Motion Planning for Autonomous Driving

Philippe Weingertner, Minnie Ho

November 14, 2019

1 Introduction

In Autonomous Driving, the behavioral planner defines a short term objective: where the ego vehicle shall go, with associated constraints. The local planner then computes a set of paths that are kinematically feasible and collision-free with respect to static obstacles. For every candidate path, a velocity profile is generated to avoid dynamic obstacles. In this project, we explore ways to generate a velocity profile, assuming the candidate path is given. We begin with a deterministic environment, using a simple rule-based algorithm before moving to Oracle algorithms based on Dynamic Programming and Uniform Cost Search algorithms. We introduce randomness in the form of non-ideal sensors into the environment transforming our problem into an MDP, and we explore methods to produce a velocity profile in real-time.

2 Related Work

There is a rich literature related to Motion Planning and a very detailed survey is provided in [10]. Among the first four successful participants of DARPA Urban Challenge in 2007, the approaches vary, but fundamentally rely on a graph search where nodes correspond to a configuration state and edges correspond to elementary motion primitives. The run-time and state space can grow exponentially large. In this context, the use of an efficient heuristic is important.

More recently, Reinforcement Learning (RL) and Deep RL have been investigated in the context of Autonomous Driving for decision making either at the Behavioural Planning or Motion Planning level. In papers from Volvo [5] and BMW [4], a Deep Q-Network (DQN) RL agent is trained to make decisions by selecting maneuvers. Unfortunately, even if the utility is designed to avoid collisions, for RL this is optimized in expectation only. In [4] an additional safety check layer is added after the DQN agent in case an override decision is needed. Alternatively, safety is enforced before Deep RL. In [1], the agent is constrained to choose among a restricted set of safe actions per state, where the action space is a set of longitudinal accelerations applied along a given path at a T-intersection. Ultimately we may want to combine safety checks before and after an RL agent.

AlphaGo Zero [13] has defeated human world champions, by combining a Monte Carlo tree search (MCTS) with Deep RL. A neural network biases the sampling towards the most relevant parts of the search tree, and a learned policy-value function is used as a heuristic during inference. There are a few differences for Motion Planning. The state space is continuous, not discrete, and only partially observable. Also self-play can not be used. These challenges have been recently tackled in different publications, including [6, 2, 11]. In [11] the Motion Planning problem is addressed in a two-step path-velocity decomposition. First, the path planner employs an extension to the A* algorithm to propose paths that are drivable and collision-free with respect to static obstacles. Secondly, a velocity profile is generated by issuing acceleration commands. The problem is formulated as a Partially Observable Markov Decision Process (POMDP) model and solved with an online DESPOT solver. DESPOT is a sampling based tree search algorithm like MCTS, which uses additional values for lower and upper bounds. A NavA3C neural network is trained to provides tighter bounds to guide the DESPOT tree search. In [6] the problem is considered at the behavioral planning level, where a set of lane changing decisions are taken to navigate a highway and to reach an exit. The problem is formulated as an MDP problem. An MCTS tree search is used as an online MDP solver and a learned policy-value network is used to efficiently guide the search. We consider the two-step path and velocity profile generation, as in [11], but with an algorithm approach similar to [6].

3 Environment and Test Setup

The problem statement is as follows. Given an ego vehicle (E) with a given path of (x, y) coordinates, find a set of acceleration decisions (a_x, a_y) at discrete

time steps to enable E to avoid a set of intersecting vehicles $\{V\}$.

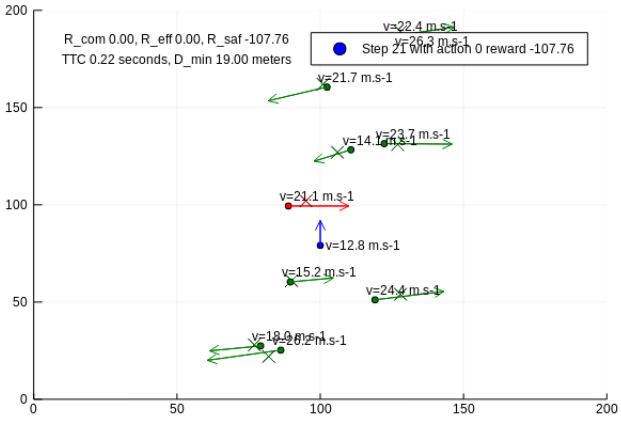

Figure 1: Project Environment using OpenAi Gym

The ego car in blue has to avoid 10 intersecting vehicles to reach a goal point. The position and speed of intersecting vehicles is not known precisely: the ground truth is represented by an 'o'; the position reported by the sensors is represented by an 'x'; and arrows represent the velocity vectors. The minimum Time To Collision (TTC) between the ego car and the set of intersecting vehicles is displayed; if there exists an intersecting car with a computed TTC below 10 seconds, it is displayed in red. We developed our own customized test framework that is compatible with standard openai gym interfaces. Our simulation and test environment can be downloaded and installed from proj-gym-act.

4 Approach

We are dealing with a local planning task where the path to be followed has already been derived by a variational technique or a A* or D* tree search algorithm. Therefore, given a path, we can focus on generating a velocity profile, which consists of a set of acceleration commands we can use to avoid dynamic obstacles that may cross our path. We initially define our environment to be one without real-time constraints, where sensors are perfect, and we know the (randomly initialized) positions and velocities of all the intersecting vehicles.

In our baseline solution, we set a simple, reflex-based rule for decision-making: if the smallest TTC with respect to other cars is below 10 seconds we decelerate by $-2\ m/s^2$ (as long as we see this collision risk). Otherwise we accelerate by $1\ m/s^2$ (as long as we are below the speed limit) to reach our target as fast as possible. Our baseline is clearly a sub-optimal algorithm, as will be seen later in the results.

For our Oracle solution, we implement algorithms such as backtracking search (BS), dynamic programming (DP), and uniform cost search (UCS) to determine the best actions and to minimize a cost that heavily penalizes collisions. In autonomous driving, typical decision update frequencies are on the order of every 250 ms with even stronger requirements in case of emergency situations. We see from our results that our Oracle algorithms, while optimal, cannot provide decisions in an online (real-time) way.

Once we have our baseline, we expand our problem to a more realistic one with uncertainty, where the sensors provide noisy estimates. In this case, our problem becomes a Markov Decision Process (MDP). In addition, we would like decisions to be computable every 100 ms. Hence, we must reduce the search space.

We explore a model-based approach using an online MCTS (sampling-based) online tree-search algorithm, and we benchmark this approach against a model-free approach using Q-learning or Deep Q-learning. A model-based approach such as MCTS has the advantage that it is explainable - an important consideration for autonomous vehicles. However, a model-free approach such as Q-learning could be faster. Hence, we propose to combine the approaches, guiding the model-based MCTS tree search with a learned model-free heuristic.

Since the baseline and Oracle algorithms were covered in class, we do not describe these in detail, and we instead describe only the newer algorithms in the following sections.

4.1 Model

In this section, we describe our model for our baseline, Oracle, and MDP environments. A representation of the states, position and speed information, in absolute coordinates would be

$$S_t = \left\{ (x, y, v_x, v_y)_{\text{ego}}, (x, y, v_x, v_y)_{\text{obj}_{1..10}} \right\}$$

In our setting the ego car drives along the y-axis only, and we normalize the states as follows:

- **States:** $S \in \mathbb{R}^{42}$

$$S = \left\{ \left(\frac{y}{y^{max}}, \frac{v_y}{v_y^{max}} \right)_{\text{ego}}, \left(\frac{\Delta x}{\Delta x^{max}}, \frac{\Delta y}{\Delta y^{max}}, \frac{\Delta v_x}{\Delta v_x^{max}}, \frac{\Delta v_y}{\Delta v_y^{max}} \right)_{\text{obj}_{1..10}} \right\}$$

We use a discrete action space, of longitudinal accelerations along a predetermined path as follows:

- **Actions:**

$$A = \left[-2\ m/s^{-2}, -1\ m/s^{-2}, 0\ m/s^{-2}, 1\ m/s^{-2}, 2\ m/s^{-2}\right]$$

Our baseline and Oracle environments are deterministic. However, for our MDP model, transitions correspond to standard linear Gaussian dynamics and a constant velocity model, as follows:

- **Transitions:** $P\left(S_i^{t+1} \mid S_i^t, a_i^t\right) = \mathcal{N}\left(T_s S_i^t + T_a a_i^t, \Sigma\right)$

$$T_s = \begin{bmatrix} 1 & 0 & \mathrm{dt} & 0 \\ 0 & 1 & 0 & \mathrm{dt} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, S_i^{t+1} = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}, T_a = \begin{bmatrix} \frac{\mathrm{dt}^2}{2} & 0 \\ 0 & \frac{\mathrm{dt}^2}{2} \\ \mathrm{dt} & 0 \\ 0 & \mathrm{dt} \end{bmatrix}, a_i^t = \begin{bmatrix} a_x \\ a_y \end{bmatrix}.$$

The reward model accounts for efficiency (we penalize every step), safety (we heavily penalize collisions) and comfort (we penalize strong accelerations and decelerations:

- **Reward**:

$$R(s,a) = -1 - 1000 \times \mathbb{1}\left[\min_{\text{ego,obj} \in s} \|\text{ego-obj}\| \leq 10\right] - \mathbb{1}\left[\|a\| \geq 2\right]$$

4.2 Algo 1, MCTS tree search

Solving an MDP offline using Value Iteration is not an option as we are dealing with a huge state space. Instead, to solve the MDP online, we use an MCTS tree search. MCTS [8] is one of the most successful sampling-based online approaches used in recent years, and it is the core of AlphaGo Zero [12]. This algorithm involves running many simulations from the current state while updating an estimate of the state-action value function $Q(s,a)$ along its path of exploration.

MCTS balances exploration and exploitation via a method called the Upper Confidence Bound: during the search we execute the action that maximizes $Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}$, where $N(s), N(s,a)$ track the number of times a state and state-action pair are visited. Here c is a hyperparameter controlling the amount of exploration in the search: it encourages exploring less visited (s,a) pairs and relies on the learned policy via $Q(s,a)$ estimates for pairs that are well explored. Once we reach a state that is not part of the explored set, we iterate over all possible actions from that state and expand the tree.

After the expansion stage, a rollout evaluation phase is run, where many random simulations are performed to a fixed depth. The rollout policy is different from the exploration policy; and since it is typically stochastic, it does not have to be close to optimal. Simulations, from the root of the tree down to a leaf node expansion, are followed by a rollout evaluation phase. Together, these are run until a stopping criteria is met (either a time limit or a maximum number of simulations). We then execute the action that maximizes $Q(s,a)$ at the root of the tree. The pseudo code of the algorithm is provided in the Appendix.

One remaining problem with the above algorithm is the following: the same state may never be sampled more than once from the generative model, resulting in a shallow tree with just one layer. The Progressive Widening variant of MCTS [14, 3] solves this problem by controlling the sampling of new states and the sampling among already existing states to enable exploration in depth and not just in breadth.

4.3 Algo 2, Approximate Q-learning

Approximate Q-learning is Q-learning with a reduced set of features: $\phi(s,a) = \begin{bmatrix} s_{6\times1} & a_{1\times1} & s^2_{6\times1} & a^2_{1\times1} \end{bmatrix}^T$. Here, we take into account the state of the ego car (2 components), the state vector of the car with the minimum TTC (4 components), the acceleration command of the ego car, and quadratic of these components, since it is expected that the value of the (s,a) tuple will depend on distance computations. This reduced set of features should speed up training.

We then have the following set of equations:

$$\hat{Q}_{\text{opt}}(s,a;\mathbf{w}) = \mathbf{w} \cdot \phi(s,a)$$

$$\text{Objective} = \left(\hat{Q}(s,a;\mathbf{w})_{\text{pred}} - \left(r + \gamma \hat{V}_{\text{opt}}(s')\right)_{\text{targ}}\right)^2$$

$$\hat{V}_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s',a')$$

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \left[\hat{Q}_{\text{opt}}(s,a;\mathbf{w})_{\text{pred}} - \left(r + \gamma \hat{V}_{\text{opt}}(s')\right)_{\text{targ}}\right] \phi(s,a)$$

One of the problems we may encounter is that the data is highly correlated from one simulation step to the other, and the targets will vary a lot. This problem is typically handled by using an experience replay buffer, which is possible with an off-line policy algorithm, and by using a different fixed Q-network for target evaluation, which is updated less frequently than the Q-function used for predictions. This is described in the DeepMind papers [7, 9].

4.4 Algo 3, Deep Q-learning

Just as in the previous section, we use a DQN algorithm with a replay memory buffer and a separate target network. We use a Huber Loss which is more robust to outliers when the Q estimates are noisy: it acts as MSE when the error is small and as a mean absolute error when the error is large. Exploration is done with an ϵ-greedy policy. We use a batch size of 128, $\gamma = 0.999$ and Adam optimizer for the Batch Gradient Descent updates. Programming is

done with pytorch and we leverage on this code pytorch.org as a starting point for the DQN setup.

Our neural network has a 42-dimensional input, corresponding to the state dimensions as defined in Section 4.1 and a 5-dimensional output, corresponding to the 5 possible control commands. We use a CNN architecture similar to [5], leveraging CNN translational invariance properties, since the order of information about the different cars should not matter.

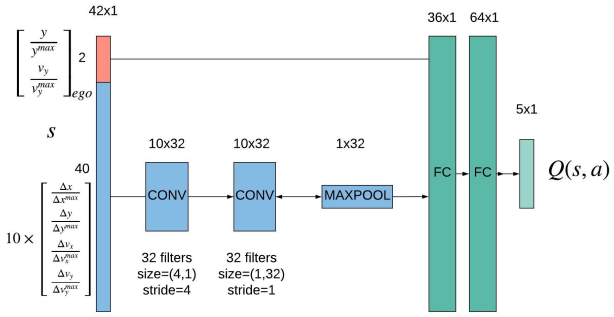

Figure 2: Q-function as a CNN network

4.5 Algo 4, MCTS tree search with a learned heuristic

We plan to use our learned Q-network $\hat{Q}(s, a; \mathbf{w})$ via approximate Q-learning or Deep Q-learning to guide the MCTS tree search, similar to [13, 2, 6, 12, 11]. In this way, we expand the tree in the most promising areas and hence can more quickly obtain a good solution. A solution is considered good as soon as it is estimated (in the presence of uncertainty) to be collision-free. We may run further MCTS tree searches up to some time limit, to find even better solutions, either in terms of speed or other metric, such as comfort.

5 Experiments

The source code is available here: CS221 Project. For our baseline (simple rule, reflex-based) algorithm, while the decision is fast and immediate, we find a collision-free velocity profile only in 35% of the cases.

For our Oracle algorithm, the search space is big. With a time step of 250 ms, we find a collision-free velocity profile in 100% of the cases over 100 meters. However, it takes 47.2 seconds with UCS and 190.7 seconds with DP running on an iCore9, where we explored the graph with a depth of 24. This makes sense, since UCS has a complexity of $\mathcal{O}(n \log n)$ with $n = |\text{states}|$ and explores fewer states than DP. Backtracking simply took too long. For the more advanced methods, some initial experimental results of a preliminary version of a Deep Q-learning algorithm are presented here: DQN notebook or alternatively here DQN pdf. Next, we will evolve the DQN network to a CNN architecture, tune hyper-parameters, train for much longer time and provide a custom Progressive Widening MCTS implementation enhanced with a learned DQN heuristic.

References

[1] Maxime Bouton et al. “Reinforcement Learning with Probabilistic Guarantees for Autonomous Driving”. In: *CoRR* abs/1904.07189 (2019). arXiv: 1904.07189. URL: http://arxiv.org/abs/1904.07189.

[2] Panpan Cai et al. “LeTS-Drive: Driving in a Crowd by Learning from Tree Search”. In: *CoRR* abs/1905.12197 (2019). arXiv: 1905.12197. URL: http://arxiv.org/abs/1905.12197.

[3] Adrien Couëtoux et al. “Continuous Upper Confidence Trees”. In: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*. LION’05. Rome, Italy: Springer-Verlag, 2011, pp. 433–445. ISBN: 978-3-642-25565-6. DOI: 10.1007/978-3-642-25566-3_32. URL: http://dx.doi.org/10.1007/978-3-642-25566-3_32.

[4] Andreas Folkers, Matthias Rick, and Christof BÃŒskens. “Controlling an Autonomous Vehicle with Deep Reinforcement Learning”. In: June 2019. DOI: 10.1109/IVS.2019.8814124.

[5] Carl-Johan Hoel, Krister Wolff, and Leo Laine. “Automated Speed and Lane Change Decision Making using Deep Reinforcement Learning”. In: *CoRR* abs/1803.10056 (2018). arXiv: 1803.10056. URL: http://arxiv.org/abs/1803.10056.

[6] Carl-Johan Hoel et al. “Combining Planning and Deep Reinforcement Learning in Tactical Decision Making for Autonomous Driving”. In: *CoRR* abs/1905.02680 (2019). arXiv: 1905.02680. URL: http://arxiv.org/abs/1905.02680.

[7] Thomas Howard et al. “State Space Sampling of Feasible Motions for High Performance Mobile Robot Navigation in Complex Environments”. In: *J. Field Robotics* 25 (June 2008), pp. 325–345. DOI: 10.1002/rob.20244.

[8] Mykel J. Kochenderfer. *Decision Making Under Uncertainty: Theory and Application*. MIT Press, 2015.

[9] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[10] B. Paden et al. "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles". In: *IEEE Transactions on Intelligent Vehicles* 1.1 (Mar. 2016), pp. 33–55. DOI: 10.1109/TIV.2016.2578706.

[11] F. Pusse and M. Klusch. "Hybrid Online POMDP Planning and Deep Reinforcement Learning for Safer Self-Driving Cars". In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. June 2019, pp. 1013–1020. DOI: 10.1109/IVS.2019.8814125.

[12] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[13] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–359.

[14] Zachary Sunberg and Mykel J. Kochenderfer. "POMCPOW: An online algorithm for POMDPs with continuous state, action, and observation spaces". In: *CoRR* abs/1709.06196 (2017). arXiv: 1709.06196. URL: http://arxiv.org/abs/1709.06196.

6 Appendix

The following pseudo-code describes the MCTS tree search:

function SELECTACTION(s, d)
  loop
    SIMULATE(s, d, π_0)
  end loop
  return arg max$_a$ $Q(s, a)$
end function

Dynamic Programming (DP) with Value or Policy Iteration can be used for problems with small state and action spaces and has the advantage of being guaranteed to converge to the optimal policy. Approximate Dynamic Programming (ADP) is often used for problems with large or continuous state and action spaces. In these problems, DP may be intractable so an approximation to the optimal policy is often good enough. Online methods are used for problems with very large or continuous state and action spaces where finding a good approximation to the optimal policy over the entire state space is intractable. In an online method, like MCTS, the optimal policy is approximated for only the current state. This approximation greatly reduces the computational complexity but also requires computation every time a new state is reached.

function SIMULATE(s, d, π_0)
  if $d = 0$ **then**
    return 0
  end if
  if $s \notin T$ **then**
    for $a \in A(s)$ **do**
      $(N(s, a), Q(s, a)) \leftarrow (N_0(s, a), Q_0(s, a))$
    end for
    $T = T \cup \{s\}$
    return ROLLOUT(s, d, π_0)
  end if
  $a \leftarrow$ arg max$_a$ $Q(s, a) + c\sqrt{\frac{logN(s)}{N(s,a)}}$
  $(s', r) \sim G(s, a)$
  $q \leftarrow r + \lambda$ SIMULATE($s, d - 1, \pi_0$)
  $N(s, a) \leftarrow N(s, a) + 1$
  $Q(s, a) \leftarrow Q(s, a) + \frac{q - Q(s,a)}{N(s,a)}$
  return q
end function

function ROLLOUT(s, d, π_0)
  if $d = 0$ **then**
    return 0
  end if
  $a \sim \pi_0(s)$
  $(s', r) \sim G(s, a)$
  return $r + \lambda$ ROLLOUT($s', d - 1, \pi_0$)
end function