Monte Carlo Tree Search With Reinforcement Learning for Motion Planning

Philippe Weingertner[1,2], Minnie Ho[2,3], Andrey Timofeev[4] and Sébastien Aubert[1]

***Abstract*— Motion planning for an autonomous vehicle is most challenging for scenarios such as large, multi-lane, and unsignalized intersections in the presence of dense traffic. In such situations, the motion planner has to deal with multiple crossing-points to reach an objective in a safe, comfortable, and efficient way. In addition, motion planning challenges include real-time computation and scalability to complex scenes with many objects and different road geometries. In this work, we propose a motion planning system addressing these challenges. We enable real-time applicability of a Monte Carlo Tree Search algorithm with a deep-learning heuristic. We learn a fast evaluation function from accurate, but non real-time models. While using Deep Reinforcement Learning techniques we maintain a clear separation between making predictions and making decisions. We reduce the complexity of the search model and benchmark the proposed agent against multiple methods: rules-based, MCTS, A* search, deep learning, and Model Predictive Control. We show that our agent outperforms these other agents in a variety of challenging scenarios, where we benchmark safety, comfort and efficiency metrics.**

I. INTRODUCTION

We consider the problem of Motion Planning for Autonomous Driving (AD), where a behavioral planner defines a short-term objective corresponding to a desired maneuver. The behavioral planner formulates the problem for a local planner, which then has to find a feasible trajectory achieving the desired objective under a set of constraints. To reduce complexity, trajectory planning is typically done in a two-stage (path-velocity) process. First, the local planner computes a set of paths that are kinematically feasible and collision-free with respect to static obstacles. Secondly, for every candidate path, a velocity profile is generated to avoid dynamic obstacles. We focus on the Motion Planning problem, where we must determine a set of acceleration commands along a candidate path, or a set of candidate paths, to avoid dynamic obstacles, ensure comfort, and reach a target goal at a desired speed.

We tackle a set of four core challenges in Motion Planning for AD. First, the solution must be both accurate and real-time. In particular, a motion plan should define control commands every 250 ms, but the motion planner may be triggered every 40 ms. Most papers focus on accuracy and do not provide details on real-time applicability of the proposed methods. However, this is the main challenge and a core issue for a car manufacturer.

[1]Philippe Weingertner and Sébastien Aubert are with Groupe Renault, Renault Software Labs, France. Correspondences: philippe.weingertner@renault.com

[2]Stanford University {pweinger,minnieho}@stanford.edu

[3]Intel Corporation minnie.ho@intel.com

[4]Experis Switzerland andr.timofeev@gmail.com

The second core challenge is to derive a generic solution that is usable for all road geometries and scalable with the number of objects in the scene. Third, in order to drive in dense, large, multi-lane, and unsignalized intersections, we must account for multiple crossing-points along the path. As described in [1], a simple controller is not sufficient. Fourth, we want to clearly separate predictions from decisions. In some proposals [2]–[5], predictions and decisions are made at the same time. This makes it challenging for a car manufacturer to root-cause the problem when the system does not behave as expected, especially given uncertain driving models and sensor measurements.

II. RELATED WORK

There is rich literature related to Motion Planning, and a very detailed survey is provided in [6]. 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 runtime and state space can grow exponentially large, and 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 AD for decision making either at the Behavioural Planning or Motion Planning level. In papers from Volvo [3] and BMW [7], 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 [7], a safety check layer is added after the DQN agent in case an override decision is needed. Alternatively, safety is enforced before Deep RL. In [8], 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 [9] 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. These techniques cannot be directly applied to Motion Planning, as the state space is continuous and only partially observable. Also self-play can not be used. These challenges have been recently addressed in several publications, including [10]–[13].

In [12], the Motion Planning problem is addressed in a two-stage (path-velocity) process. 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 solver called DESPOT. DESPOT is a sampling-based tree search algorithm like MCTS, with the addition of a neural network, called NavA3C [13], trained to help guide the construction of a belief tree used in the DESPOT tree search.

In [10] 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 [12], but with an algorithm approach similar to [10].

This paper is structured as follows. Section III introduces the problem formulation. Section IV describes the agents. In section V, experiments and results are presented and discussed. Finally, we draw conclusions in section VI.

III. Problem Formulation

To have a generic problem formulation and to reduce complexity, we project the trajectories of the surrounding objects onto the ego path and look for a one dimensional motion plan along this path. In other words, we define a set of spatio-temporal points $\{(s_i, t_i)\}_{i=1..n}$ we want to avoid, where s is the distance along the path from the ego position, t the time, n the number of crossing-points, and $\dot{s}_{des}$, the desired speed.

To handle sharp turns at intersections, we define a set of maximum values $(\dot{s}_{max}(\kappa), \ddot{s}_{max}(\kappa))$ for ego-acceleration and ego-speed depending on the curvature κ, with $(s_f, \dot{s}_f)$ the target for the ego vehicle. In order to ensure comfort, we want to avoid strong accelerations and decelerations.

A. Search model

We define a MDP model with deterministic transitions and reward models based on a set $\mathcal{C}$ of crossing-points requirements (pre-defined by a prediction module) as follows:

- **States:** $\mathcal{S} = \left\{ \frac{s}{s_{max}}, \frac{\dot{s}}{\dot{s}_{max}}, t \right\}$
- **Desired Terminal State:** $\left(\frac{s_f}{s_{max}}, \frac{\dot{s_f}}{\dot{s}_{max}} \right)$
- **Undesired Terminal States:**
$$\mathcal{S} \cap \mathcal{C} = \left\{ \left(\frac{s_i}{s_{max}}, t_i \right) \right\}_{i=1..n}$$
- **Actions:** $\mathcal{A} = [-4, -2, -1, 0, 1, 2]$ ms^{-2}
- **Transitions:**
$$\begin{bmatrix} s \\ \dot{s} \end{bmatrix}_{t+1} = A_d \begin{bmatrix} s \\ \dot{s} \end{bmatrix}_t + B_d \left[\ddot{s}\right]_t \text{ with } A_d = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, B_d = \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix}$$
- **Reward or cost model:**
$$R(s,a) = -0.001 - 1 \times \mathbb{1}\left[\text{distance(ego, obj)} \leq d_{\text{col}}\right] - 0.002 \times \mathbb{1}\left[a \leq -4\right]$$

Surrounding dynamic objects are captured in our model by defining a list of spatio-temporal points to avoid. Any intersection results in an undesired terminal state, while the desired terminal state we want to reach is typically defined by a Behavioral Planner. Cost and reward can be used interchangeably, with the difference only in the sign. This model accounts for efficiency (penalizing every step), safety (penalizing collisions) and comfort (penalizing hard braking at $-4\ ms^{-2}$).

B. Complexity reduction

Our model differs from a typical state-space representation in that we do not model the state vector of all objects in the scene (which would lead to a huge state space). By defining the intersection points to avoid as a projected space onto the ego path, we have a generic problem formulation for the Motion Planner that is independent of road geometries and applicable to many objects in the scene. Furthermore, we have only a single kinematic model (the ego vehicle) embedded in a two-dimensional spatio-temporal representation. Tree search and model-based algorithms will run much faster than when dealing iteratively with the (potentially uncertain) motion models of the surrounding objects of a complex POMDP model. With this deterministic and generic search model from which sources of uncertainties have been pre-analyzed and projected onto the ego path, we combine the search with the set of requirements to which we must comply.

IV. Agents

Our implementation is available at MCTS-NNET.

A. Baseline and Oracle agents

With the first baseline (Baseline v1), when a Time To Collision (TTC) is below 10 seconds, the agent decelerates at -2 ms^{-2} to avoid a collision; otherwise the agent accelerates at 1 ms^{-2}, up to a speed limit, to reach the target as fast as possible. The second Baseline (Baseline v2), applies a similar rule but with a very strong deceleration, -4 ms^{-2}, to further limit the collision risk with the object with the lowest TTC. These are simple rules which are very intuitive for human drivers. The Oracle relies on an exhaustive search optimized via Uniform Cost Search (UCS or A*) or Dynamic Programming (DP). It finds the optimal solution, but it is computationally expensive.

B. MCTS agent

MCTS [14] is one of the most successful sampling-based online approaches used in recent years, and it is the core of AlphaGo Zero [15]. 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 hyper-parameter 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 phase is run, where many random simulations are performed to a fixed depth. The rollout policy is different from the exploration policy; 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.

For AD we are more constrained in terms of hardware resources and timing requirements than for gaming [9], [15], so MCTS can not rely on as many rollouts and iterations. We refer to Table I for the parameters we used for the MCTS agent.

TABLE I

MCTS PARAMETERS

	MCTS Parameters
Iterations	100
Depth	12
Exploration Method	UCB-1
Exploration Constant	1.0
$Q(s, a)$ evaluation	by rollouts
Restrict Actions	ON

Normally, the full set of actions is considered in every state. We implement an option to restrict actions, where we remove those actions which decrease the smallest TTC, and hence are considered unsafe in a particular state. If all actions decrease the smallest TTC, we restrict the action set to the action with the minimum decrease.

C. MPC agent

We use an (s, t) longitudinal formulation and a constrained linear-quadratic Model Predictive Control (MPC) model, solving the following finite-horizon optimal control problem at each time step for $t \in [0, T]$:

$$\min_{u_0,\ldots,u_{T-1}} \left(x_T\text{-}x_r\right)^\intercal Q_T \left(x_T - x_r\right) + \sum_{t=0}^{T-1} \left(x_t - x_r\right)^\intercal Q \left(x_t - x_t\right) + u_t^\intercal \; R \; u_k$$

$$\text{such that} \begin{cases} x_{t,min} \le x_t \le x_{t,max} \\ u_{t,min} \le u_t \le u_{t,max} \\ x_{t+1} = A_d x_t + B_d u_t \\ x_0 = x_{init} \\ \forall \left(t_{\text{crossd}}, s_{\text{cross}}\right) \quad x_{t_{\text{crossd}}}[0] < s_{\text{cross}} - \Delta_{\text{margin}} \\ \text{with } t_{\text{crossd}} = \left\lfloor \frac{s_{\text{cross}}}{\Delta t} \right\rfloor \end{cases}$$

Here, the state vectors x_t,x_T,$x_r \in \mathbb{R}^2$ are of the form $\begin{bmatrix} s & \dot{s} \end{bmatrix}^T$ and the control command $u_t \in \mathbb{R}$ is of the form $\begin{bmatrix} \ddot{s} \end{bmatrix}$. Q_N and Q are two 2×2 symmetric positive definite matrices, R is in general a symmetric positive definite matrix (a simple scalar here), and A_d and B_d define a Constant Acceleration Model in between two time steps, with A_d and B_d as defined in Section III-A.

We use the following numerical values: a time step $\Delta t =$ 250 ms, a time horizon $T = 20$ steps (5 seconds), $Q_N = Q = \operatorname{diag}(1, 50)$, $R = 0.001$, and a reference target vector $x_r = \begin{bmatrix} 200 & 20 \end{bmatrix}^\intercal$. An optimized implementation is done in Julia with JuMP [16] and Ipopt [17] solver.

The difficulty of the problem relates to the multiple spatio-temporal points to avoid. An optimal sequence of decisions may require alternatively proceeding and yielding to the crossing vehicles. We would like to define a constraint, such as $s_{ego} < s_{cross} - \Delta_{margin}$ or $s_{ego} > s_{cross} + \Delta_{margin}$, but this does not correspond to a convex set.

In practice we look for a solution where we first yield to the three closest crossing vehicles (in time). If no such solution exists, we look for a solution where we first proceed with respect to these three closest vehicles. We are constrained by the convex formulation of the problem which restricts the set of candidate solutions, so if the solver does not find a solution, we return to our baseline decision rule. An important difference with the other agents is that the action command may be any value in the range $[-4, 2]$ ms^{-2}.

D. DDQN agent

We implement a Double DQN (DDQN) agent [18]. The objective here is not to learn a model from collected data or from an unknown model, but to best fit a faster parametric model (a Neural Network in this case) to a slow but very accurate agent. Indeed, with UCS and DP search, we have a motion planner that can fulfill our accuracy requirements, but this motion planner is very slow.

In other words, given a model, with known ego-vehicle dynamics and known terminal states, a set of spatio-temporal points we should avoid, and a terminal state we want to reach, we perform DDQN training to learn a fast parameterized Q-value function, which evaluates a specific action in a specific state. This function is then used to define a policy when using DDQN in standalone mode or used as a heuristic to efficiently guide MCTS tree search, by replacing rollouts and initializing $Q(s, a)$ values. Here s refers to the state vector to keep the notation consistent with RL literature.

The Q-function is approximated by a neural network $\hat{Q}(s, a; w)$ with w as the trainable parameters. The loss is defined as

$$\text{Loss}_w = \hat{Q}(s, a; w)_{\text{pred}} - \Big(r + \gamma \; \max_{a'} \hat{Q}(s', a'; w-)\Big)_{\text{target}}$$

with a fixed Q-target network $\hat{Q}(s', a'; w-)$ used to stabilize training. Every 1000 iterations, the weights of the target Q-target network are updated. An experience replay buffer is used to avoid training with correlated samples. The parameter update is computed with a batch size of 32 as follows:

$$w \leftarrow w - \eta \nabla_w \text{Loss}_w(s, a)$$

The parameters used for training are summarized in Table II below.

TABLE II
TRAINING PARAMETERS

	DDQN Training Parameters
Neural Net	8 inputs, 2 FC_{200} with ReLu + 1 FC_{200}, 6 outputs
Inputs (Normalized)	state=$s_{ego}, \dot{s}_{ego}, \left\{(s_i, \text{ttc}_i)_{1..3}\right\}$
Outputs	$Q\left(s,a\right)$ values for $a \in [-4,-2,-1,0,1,2] \quad ms^{-2}$
DQN	local and target network with double DQN implementation
Target Network Updates	Every 10K samples
Training Set, Dev Set	50K episodes (test scenes), 100 episodes
Discount Factor	1
$\epsilon-$greedy exploration	$\epsilon_{\text{start}} = 1, \epsilon_{\text{end}} = 0.01, \epsilon_{\text{decay}} = 0.995$ every episode
Replay Buffer	10K samples
Batch Size	32
Optimizer	Adam(learning rate=2.5e-4)
Clip Gradient Norm	10

In practice, our main issue was to derive a state-space representation enabling convergence to a useful function. Providing raw information about all constraints with the full set of $\left\{(s_i, t_i)_{i=1..n}\right\}$ spatio-temporal points to avoid was unsuccessful, since the network was not able to converge. We had to use a smaller and more importantly - a sorted list - to enable training. When the state-space representation is restricted to a list of the three most problematic crossing-points, sorted by criticality in terms of smallest TTC, the network quickly learns a useful Q-function. But then by design, it is clear such a DDQN network will not be able to handle the most difficult cases, where reasoning about more than three crossing-points is required.

E. MCTS-NNET agent

The pseudo code of the MCTS-NNET agent is provided below. The learned Q-network $\hat{Q}(s, a; \mathbf{w})$ is used to guide the MCTS tree search, similar to [9]–[12], [15]. In this way, we expand the tree in the most promising areas. The main problem of MCTS in our context is the dependency on many online statistics and iterations to come up with a good solution.

```
function SELECTACTION($s, d$)
    loop
        SIMULATE($s, d, \pi_0$)
    end loop
    return arg max$_a$ $Q(s, a)$
end function
```

Even if the parametric Q-function is not perfect, as it is using a state space representation limited to the three most dangerous objects in the scene (for practical reasons explained before), it is expected to provide a powerful heuristic. The DDQN network, with a representation limited to these three most dangerous objects used as heuristic, and the MCTS tree search dealing with the full set of requirements (not limited to the three most dangerous objects) are clearly complementary. As an additional refinement, we estimate how discriminating the Q-network is about a $Q\left(s,a\right)$ evaluation, by checking the difference $\left|\max\limits_a Q\left(s,a\right) - \min\limits_a Q\left(s,a\right)\right|$.

```
function SIMULATE($s, d, \pi_0$)
    if $d = 0$ then
        return 0
    end if
    if $s \notin T$ then
        for $a \in A(s)$ do
            if mcts-nnet then
                $Q(s, a), N(s, a) \leftarrow$nnet.getQ(s,a), $1$
            else
                $Q(s, a), N(s, a) \leftarrow 0, 0$
            end if
        end for
        $T = T \cup \{s\}$
        return ROLLOUT($s, d, \pi_0$)
    end if
    if mcts-nnet and $\Delta_a Q(s, a) > 0.1$ then
        $c \leftarrow 0$
    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($s, d, \pi_0$)
    if $d = 0$ then
        return 0
    end if
    if mcts-nnet then
        return nnet.getV(s)
    else
        $a \sim \pi_0(s)$
        $(s', r) \sim G(s, a)$
        return $r + \lambda$ ROLLOUT($s', d - 1, \pi_0$)
    end if
end function
```

When this difference is above some threshold, the level of exploration is increased. To the best of our knowledge, this is a novel feature. It is implemented in MCTS-NNET v2.

When using MCTS in standalone mode, we had to rely on the option to restrict actions to significantly improve the results. This function provides a restricted but safer set of actions per state. It appeared that while using MCTS in combination with NNET this option had a negative impact: it was too restrictive and the results of MCTS-NNET were exactly the same as MCTS in standalone. This is most probably because the restrict action function is considering only the most dangerous object in the scene while the DDQN is considering the three most dangerous objects in the scene.

V. EXPERIMENTS

We use five metrics to evaluate the performance of our agents. The main success metric is the percentage of cases where we reach a target state without collision. The second metric is the agent runtime. The absolute value of this metric is only indicative as the experimental software is not optimized and the hardware is a powerful laptop, but it nevertheless provides information in terms of real-time applicability and how the different algorithms compare to each other. Anything above 40 ms is problematic, and ideally we would like to be below 1 ms. The third metric is a comfort metric: the number of hard braking decisions. The fourth metric relates to efficiency: how fast we reach a target while complying to some speed limitation. The last metric is a safety metric: for eleven of our randomly generated test cases, a collision was unavoidable. In these cases, we aim for a lower speed at collision.

The agents described in the previous section are first tested in scenarios with multiple crossing-points. We then proceed with Intersection and Automatic Cruise Control (ACC) tests. We finally check the results on single crossing-point tests. All simulations are run on a laptop with an Intel® Core-i9 processor for runtime consistency.

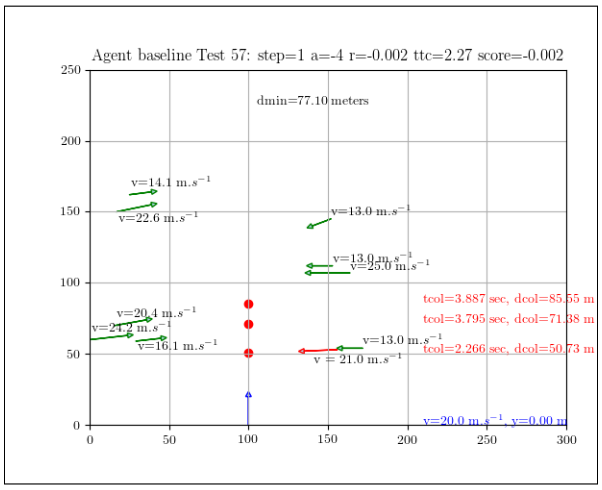


Fig. 1. Example of a Multiple Crossing-Points Test

A. Scenarios and Results

1) Multiple crossing-points tests: Scenarios with multiple crossing-points are challenging, since the ego vehicle may have to alternatively proceed and yield to avoid collisions with multiple vehicles in order to efficiently reach a goal. We randomly generate these scenes with five vehicles on the right and five vehicles on the left moving at different speeds according to different paths, as shown in Figure 1. We generate 100 test cases for this scenario, where each scenario is initialized in such a way that random actions or no actions will lead to a collision.

In Table III, the success statistics have been normalized with respect to the Oracle to exclude unavoidable collisions. We observe that the smooth Baseline v1, which excludes hard-braking, succeeds in 43% of the cases. The emergency Baseline v2, which includes hard-braking when the TTC is below 10 seconds, succeeds in 72% of the cases. When a collision cannot be avoided, we limit collision speed.

TABLE III
RESULTS WITH MULTIPLE CROSSING-POINTS TESTS

Mean over 89 Tests	Success	Runtime	Hardbrakes	Steps	Collision Speed
Baseline v1	43%	40 μs	0	43	13.04 m.s^{-1}
Baseline v2	72%	40 μs	10.57	53	7.38 m.s^{-1}
MCTS	85%	3 ms	4.43	42	14.77 m.s^{-1}
DDQN	84%	300 μs	5.15	62	17.13 m.s^{-1}
MPC	82%	18 ms	6.22	47	8.72 m.s^{-1}
MCTS-NNET v1	98%	4 ms	4.79	62	19.04 m.s^{-1}
MCTS-NNET v2	96%	4 ms	4.89	48	18.40 m.s^{-1}
Oracle (A* search)	100%	6 sec	1.52	40	-

Unfortunately, by focusing only on the most dangerous car with the smallest TTC, the baseline strategies cannot avoid collisions in those test cases where we have to reason about multiple colliding vehicles.

The Oracle performs an exhaustive search via UCS or DP. While it is always able to fulfill our requirements (when possible), it takes a few minutes, which is clearly not suitable for a real-time application. With the MPC agent, as we are only looking for solutions over convex sets, more complex and clever strategies are excluded. However, the MPC solution is guaranteed to be a global optimal, so when it does find a solution without collision, it is the closest to the optimal Oracle solution in almost all cases.

With MCTS we rely on online statistics during the rollout phase, which is not ideal for a real-time solution. The initial results were very poor; with a 33% success rate, it performs worse than the baseline. However, by restricting the action set to actions that have limited impact on the smallest TTC (no increase or minimal increase) we reach an 85% success rate.

Fig. 2 provides the test-by-test score (the cumulated reward $\sum R(s,a)$, higher is better) for one hundred multiple crossing-point episodes. As expected with MPC, when a solution is found, the results are close to optimal. MCTS-NNET results correlate with DDQN results, but are better due to the additional tree search iterations.

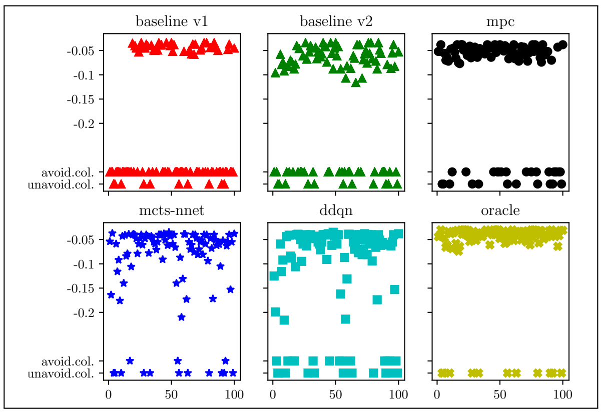


Fig. 2. Scores (higher is better) for Multiple Crossing-Points Tests

DDQN was difficult to train: we had to tune the state space representation to a sorted list of three objects (adding more objects neither improve nor degrade the results) based on minimum TTC values, to converge to a good solution. Ultimately we obtain the best results by combining MCTS and DDQN. We avoid time-consuming rollouts that depend on online statistics, since with the DDQN, we leverage an efficient parametrization of the Q-value function to guide the tree search. In addition to MCTS rollouts and $Q(s,a)$ initialization modifications, we also modified MCTS exploration such that when the Q-network provides a minor difference between $\min_a Q(s,a)$ and $\max_a Q(s,a)$, we increase the level of exploration. With this approach, the rollout computational effort is shifted offline.

2) Intersection and ACC tests: We generate 100 test scenes with cars coming from a four-way intersection, and cars ahead of the ego vehicle. As shown in Figure 3, the agents have to deal with several crossing-points at intersection simultaneously with an ACC task (driving safely behind other cars). Crossing-points are more concentrated in these tests, compared to the previous series of tests.

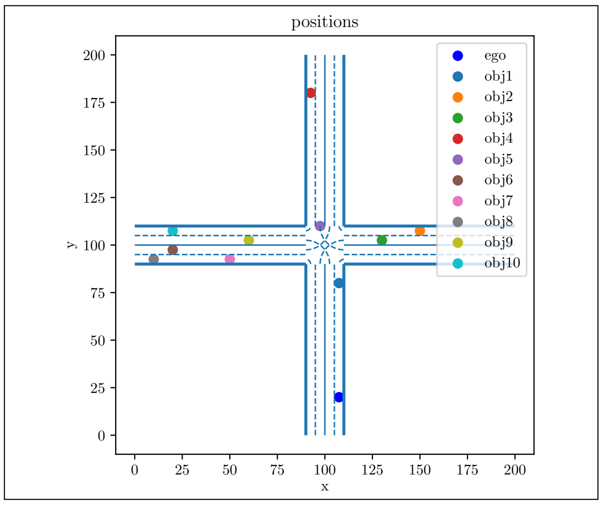


Fig. 3. Intersection and ACC Tests

TABLE IV
RESULTS WITH INTERSECTION AND ACC TESTS

Mean over 100 Tests	Success	Runtime	Hardbrakes	Steps	Collision Speed
Baseline v1	35%	40 μs	0	59	6.64 m.s^{-1}
Baseline v2	97%	40 μs	17.67	65	11.17 m.s^{-1}
MCTS	98%	3 ms	8.14	70	13.62 m.s^{-1}
DDQN	88%	300 μs	11.50	73	11.81 m.s^{-1}
MPC	82%	18 ms	8.21	64	5.65 m.s^{-1}
MCTS-NNET v2	100%	4 ms	12.99	68	-

From Table IV, we see that MCTS-NNET is collision-free while being more comfortable than the emergency braking baseline v2 which has 3% collisions, and their efficiencies are close. The baseline focuses only on a single car, with smallest TTC, which may explain why it is unable, in some cases, to avoid a collision. NNET is using a projected state representation and was trained on different road geometries and different traffic condition, not on the specific intersection and ACC scenarios. MPC finds fewer but better solutions. These results align with previous MPC results in Fig. 2.

TABLE V
RESULTS WITH SINGLE CROSSING-POINT TESTS

Mean over 99 Tests	Success	Runtime	Hardbrakes	Steps	Collision Speed
Baseline v1	65%	40 μs	0	41	13.96 m.s^{-1}
Baseline v2	89%	40 μs	6.26	43	9.25 m.s^{-1}
MCTS	97%	2 ms	2.67	37	9.25 m.s^{-1}
DDQN	98%	200 μs	3.03	47	14.50 m.s^{-1}
MPC	90%	7 ms	3.48	43	10.22 m.s^{-1}
MCTS-NNET v1	100%	3 ms	2.70	46	-
MCTS-NNET v2	100%	3 ms	2.64	38	-
Oracle (A* Search)	100%	6 sec	0.37	33	-

3) Single crossing-point tests: Table V shows the single crossing-point tests results. To our surprise, MCTS without NNET clearly outperforms MPC on the single crossing-point tests. In [1], combining a lower level MPC controller with a higher level, e.g. DRL, agent, improved accuracy. The MPC agent is instructed to proceed or to yield with respect to every other crossing vehicle, to keep the problem formulation over a convex set. In our current implementation we first look for a solution where we proceed, and then, if no solution is found, for a solution where we yield. Ultimately if no solution is found by the MPC solver, we come back to the baseline decision rule. We think the MPC results can be improved with further investigation.

We observe that MCTS and DDQN in standalone can solve 97% and 98% of the tests, while MCTS-NNET can solve all of them while ensuring a good level of comfort, a limited number of hard-brakes, and a reasonable number of steps. We see from Figure 4 that the MCTS-NNET agent has the best trade-off in terms of accuracy and runtime.

B. Discussion

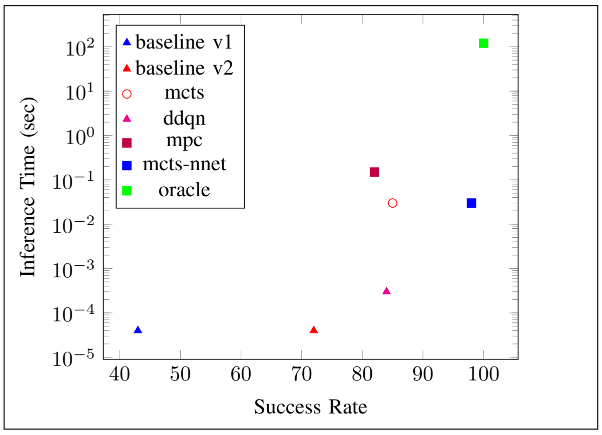
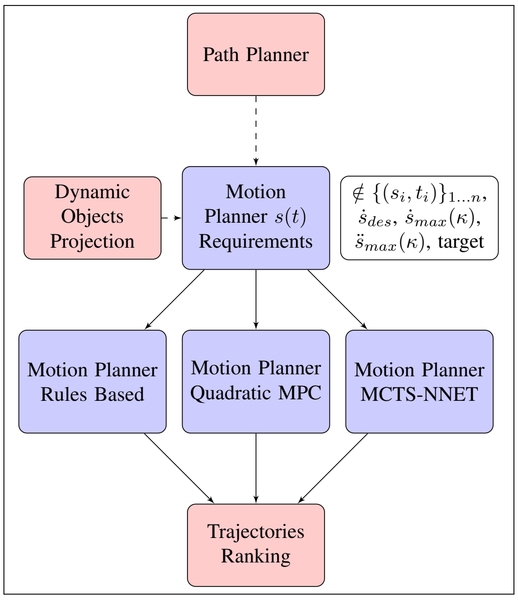


Fig. 4. Runtime vs Accuracy (Multiple Crossing-Points Tests)

While MCTS-NNET provides the best trade-off between runtime and accuracy, it does not achieve 100% accuracy in all cases. This is why we propose ranking different candidate motion plans, proposed by different agents, against predefined criteria, as depicted in Figure 5. The performance of the ranked planner can be extrapolated from the results in Tables III, IV, V, but is a topic for further investigation.

MCTS-NNET is still much slower than DDQN or the baseline in standalone. When planning velocity profiles for a set of tens of candidate paths, we may want to have even faster motion planning techniques than MCTS-NNET. One candidate technique would be to solve offline a discretized version of the MDP model via Value Iteration and complement it online via interpolation. The challenge here is the tractability of the MDP model, which has to consider a huge set of possible states. This is a topic for further investigation.


Path Planner
Dynamic Objects Projection
Motion Planner $s(t)$ Requirements
$\notin \{(s_i,t_i)\}_{1...n}$, $\dot{s}_{des}$, $\dot{s}_{max}(\kappa)$, $\ddot{s}_{max}(\kappa)$, target
Motion Planner Rules Based
Motion Planner Quadratic MPC
Motion Planner MCTS-NNET
Trajectories Ranking
Fig. 5. Motion Planner

VI. Conclusion

Real-time applicability of a MCTS tree search algorithm is enabled with a neural network heuristic. Deep Learning is used to learn a faster parametric model of a slower but known and accurate model. The complexity of the problem is reduced by projecting in the (s, t) domain the trajectories of the surrounding objects onto the ego path defining a set of spatio-temporal constraints. This projection step accounts for the different sources of uncertainties such that the Motion Planning system is left with unambiguous specification requirements. In our proposed system, predictions and decisions are clearly separated, which is usually not the case when applying Deep Reinforcement Learning or pure Deep Learning to motion planning. Previous work [2]–[5] learn to make predictions and to take decisions at the same time.

The proposed system is bench-marked against six other agents, covering a wide spectrum of techniques in challenging scenarios where we have to deal with multiple crossing-points. Our results showed that the proposed agent outperforms other agents for many different metrics: safety, comfort, and efficiency, with a runtime below 40 ms and a success rate close to 100% over a set of 300 tests. There is nevertheless no unique solution that performs best in all cases. As a consequence, we propose to combine different agents in a competing setting where a trajectory ranking system determines the best motion plan for the current situation from multiple agents.

References

[1] Tommy Tram, Ivo Batkovic, Mohammad Ali, and Jonas Sjöberg. Learning when to drive in intersections by combining reinforcement learning and model predictive control. *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 3263–3268, 2019.

[2] Andreas Folkers, Matthias Rick, and Christof Büskens. Controlling an autonomous vehicle with deep reinforcement learning. 06 2019.

[3] Carl-Johan Hoel, Krister Wolff, and Leo Laine. Automated speed and lane change decision making using deep reinforcement learning. *CoRR*, abs/1803.10056, 2018.

[4] Mariusz Bojarski, Davide Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Larry Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. 04 2016.

[5] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. *CoRR*, abs/1807.00412, 2018.

[6] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):33–55, March 2016.

[7] Branka Mirchevska, Christian Pek, Moritz Werling, Matthias Althoff, and Joschka Boedecker. High-level decision making for safe and reasonable autonomous lane changing using reinforcement learning. 09 2018.

[8] Maxime Bouton, Jesper Karlsson, Alireza Nakhaei, Kikuo Fujimura, Mykel J. Kochenderfer, and Jana Tumova. Reinforcement learning with probabilistic guarantees for autonomous driving. *CoRR*, abs/1904.07189, 2019.

[9] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, L Robert Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[10] Carl-Johan Hoel, Katherine Rose Driggs-Campbell, Krister Wolff, Leo Laine, and Mykel J. Kochenderfer. Combining planning and deep reinforcement learning in tactical decision making for autonomous driving. *CoRR*, abs/1905.02680, 2019.

[11] Panpan Cai, Yuanfu Luo, Aseem Saxena, David Hsu, and Wee Sun Lee. Lets-drive: Driving in a crowd by learning from tree search. *CoRR*, abs/1905.12197, 2019.

[12] 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)*, pages 1013–1020, June 2019.

[13] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, and Raia Hadsell. Learning to navigate in complex environments. *CoRR*, abs/1611.03673, 2016.

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

[15] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[16] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[17] Andreas Wachter and Lorenz Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106:25–57, 03 2006.

[18] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.