 1.2]. Several state constraints apply to both players as well. Vehicles' positions must be between 0 and 5, and their velocities cannot exceed 1.2 or be lower than -.2. Since the horizontal position of the vehicles is controlled by the road, it is sufficient to model vehicle position with a single variable.

The merging game setup is not reflective of the complexity of real life environments, but the increase in complexity from the fence escape game to the merge game shows that our imitation learner can generalize to more difficult to solve cases.

5. Evaluation

5.1 Fence Escape Game

To ensure a simple policy was learnable by the imitation student, an instance of the iGame* algorithm was created utilizing the dynamics of the fence escape game. The dynamical system is used to construct the value function and policy in the form of labeled states. Each sampled state has a corresponding value and control inputs stored with it in a python style dictionary. The output of iGame* in the form of an approximated value function is shown in fig. 8, where the values of each state are plotted with respect to the pursuer and evader positions. A line appears through the center of the diagram marking the states from which the evader will lose. Based on this value function, iGame* generates a control policy. The paired states and control inputs are used to train the imitation student.

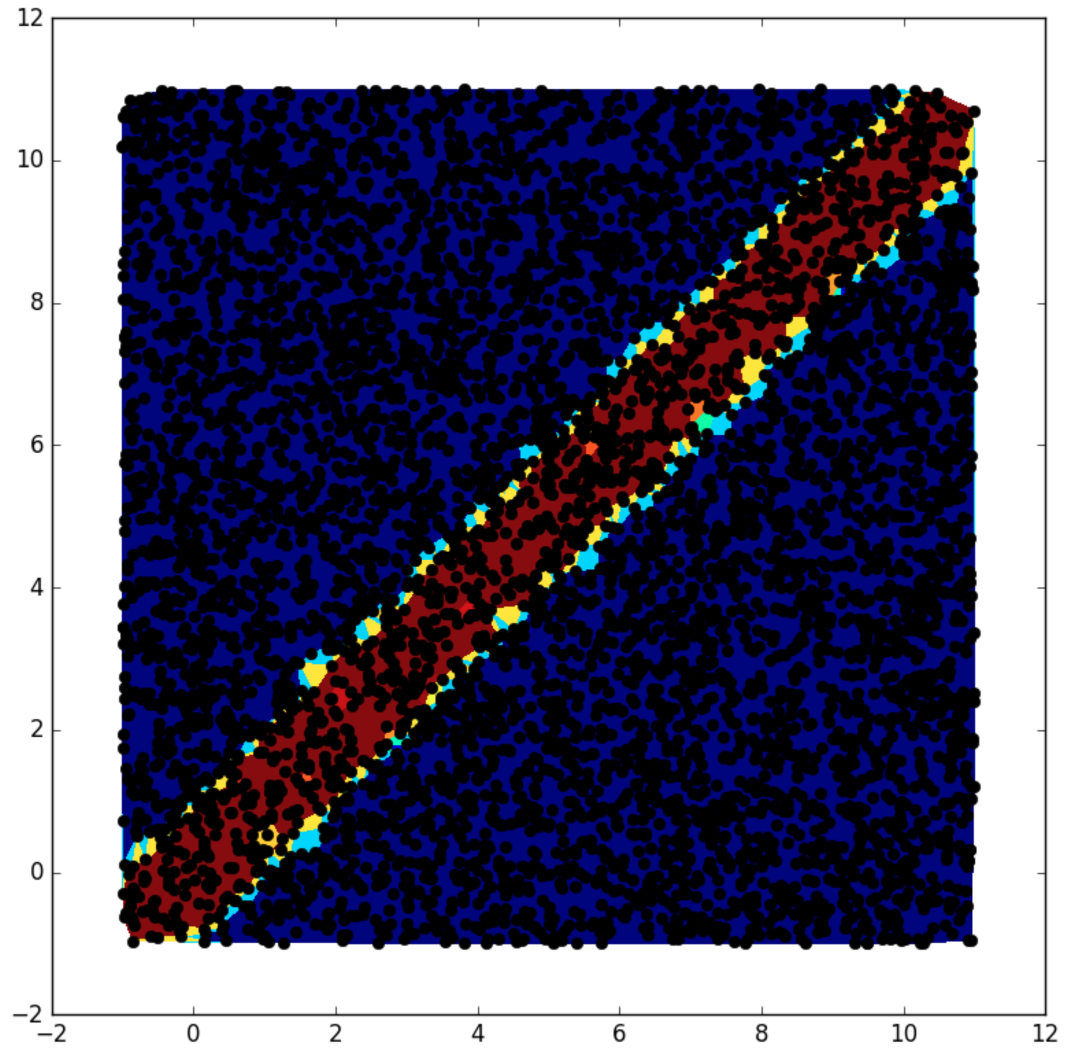


Figure 8: The estimated value function for the fence escape game. The x-axis is the position of the pursuer and the y-axis is the position of the evader. A clear line is observed through the states in the center of the diagram where the value is 1 and the evader loses the game.

With DAgger configured to run using 20 total rounds and generating 100 trajectories per round, the deep learning student was taught the generated policy. The student produces comparable outputs, and manages to do so with considerable speed up from referencing iGame*'s dictionary of states directly. On average in simulation, generating pursuer and evader control inputs from iGame*'s policy requires 83.1 milliseconds to complete a game, while it takes the imitation student 41.8ms, making it 49.7% faster. Some of the speed increase may be due to the implementation of iGame* as a dictionary lookup in Python; however, the more complex merging game shows that the imitation student is able to generalize in the increased state space and compress the policy, reducing the time required more than iGame* is able.

Additionally for this game, reinforcement learning was applied as a second machine learning algorithm to further optimize the control policy. Using Proximal Policy Optimization (PPO), a model was learned directly from playing the game from the perspective of the evader(Schulman, John, et al.). The resulting model won the winnable games it played 95% of the time. Although PPO specifically addresses how difficult it can be to tune reinforcement learning algorithms, some tuning is still necessary, and additional adjustments could yield a 100% accurate model. With such a model, the DAgger learner would have additional, superhuman performance policy data to learn from.

5.2 Merging Cars Game

With the system successfully learning a simple fence escape game, it was expanded to learn the merging game to see the effect of its more complicated state and control space. Like the fence escape game, the merging game's dynamics were used in iGame* to construct the value function and policy. A graph of the approximated value function produced is provided in fig. 9. As in the fence escape game, the value of states where the evader loses is indicated in the middle of the graph. DAgger was set up with similar settings to the previous game, with the exception of using a neural network with a different input size due to the increased state sizes of the merging game.

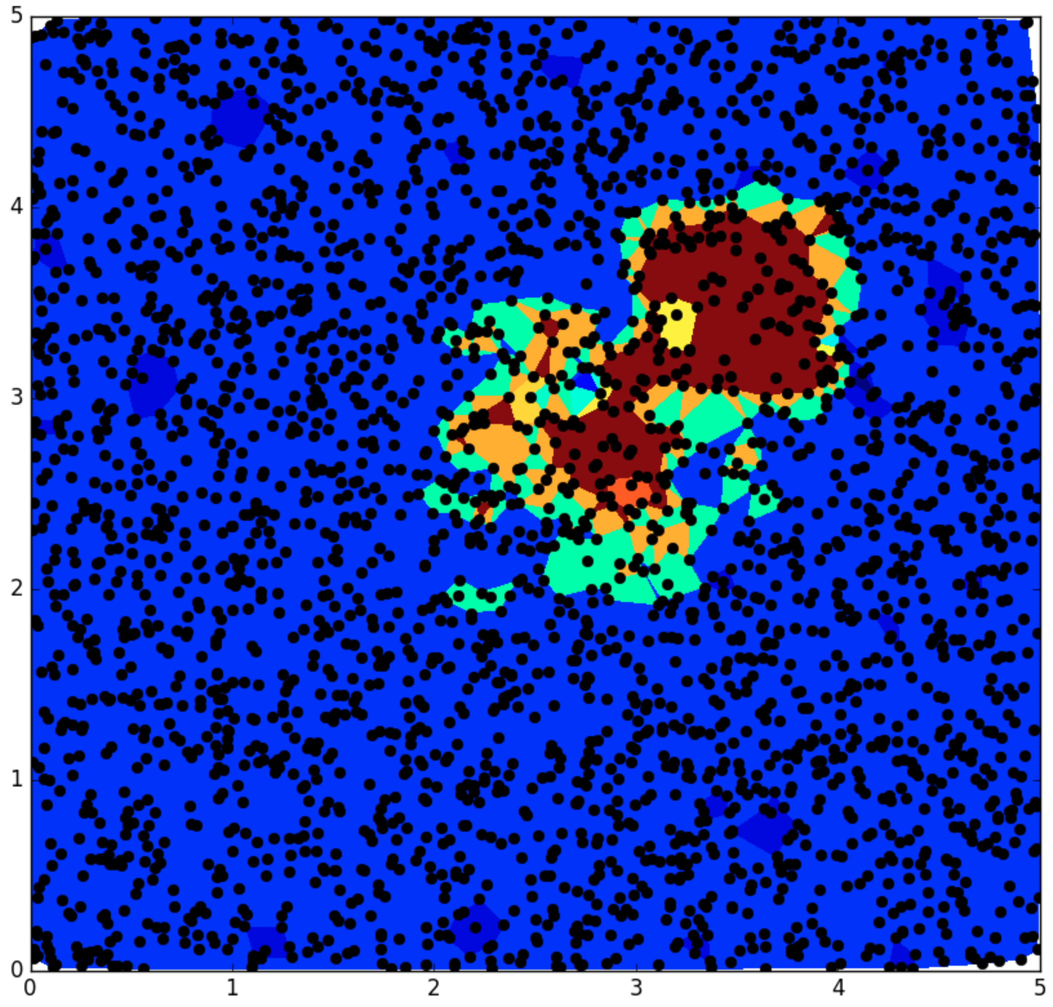


Figure 9: The value function for the merging cars game. The x axis represents the position of the pursuer and the y axis the evader. There is a clear collision zone between positions 3 and 4, and states between positions 2 and 3 where collisions become unavoidable.

The merging imitation student was also able to learn iGame*'s policy and produces comparable control values. Both iGame*'s policy and the student completed merging simulations more slowly than the fence escape game, but iGame* experienced considerably more slowdown. iGame*'s policy requires on average 147.6ms to complete the simulation while fetching the control values, while the imitating student only required 61.4ms to generate its solutions (a 58% decrease in required time). Interestingly, the runtime of referencing the plain policy increased 77.6% compared to the student's increase of 46.8%, showing that the imitation learner much more efficiently generalizes to complex environments, such as those found on autonomous vehicles. The large increase in runtime compared to the fence escape game is likely do to searching through the much larger dictionary of states required in the more complicated state space of the merging game. The student compresses the policy and reduces the effect of the increased state size.

6. Future Work and Discussion

In both test cases, the imitation learner was able to model the policy created by iGame*. Going forward however, there are improvements and extensions to the system that would make it more applicable to an autonomous vehicle. The resulting policy learned by the student relies, like its game theory based expert, on having a full knowledge of the state space in order to derive the correct controls. Tracking neural nets exist that are feasible for use in a real time environment, with the ability to track objects at upwards of 100fps (Held, David, et al.). Any error in tracking, however, that would translate into incorrect state information would cause this system to generate incorrect controls. An imitation learner utilizing input remapping, or learning the correlation between controls and the image input directly, would avoid this issue.

iGame* itself is also very sensitive to its initial parameters. The value of the dispersion at the start of the algorithm, for instance, is determined by the user, and it must be not too small or too large in order for states to be able to find their neighbors but not consider too much of the state space. Dispersion values that are improperly scaled result in inaccurate value functions. It is possible to compute a good starting value of the dispersion for less complex games, such as fence escape, but for state spaces more complex than the merging game calculation becomes less intuitive. A more efficient way to estimate a good value for the initial dispersion would decrease the time spent tuning iGame*, and would increase the accuracy of its final policy.

7. Conclusion

We set up two test cases to test the ability of imitation learning to model a discretely approximated policy as a continuous control policy. The fence escape game showed that our system architecture could learn a policy generated by iGame* and the merging game showed that the process is generalizable to more complex environments. Modeling situations faced by autonomous vehicles as differential games provides a vehicle with measures to predict how other agents in its environment may react to its own actions and goals so that their future positions can be estimated. This knowledge allows a vehicle to make control decisions to avoid future collisions. Although the environment faced by an autonomous vehicle may be complex, the iterative sampling methods used in iGame* should be able to generate the value function given sufficient processing time, and the resulting policy can be modeled with imitation learning to drastically decrease runtime.

Bibliography

Cardaliaguet, Pierre, et al. “Set-Valued Numerical Analysis for Optimal Control and Differential Games.” *Stochastic and Differential Games*, 1999, pp. 177–247., doi:10.1007/978-1-4612-1592-9_4.

Chen, Zhilu, and Xinming Huang. “End-to-End Learning for Lane Keeping of Self-Driving Cars.”*2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, doi:10.1109/ivs.2017.7995975.

Bojarski, Mariusz, et al. “End to End Learning for Self-Driving Cars.” *[1604.07316] End to End Learning for Self-Driving Cars*, Cornell University Library, 25 Apr. 2016, arxiv.org/abs/1604.07316.

Driggs-Campbell, Katherine, and Ruzena Bajcsy. “Identifying Modes of Intent from Driver Behaviors in Dynamic Environments.” *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, 2015, doi:10.1109/itsc.2015.125.

Exarchos, Ioannis, et al. “On the Suicidal Pedestrian Differential Game.” *Dynamic Games and Applications*, vol. 5, no. 3, Feb. 2014, pp. 297–317., doi:10.1007/s13235-014-0130-2.

Goodfellow, Ian, et al. *Deep Learning*. The MIT Press, 2017.

Gu, Shixiang, and Luca Rigazio. “Towards Deep Neural Network Architectures Robust to Adversarial Examples.” *Towards Deep Neural Network Architectures Robust to Adversarial Examples*, Cornell University Library, 9 Apr. 2015, arxiv.org/abs/1412.5068v4.

Held, David, et al. “Learning to Track at 100 FPS with Deep Regression Networks.” *Computer Vision â€□ ECCV 2016 Lecture Notes in Computer Science*, 2016, pp. 749–765., doi:10.1007/978-3-319-46448-0_45.

Ho, Yu-Chi, et al. “Differential Games and Optimal Pursuit-Evasion Strategies.” *IEEE Transactions on Automatic Control*, IEEE., 1965, pp. 385–389.

Huang, Sandy, et al. “Adversarial Attacks on Neural Network Policies.” *[1702.02284v1] Adversarial Attacks on Neural Network Policies*, Cornell University Library, 8 Feb. 2017, arxiv.org/abs/1702.02284v1.

Isaacs, Rufus. *Differential Games: a Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimisation*. Dover Publications, 1999.

Jiang, Frank, et al. “Using Neural Networks to Compute Approximate and Guaranteed Feasible Hamilton-Jacobi-Bellman PDE Solutions.” *[1611.03158] Using Neural Networks to Compute Approximate and Guaranteed Feasible Hamilton-Jacobi-Bellman PDE Solutions*, Cornell University Library, 27 Mar. 2017, arxiv.org/abs/1611.03158.

Karaman, Sertac, and Emilio Frazzoli. “Sampling-Based Algorithms for Optimal Motion Planning.” *The International Journal of Robotics Research*, vol. 30, no. 7, 2011, pp. 846–894., doi:10.1177/0278364911406761.

Kochenderfer, Mykel J., Jessica E. Holland, and James P. Chryssanthacopoulos. *Next-generation airborne collision avoidance system*. Massachusetts Institute of Technology-Lincoln Laboratory Lexington United States, 2012.

Lachner, R. “Collision Avoidance as a Differential Game: Real-Time Approximation of Optimal Strategies Using Higher Derivatives of the Value Function.” *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, vol. 3, Oct. 1997, pp. 2308–2313., doi:10.1109/icsmc.1997.635270.

Lewin, J., and G. J. Olsder. “Conic Surveillance Evasion.” *Journal of Optimization Theory and Applications*, vol. 27, no. 1, 1979, pp. 107–125., doi:10.1007/bf00933329.

Mitchell, Tom M. "Machine learning. WCB." (1997).

Morrison, David D., et al. “Multiple Shooting Method for Two-Point Boundary Value Problems.”*Communications of the ACM*, vol. 5, no. 12, Jan. 1962, pp. 613–614., doi:10.1145/355580.369128.

Mueller, Erich, et al. “Anytime Computation Algorithms for Approach-Evasion Differential Games.” *[1308.1174v2] Anytime Computation Algorithms for Approach-Evasion Differential Games*, Cornell University Library, 28 Sept. 2013, arxiv.org/abs/1308.1174v2.

Mueller, Erich, et al. “Anytime Computation Algorithms for Stochastically Parametric Approach-Evasion Differential Games.” *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, doi:10.1109/iros.2013.6696902.

Neumann, John Von, and Oskar Morgenstern. *Theory of Games and Economic Behavior*. University University Press, 1953.

Pachter, M., and Y. Yavin. “A Stochastic Homicidal Chauffeur Pursuit-Evasion Differential Game.” *Journal of Optimization Theory and Applications*, vol. 34, no. 3, 1 July 1981, pp. 405–424., doi:10.1007/bf00934680.

Ross, Stephane, et al. “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning.” *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*, Cornell University Library, 16 Mar. 2011, arxiv.org/abs/1011.0686.

Royo, Vicenç Rubies, and Claire Tomlin. “Recursive Regression with Neural Networks: Approximating the HJI PDE Solution.” *[1611.02739] Recursive Regression with Neural Networks: Approximating the HJI PDE Solution*, Cornell University Library, 23 Mar. 2017, arxiv.org/abs/1611.02739.

Selten, R. “Reexamination of the Perfectness Concept for Equilibrium Points in Extensive Games.” *International Journal of Game Theory*, vol. 4, no. 1, 1975, pp. 25–55., doi:10.1007/bf01766400.

Schulman, John, et al. “Proximal Policy Optimization Algorithms.” [1707.06347] Proximal Policy Optimization Algorithms, 28 Aug. 2017, arxiv.org/abs/1707.06347.

Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: an Introduction*. The MIT Press, 2012.

Szegedy, Christian, et al. "Intriguing properties of neural networks." *arXiv preprint arXiv:1312.6199* (2013).

Wang, Pin, and Ching-Yao Chan. “Formulation of Deep Reinforcement Learning Architecture Toward Autonomous Driving for On-Ramp Merge.” *[1709.02066] Formulation of Deep Reinforcement Learning Architecture Toward Autonomous Driving for On-Ramp Merge*, Cornell University Library, 7 Sept. 2017, arxiv.org/abs/1709.02066.

Xu, Huazhe, et al. “End-to-End Learning of Driving Models from Large-Scale Video Datasets.”*2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, doi:10.1109/cvpr.2017.376.