Motion Planning for Autonomous Driving

Philippe Weingertner, Minnie Ho

November 11, 2019

1 Introduction

2 Related Work

There is a rich litterature related to Motion Planning and a very detailed survey of traditional methods is provided in [16]. Among the first 4 successful participants of DARPA Urban Challenge in 2007, the approaches vary. The winner, CMU Boss vehicle used variational techniques for local trajectory generation in a structured environment. This was done in a 2 steps path-velocity decomposition. A first step of path planning, using variational techniques, is performed and for every candidate path, a combination of different velocity profiles (constant, linear, linear ramp, trapezoidal) is applied : the combination of a path and velocity profile defines a trajectory. In unstructured environments (parking lots) or in error recovery situations a lattice graph in 4-dimensional configuration space (position, orientation and velocity) is searched with Anytime D* algorithm to find a collision-free path. More details are provided in [7, 1, 13]. The vehicle from Stanford used a search strategy coined Hybrid A* that constructs a tree of motion primitives by recursively applying a finite set of maneuvers. The search was guided by a carefully designed heuristic. The vehicle arriving 3rd, Victor Tango from Virginia Tech, constructs a graph discretization of possible maneuvers and searches the graph with the A* algorithm. The vehicle arriving 4th, developed by MIT used a variant of RRT algorithm with biased sampling. While all these techniques differ, they fundamentally rely on a graph search where nodes correspond to a configuration state and edges correspond to elementary motion primitives. Although they provide solutions, the runtime and state space can grow exponentially large. In this context, the use of heuristic to guide the search is important. (**NB: uncertainty is not properly considered here in these traditional MP methods. It is like reasoning with the mean state vector provided by sensor fusion output and ignoring the covariance matrix !**)

More recently, Reinforcement Learning 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 some research papers from Volvo [9] and BMW [8] , an RL agent is trained in simulation to take decision at a higher tactical level: the decisions relate to a maneuver selection, like lane change, rather than a low level acceleration command. DQN is used to train an agent. But the problem with Reinforcement Learning is that the utility is optimized in expectation. So even if the reward is designed to avoid collisions, this will be optimized in expectation: ultimately it is as if safety would be enforced with soft constraints rather than hard constraints. Which is of course not acceptable for a real vehicle. To solve this problem in [8] an additional safety check layer is added after the DQN agent to eventually override the DQN agent decision if it is considered unsafe. Checking a decision wrt to a specific criteria is simpler than designing a decision making system that jointly optimizes efficiency, comfort and safety objectives. With RL applied to AD we have to account for additional safety checks. In [2] Deep RL is applied at the local planner level: the action space is a set of longitudinal accelerations $\left\{-4m/s^2, -2m/s^2, 0m/s^2, 2m/s^2\right\}$ applied along a given path at a T-intersection. Safety is handled in a different way here compared to previous BMW approach: the agent is constrained to choose among a restricted set of safe actions per state. So the safety is enforce before Deep RL. Ultimately car manufacters may want to combine both types of safety checks: constraining the action set per state before enabling an RL agent to make its own decision, and checking again the final sequence of decisions proposed by the RL agent.

Now the interesting topic is how to best combine traditional Motion Planning with RL. What are the limitations of these techniques in isolation and how to use the strengths of both approaches and circumvent their weaknesses. Traditional motion planning relies heavily on tree search and to enable real time solutions good heuristics are required. Designing a good heuristic is hard. What if we could learn it ? By training an agent with model free RL we can potentially end up with an agent that performs pretty well most of the time and from time to times fails miserably in a way that is hard to explain. The main problems with model free RL are sample efficiency (we need a lot of data), enforcing hard constraints and explainability (how can we explain the decision taken by a RL agent which may become a problem for a car manufacturer). While a model based planning method has the advantages of explainability, do not rely on data and can deal in a more systematic way with hard constraints. As demonstrated in [3] in simple situations RL methods have no benefit over rule based methods, pure RL does not enable the agent to act in safer way. But when the situation becomes much more complex with an increasing number of cars and pedestrians, the benefits of Deep RL methods become clear.

In the gaming domain, chess and go, performances superior to human performances have been achieved with AlphaGo Zero [20]: by combining planning with MCTS tree search and learning with RL. A neural network biases the sampling towards the most relevant parts of the search tree: a learnt policy-value function is used as a heuristic during inference. While during training, MCTS is used to improve the sample efficiency of RL training. Now there are a few major differences between a game like chess or go and our initial Motion Planning problem. In chess or go the state space is discrete and fully observable while in AD the state space is continuous and partially observable. In terms of action sets in both cases, we can deal with discrete action sets. But another challenge is that self-play can not be used in the context of Motion Planning. These challenges have been recently tackled in different publications. The applicability of AlphaGo Zero to Autonomous Driving has been studied in [10, 4, 17].

In [17] the Motion Planning problem is adressed in a 2 steps path-velocity decomposition. The path planner employs hybrid A* to propose paths that are driveable and collision free wrt static obstacles. In a second step a velocity profile is generated by issuing acceleration commands. The problem is formulated as a POMDP model and solved with an online DESPOT solver. DESPOT is a sampling based tree search algorithm like MCTS which uses additional lower bounds and upper bounds values. To guide the tree search of DESPOT, a NavA3C neural network is used. The NavA3C network is trained in simulation and is expected to provide tighter bounds than the heuristic commonly used for lower and upper bounds estimation.

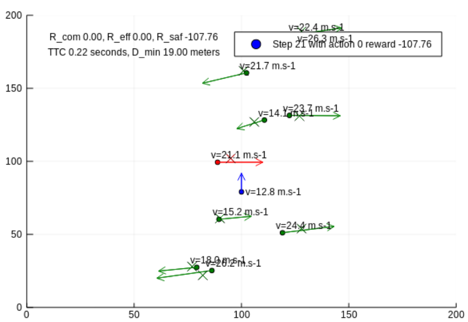
In [4] the problem of pedestrians collision avoidance in dense urban traffic areas is considered. Again the problem is formulated as a POMDP and DESPOT is used as an online solver. The 2 steps path-velocity decomposition is used: Hybrid A* plans a path and POMDP is restricted to control only the accelarations along the path. Two stages are considered: an imitation stage where a Neural Network is trained in simulation to solve the POMDP problem. In the improvement stage, LeTS-Drive uses Hyp-DESPOT, a massively parallel belief tree search algorithm, to plan vehicle motions. LeTS-Drive incorporates the prior knowledge learned in the policy and value networks into the heuristics of HyP-DESPOT in order to search efficiently within the limited planning time.

In [10] the problem is considered at a higher level: at the behavioral planning level. It is mainly a set of lane changes decisions that are taken to navigate a highway or to reach a highway exit. The problem is formulated as a MDP problem. MCTS tree search is used as an online MDP solver and a learned policy-value network is used to efficiently guide the search.

In our case we consider the problem of the local planner and velocity profile generation, similar to the 2 first papers, but with an approach mainly aligned with the later one.

3 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 $\{V\}$.



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 via dots while the position reported by the sensors is represented by crosses. A Time To Collision based on ground truth information is displayed and if there exist an intersecting car with a predicted TTC below 10 seconds it is displayed in red. This test framework is a custom one we have developped. We have a version of this test framework that is compatible with open ai gym interfaces: so that any standard Deep RL setup, can be directly used with this environment. Typically we intend to use a DQN setup from pytorch.org initially tested on cartpole-v1 with our own Act-v1 environment. Our simulation and test environment can be downloaded and installed from gym-act.

4 Approach

4.1 MDP model

A representation of the states in absolute coordinates would be $S_t = \left\{ (x, y, v_x, v_y)_{\text{ego}}, (x, y, v_x, v_y)_{\text{obj}_{1..10}} \right\}$. But we use a relative and normalized $\in [-1, 1]$ representation of the state (for easier generalization and learning) and account for the fact that the ego car drives along the y-axis only.

- **States:** $S = \left\{ \left(\frac{y}{y^{max}}, \frac{v_y}{v_y^{max}} \right)_{\text{ego}}, \left(\frac{\Delta x}{\Delta x^{max}}, \frac{\Delta y}{\Delta y^{max}}, \frac{\Delta v_x}{\Delta v_x^{max}}, \frac{\Delta v_y}{\Delta v_y^{max}} \right)_{\text{obj}_{1..10}} \right\}$ ie a vector $\in \mathbb{R}^{42}$

While the state space is continuous we use a discrete action space.

- **Actions:** $A = \left[-2\ ms^{-2}, -1\ ms^{-2}, 0\ ms^{-2}, 1\ ms^{-2}, 2\ ms^{-2} \right]$ corresponding to the longitudinal acceleration.

The Transition model corresponds to standard linear Gaussian dynamics with:

- **Transitions:** $T(s' \mid s, a) = P\left(S_i^{t+1} \mid S_i^t, a\right) = \mathcal{N}\left(T_s S_i^t + T_a \begin{bmatrix} a_x \\ a_y \end{bmatrix} \right)$

Using a Constant Velocity Model with $T_s = \begin{bmatrix} 1 & 0 & \text{dt} & 0 \\ 0 & 1 & 0 & \text{dt} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, $T_a = \begin{bmatrix} \frac{\text{dt}^2}{2} & 0 \\ 0 & \frac{\text{dt}^2}{2} \\ \text{dt} & 0 \\ 0 & \text{dt} \end{bmatrix}$ which corresponds to $\begin{bmatrix} x^{t+1} \\ y^{t+1} \\ v_x^{t+1} \\ v_y^{t+1} \end{bmatrix} = \begin{bmatrix} x^t + v_x^t \text{dt} \\ y^t + v_y^t \text{dt} \\ v_x^t \\ v_y^t \end{bmatrix} + \begin{bmatrix} a_x \frac{\text{dt}^2}{2} \\ a_y \frac{\text{dt}^2}{2} \\ a_x \text{dt} \\ a_y \text{dt} \end{bmatrix} + \begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_{v_x} \\ \sigma_{v_y} \end{bmatrix}$.

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

- **Reward**: $R(s, a) = -1 - 1000 \times 1\left[\text{d(ego,obj)}_s \leq 10\right] - 1\left[|a| = 2\right]$

4.2 Algo 1, MCTS tree search

The MDP is solved online with MCTS tree search. Solving it offline with Value Iteration is not an option as we are dealing with a huge state space. MCTS is one of the most successfull sampling-based online approaches used in recent years. It is the core part of AlphaGo Zero [19]. A description of the algorithm is provided in [11]. 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. Online algorithms enable to reduce the search space to the portion of the state space that is reachable from a current state. MCTS has the possibility to balance exploration and exploitation typically via a method called 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 . c is a parameter that controls the amount of exploration in the search: it will encourage exploring less visited (s,a) pairs and rely on the learned policy via $Q(s,a)$ estimates for pairs that are well expolored, to choose an action from. 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 is performed: the rollout consists in running many random simulations till we reach some depth. It is a Monte Carlo estimate of a state so the rollout policy are typically stochastic and do not have to be close to optimal. The rollout policy is different than the policy used for exploitation/exploration presented above. Simulations, running from the root of the tree down to a leaf node expansion, followed by a rollout evaluation phase, are run until some stopping criterion is met: a time limit or a maximum number of iterations. We then execute the action that maximizes $Q(s,a)$ at the root of the tree. The pseudo code of the algorithm is provided below:

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

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
  $a \leftarrow \arg\max_a Q(s,a) + c\sqrt{\frac{log N(s)}{N(s,a)}}$
  $(s', r) \sim G(s,a)$
  $q \leftarrow r + \lambda$ 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

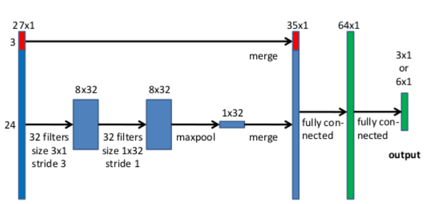
One remaining problem is that in chess or go, state space is discrete and the above algorithm does not cope with continuous state space: the same state may never be sampled more than once from the generative model which will result in a shallow tree with just one layer. The Progressive Widening variant of MCTS [21, 5] 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

While ultimately and similarly to the papers referenced in the literature review section [20, 4, 10, 19, 17] we intend to use Deep Learning to learn an evaluation function that will be used later on as a heuristic to guide the MCTS tree search, we will first try Approximate Q-learning to check how it performs with a simple set of handcrafted features. We first derive a simple features extractor. $\phi(s,a) = \begin{bmatrix} s_{6\times1} \\ a_{1\times1} \\ s^2_{6\times1} \\ a^2_{1\times1} \end{bmatrix}$ where we take into account the state vector of the ego car (2 components) and the state vector of the car with the smallest TTC (4 components). We also take into account the acceleration command of the ego car. We use quadratic components as well: as it is expected that the value of a (s, a) tuple will depend on distances computations. Typically when computing Time To Collision, quadratic terms appear so we want to provide these relevant features and figure out by learning the weights associated to these features. We try to focus on a reduced set of relevant features to speed up training. The Q-function is parametrized by a vector w with $\hat{Q}_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$. We have $\hat{V}_{\text{opt}}(s') = \max\limits_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a')$ and use the objective $\left(\hat{Q}(s, a; \mathbf{w})_{\text{pred}} - \left(r + \gamma \hat{V}_{\text{opt}}(s')\right)_{\text{targ}}\right)^2$ which leads to the following update rule while performing Sochastic Gradient Descent: $\mathbf{w} \leftarrow \mathbf{w} - \eta \left[\hat{Q}_{\text{opt}}(s, a; \mathbf{w})_{\text{pred}} - \left(r + \gamma \hat{V}_{\text{opt}}(s')\right)_{\text{targ}}\right] \phi(s, a)$. One of the problem we may encounter is that the data in simulation is not iid, but 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 policy algorithm) and using a different fixed Q-network for targets evaluation: which is updated less frequently than the Q-function used for predictions: as described in DeepMind papers [1, 14].

4.4 Algo 3, Deep Q-learning

Model Free RL method. Our Neural Network has 42 neurons as input, corresponding to the state vector, and 5 neurons as output. We will use a Neural Network architecture slightly adapted from [9]. It is based on a CNN network as we want to have translational invariance of the input. It should not matter to provide information about different cars in one order or the other.



4.5 Algo 4, MCTS tree search with a learned heuristic

Combining Planning and Learning, Model Based and Model Free RL. TODO post progress report.

We want to use our learned Q-network $\hat{Q}(s, a; \mathbf{w})$ via approximate Q-learning or Deep Q-learning as a heuristic for MCTS tree search: to expand the tree in the most promising areas and hence come up faster with a good solution. A solution is considered good as soon as it is estimated collision free; we may run further MCTS tree searches up to some time limit, to find even better solutions: faster or more comfortable. The reward takes into account safety, comfort and efficiency. While there is a big penalty for collisions of -1000, at every time step we penalize the reward by -1 to enforce efficiency and penalize every strong acceleration or deceleration, when $|a| = 2 \quad ms^{-2}$, by -2 to encourage more comfortable trajectories.

5 Experimental Setup and Status

The source code is available here: CS221 Project

- Baseline: simple rule - reflex based. DONE
- Oracle: assumes no uncertainty, UCS/A* tree search. DONE
- Sequential Decision Making with Uncertainty => solve a MDP
  - Planning with MCTS tree search. ON-GOING: MDP model implementation + our own MCTS algo implementation
  - Approximate Q-learning: leverage on CS221 hw4 blackjack setup + custom Features Extractor + interface with our MDP model
  - Deep Q-learning: basically the pytorch DQN setup from pytorch.org customized with our proposed CNN network and interfaced with our own gymai environment (gymai-act for Anti Collision Tests)

– Combining Planning and Learning: we should have MCTS and Q-learning working independently first ..

References

[1] David Bissell et al. "Autonomous automobilities: The social impacts of driverless vehicles". In: *Current Sociology* (Dec. 2018), p. 001139211881674. DOI: 10.1177/0011392118816743.

[2] 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.

[3] Maxime Bouton et al. "Safe Reinforcement Learning with Scene Decomposition for Navigating Complex Urban Environments". In: *CoRR* abs/1904.11483 (2019). arXiv: 1904.11483. URL: http://arxiv.org/abs/1904.11483.

[4] 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.

[5] 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.

[6] Michael Everett, Yu Fan Chen, and Jonathan P. How. "Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning". In: *CoRR* abs/1805.01956 (2018). arXiv: 1805.01956. URL: http://arxiv.org/abs/1805.01956.

[7] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. "Motion Planning in Urban Environments". In: *The DARPA Urban Challenge*. 2009.

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

[9] 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.

[10] 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.

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

[12] Edouard Leurent. "A Survey of State-Action Representations for Autonomous Driving". working paper or preprint. Oct. 2018. URL: https://hal.archives-ouvertes.fr/hal-01908175.

[13] M. McNaughton et al. "Motion planning for autonomous driving with a conformal spatiotemporal lattice". In: *2011 IEEE International Conference on Robotics and Automation*. May 2011, pp. 4889–4895. DOI: 10.1109/ICRA.2011.5980223.

[14] 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.

[15] Subramanya Nageshrao, H. Eric Tseng, and Dimitar P. Filev. "Autonomous Highway Driving using Deep Reinforcement Learning". In: *CoRR* abs/1904.00035 (2019). arXiv: 1904.00035. URL: http://arxiv.org/abs/1904.00035.

[16] 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.

[17] 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.

[18] Markus Schratter et al. "Pedestrian Collision Avoidance System for Scenarios with Occlusions". In: *CoRR* abs/1904.11566 (2019). arXiv: 1904.11566. URL: http://arxiv.org/abs/1904.11566.

[19] 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.

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

[21] 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.