CS230

Trajectory generation for autonomous vehicles in racing using Deep Neural Networks

Mingyu Wang
Department of Mechanical Engineering
Stanford University
mingyuw@stanford.edu

Abstract

The goal of this project is to generate trajectories for autonomous vehicles in competitive scenarios using deep neural networks. A supervised learning approach is used in this project. We generate training data using an iterated optimization approach as introduced in [1] in simulation. A fully connected network takes vehicle states as input and the output is a trajectory represented by positional waypoints. The performance of the algorithm is validated on a test dataset with mean squared error (MSE) metric. It is shown that the deep learning method greatly reduces computation time, which is vital for real-time control systems, while achieve promising motion planning performance.

1 Introduction

In this project, we use deep learning to train a motion planning controller in game-theoretic scenarios.

Specifically, we wanted to consider drone racing games. Professional players usually have various racing strategies like blocking, cutting or overtaking during the game. Such strategies depend on the player's prediction on its opponent's future trajectory and drone dynamics. To achieve expert level, human need extensive training and years of racing experience. However, with the rising attention on autonomous drone research, autonomous drone racing games have been gaining more and more attention [2, 3].

In these games, autonomous racing vehicles compete on a previously unknown track. The autonomous drones are usually equipped with multiple sensors and actuators for perception and control tasks. The goal of the game is to finish the race in the first place while ensure safety. Currently, most autonomous drone racing games does not consider the case where multiple players compete on the track simultaneously. Instead, players race on the track individually and lap times are recorded. In our previous work, a game-theoretic motion planning algorithm was proposed to generate trajectories in a two-vehicle racing game. The planner takes the state of both ego vehicle and its opponent and track geometry as input, then applies an iterated best response (IBR) algorithm to approximate the Nash equilibrium of the game. The objective is to advance along the track as far as possible with respect to the opponent, while keep in track and avoid collisions with its opponent. The planner has demonstrated rich racing behavior such as blocking while necessary and overtaking. The challenging part of this algorithm is that the nonlinear optimization problem in each iteration is time consuming. To achieve real-time control, the number of iterations has to be limited and thus compromise accuracy.

In this project, we use the generated trajectories from the game-theoretic planner as training data. Then, we apply imitation learning approach to train a deep neural network from "expert" data. Here,

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

a supervised learning approach is used. We analyze the performance of the proposed approach using mean square error (MSE) metric on test dataset.

2 Related work

There are large amount of literature on motion planning with learning-based approaches[4]. One important intuition is to emulate human/expert behavior. For example, inverse reinforcement learning (IRL) [5] has been applied to learn an objective function from demonstrations. However, since human are inherently stochastic, it is challenging to generalize to different scenarios. Generative adversarial imitation learning formulates the problem as matching the state distribution of expert policy [6, 7]. A generator attempts to emulate expert behavior while a discriminator learns to distinguish expert policy from non-expert policies.

In our work, due to time constraints, we limit the scope of the project to supervised learning approach, also called behavior cloning (BC). A deep neural network learns to emulate expert behavior by minimizing the expectation of some cost function $l(\pi, s)$ over state distribution induced by expert policy.

3 Problem Statement

Consider two players racing in a known track as shown in Fig. 1a. For simplicity, each player is modeled as a single integrator whose kinematics is given as

$$\boldsymbol{p}_{t+1} = \boldsymbol{p}_t + \boldsymbol{u}_t,$$

where $\boldsymbol{u} \in \mathbb{R}^2$ is control input and $\boldsymbol{p} \in \mathbb{R}^2$ is position in 2D space. The objective of each player is to make progress along the track as far as possible with respect to its opponents, while avoiding collisions with each other. More formally, each player solves the following problem in a receding horizon fashion.

$$\begin{aligned} \max_{\boldsymbol{\theta}_i} \; & s_i(\boldsymbol{p}_i^N) - s_j(\boldsymbol{p}_j^N) \\ \text{s.t.} \quad & h_i(\boldsymbol{\theta}_i) = 0, \\ & g_i(\boldsymbol{\theta}_i) \leq 0, \\ & \gamma_i(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) \leq 0, \end{aligned}$$

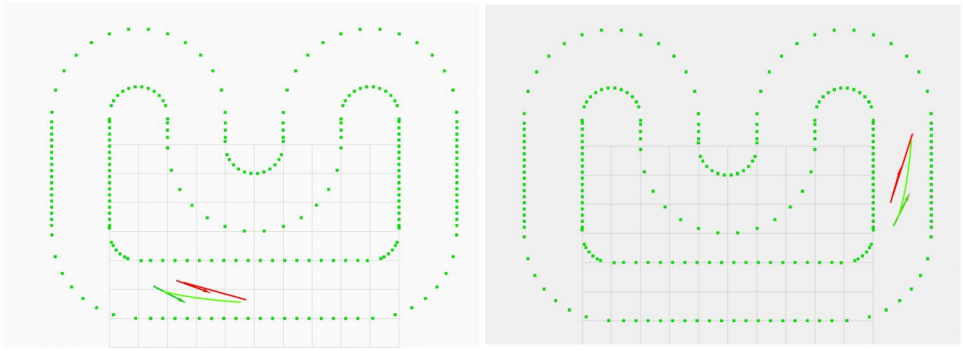
where $\boldsymbol{\theta}_i$ represents the decision variables for player i, i.e. $\boldsymbol{\theta}_i = [\boldsymbol{p}_i^0, \boldsymbol{p}_i^1, \ldots, \boldsymbol{p}_i^N]$ for N planning steps and $\boldsymbol{p}_n \in \mathbb{R}^2$s are positional waypoints. h_i includes equality constraints such as dynamic constraints. g_i denotes inequality constraints which involves only one player such as track constraints, control input constraints, etc. γ_i represents inequality constraints which involves both players, such as collision avoidance constraints.

The problem is challenging because in order to enforce safety during planning, one needs to know the future trajectory of others ($\boldsymbol{\theta}_j$) which is generally not available. Thus, we resort to the concept of Nash equilibria. A Nash equilibrium is a pair of solutions such that no single agent can increase its payoff by unilaterally changing its strategy. Formally, a Nash equilibrium is a pair of solution $(\boldsymbol{\theta}_i^*, \boldsymbol{\theta}_j^*)$ such that

$$\begin{aligned} \boldsymbol{\theta}_i &= \arg\max_{\boldsymbol{\theta}_i} s(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j^*), \\ \boldsymbol{\theta}_j &= \arg\max_{\boldsymbol{\theta}_j} s(\boldsymbol{\theta}_i^*, \boldsymbol{\theta}_j). \end{aligned}$$

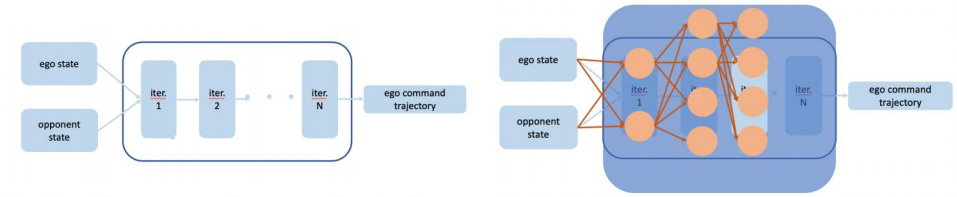
We aim to seek a Nash solution in this problem. Specifically, our planner takes the state of both ego agent and its opponent's as input, and plans a trajectory with limited horizon ($5\,\mathrm{sec}$) for the controlled agent taking into account the opponent's reaction to its motion. In general, Nash equilibria are hard to solve. Here, an iterative method is applied which solves a non-linear optimization problem for both players iteratively. Some of the example planned trajectories are illustrated in Fig. 1.

Ideally, the algorithm should be run in real-time system at $100\,\mathrm{Hz}$. However, since the planning process requires solving several iterations of non-linear optimization problem, we often have to compromise solution accuracy for computation time. On the other hand, computing the trajectories



(a) Blocking scenario 1. (b) Blocking scenario 2.

Figure 1: Two examples of the generated trajectory (training dataset). The leading vehicle is actively blocking the following vehicle to ensure maintaining the leading position.



(a) Algorithm structure for generating the dataset. (b) Proposed training architecture.

Figure 2: Algorithm structure.

offline in simulation is low-cost. One way to take advantage of the offline computed data is to store it in a state-action map and query the map whenever an action is needed. We resort to deep learning here to learn the policy from "expert" algorithm given state information.

4 Dataset and Features

To generate the dataset, 10 game iterations are solved in the algorithm described in Section 3 as shown in Fig. 2a. In each iteration, the non-linear constraints and objective are linearized using sequential quadratic programming (SQP). The algorithm is implemented using Gurobi solver in C++ [8]. We collected the following data:

1. Input data: position and velocity of ego car and opponent car $\boldsymbol{p}_{\text{ego}}, \boldsymbol{p}_{\text{opp}} \in \mathbb{R}^2$, $\boldsymbol{u}_{\text{ego}}, \boldsymbol{u}_{\text{opp}} \in \mathbb{R}^2$. There are 8 input states in total. In the comparison method we describe in Section 5, we also augment the dataset with track center information (10 waypoints) or relative position between controlled vehicle and opponent vehicle.
2. Output data: For the output, the generated trajectory is represented by a sequence of 10 waypoints, each of them is a position vector $\boldsymbol{p}_d \in \mathbf{R}^2$.

In this format, we collected 30,000 samples as initial training dataset. 3,000 samples are used as test dataset and 3,000 are used as validation dataset.

5 Methods

The code is available through the following link: https://github.com/cookiew/CS230.

Given the trajectory dataset, we implemented a feedforward neural network with fully connected layers for supervised learning. The network learns to minimize the MSE loss between prediction and

hidden layer	type	# of neurons	activation
1	FC	256	ReLU / LeakyReLU
2	FC	512	ReLU / LeakyReLU
3	FC	512	ReLU / LeakyReLU
4	FC	512	ReLU / LeakyReLU
5	FC	256	ReLU / LeakyReLU

Table 1: feedforward network structure

"expert" data. More explicitly, the neural network learn a policy π such that

$$\pi = \arg\min_{\Pi} \mathbb{E}_s[l(\pi, s)],$$

where s is the game state, whose distribution is induced by expert policy. $l(\pi, s)$ is MSE error between expert trajectory and learned trajectory.

$$l(\pi, s) = \frac{1}{m}\sum_{i=1}^{m}(\pi_i^{*2} - \pi_i^2)$$

The details of network structure is listed in Table 1. A schematic illustration is given in Fig. 2b. As for activation function, we experimented with both ReLU and LeakyReLU after each FC layer. We use MSEloss as the loss function for training. We use Adam optimizer as it performs better than SGD empirically.

6 Experiments and Discussion

6.1 Experiment result

We implemented the network using PyTorch [9] and train the neural network using the following hyperparameters. We use Adam optimizer with learning rate 0.001. Batch size is 500 during training and we train the neural network for 1000 epochs. It is observed that the training loss usually converges after 600 epoch with the above parameters. For the activation function, we experimented both ReLU and LeakyReLU, it is observed that LeakyReLU provides faster training speed as shown in Fig. 3a. The loss function on both training dataset and testdata as a function of number of epochs are shown in Fig. 3a. The loss on test dataset is comparable to the loss on training dataset so there is no bias.

Several examples of training result is plotted in Fig.4a. In the left figure, the result is learned from the dataset generated by expert policy. On the right, we augment the dataset with states generated from learned policy and re-train the network. The reason will be discussed in the next section.

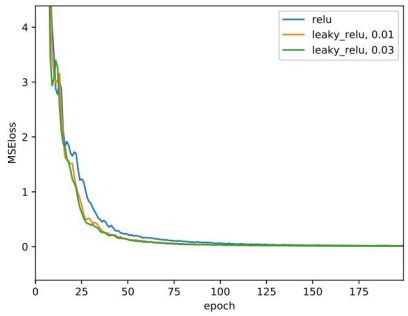
As for computation time, neural network provides a significant advantage. The average query time for the learned policy is $0.6\,\mathrm{ms}$ while solving the non-linear optimization problem for 10 iterations requires $50\,\mathrm{ms}$ on a laptop with Intel i5 processor.

6.2 Discussion

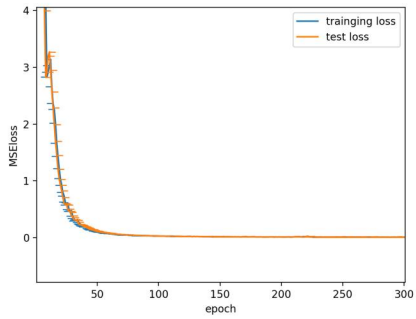
The trained neural network converges well and achieves good performance on the test dataset. However, one significant shortcoming of the approach is that the learned policy is optimized over the state distribution from expert policy. However, when applying the learned policy to real simulation and interacting with the environment, the state distribution comes from network policy. This transition is called "covariate shift" and results in accumulating error which is hard for an agent to recover from. Thus, we further applied DAGGER [4] and trained one iteration using the collected states during real simulation and computes the expert policy of those states. The training results are shown in Fig. 4b. Due to time constraints, we only trained one iteration of DAGGER.

7 Conclusion and Future Work

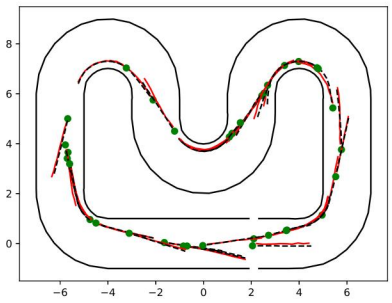
In this project, we trained a deep neural network to learn waypoint trajectories for autonomous vehicles. The approach is evaluated on test dataset using MSE metric. The network learns the



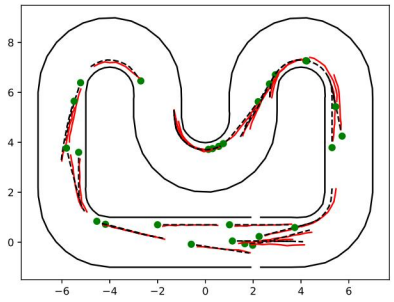
(a) Training results with different activation functions, ReLU, LeakyReLU with negative slop = 0.01 and 0.03. Leaky ReLU leads to a better learning performance in this dataset.



(b) Loss on training and testing datasets during training. Leaky ReLU leads to a better learning performance in this dataset.



(a) Training result using data from expert distribution. The result shown here is after 1000 epochs of training.



(b) Training result using augmented dataset. The new dataset consists of states from expert policy and learned policy. The result shown here is after 1000 epochs of training.

Figure 4: Illustration of performance on test dataset. Green circles are controlled car positions. To avoid cluttering the figures, the positions of the opponent car are omitted. The ground truth trajectory are black dotted lines and predicted trajectories using neural network are red lines.

trajectory planning on the given dataset and greatly reduces computation time, which is crucial for real-time control systems. For future work, we want the further explore DAGGER algorithm and address the covariate shift challenge in imitation learning problems. More specifically, the learned trajectory planner should be able to interact with an opponent in real simulation environment.

References

[1] R. Spica, D. Falanga, E. Cristofalo, E. Montijano, D. Scaramuzza, and M. Schwager, "A real-time game theoretic planner for autonomous two-player drone racing," in *Robotics: Science and Systems*, (Pittsburgh, PA-USA), June 2018.

[2] "Alphapilot ai drone innovation challenge." `https://www.lockheedmartin.com/en-us/news/events/ai-innovation-challenge.html`.

[3] "The drone racing league." `https://thedroneracingleague.com/`.

[4] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635, 2011.

[5] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*, p. 1, ACM, 2004.

[6] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in Neural Information Processing Systems*, pp. 4565–4573, 2016.

[7] R. Bhattacharyya, D. P. Phillips, B. Wulfe, J. Morton, A. Kuefler, and M. J. Kochenderfer, "Multi-agent imitation learning for driving simulation," in *iros*, 2018.

[8] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2018.

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.