Learning to Optimize Motion Planning

Mingyang Wang
Department of Informatics
Technical University of Munich
Munich, Germany
mingyang.wang@tum.de

Yufan Zhao
Department of Informatics
Technical University of Munich
Munich, Germany
ge34qom@mytum.de

***Abstract*—In the paradigm of optimization-based motion planning, motion planning is formulated as an optimization problem and a general optimization algorithm like gradient descent is applied to minimize the objective function. As other optimization problems, local optima are the main challenge, which would result in infeasible solutions for motion planning problems. In this work, we make use of reinforcement learning to learn an optimization algorithm that is specialized in optimizing motion planning objectives. We finally show that more than 30% of the local optima that stuck gradient descent can be efficiently avoided by the learned optimization algorithm. We also show the learned algorithm increases the planning speed by a factor of 2.4 compared with gradient descent. To further improve the motion planning success rate, we tried multi-initialization and adding obstacle cost to the observation space, which further improved the result as expected. We also tried to use the image of the workspace as observation, which is closer to image-based motion planning. This experiment is not successful, and we discussed the issue at the end.**

Keywords—Motion planning, Optimization, Reinforcement Learning

I. Introduction

A. Optimization-based Motion Planning

In this work, we focus on motion planning for simple 2D robots which move in the field of $[0,64]^2$. In this field, there are 10 random rectangle obstacles with random locations and sizes ($[10,20]$). With random start- and end-point, a motion planning task is formulated. Then the motion planning problem is converted to an optimization problem by an objective function that penalizes longer trajectories that collide with obstacles.

Let the trajectory ξ be discretized to a set of points $\{ x_0, x_1 \dots x_N, x_{N+1} \}$, with x_0 represents the start point and x_{N+1} the endpoint. On the line segment between x_i and x_{i+1}, we sample additional subpoints $\{x_1^{i,i+1} \dots x_S^{i,i+1}\}$. Trajectories that collide with or narrowly collide with obstacles are penalized by the obstacle cost $F_{obs}(\xi)$ defined as:

$$F_{obs}(\xi) = \frac{1}{N}\sum_{i=1}^{N} c(x_i) + \frac{1}{(N+1)S}\sum_{i=0}^{N}\sum_{j=1}^{S} c(x_j^{i,i+1})$$

Here $c(x)$ is the cost for a certain point on the trajectory, it is defined as:

$$c(x) = \begin{cases} -\mathcal{D}(x) + \frac{\varepsilon}{2} & if\ \mathcal{D}(x) < 0 \\ \frac{1}{2\varepsilon}(\mathcal{D}(x) - \varepsilon)^2 & if\ 0 < \mathcal{D}(x) < \varepsilon \\ 0 & otherwise \end{cases}$$

Here $\mathcal{D}(x)$ is the distance between a certain point x and its nearest obstacle. ε is a hyperparameter that is set as 1 in our case. A longer trajectory is penalized by length cost $F_{len}(\xi)$ defined as:

$$F_{len}(\xi) = \sum_{i=0}^{N} (x_{i+1} - x_i)^2$$

The final objective function is defined as the weighted sum of the above two terms:

$$U(\xi) = F_{obs}(\xi) + \lambda F_{len}(\xi)$$

B. Reinforcement Learning for Optimization

Given the previous objective function $U(\xi)$, a partially observable Markov decision process for reinforcement learning is formulated. The state $\mathcal{S}$ is the trajectory $\xi = \{x_1 \dots x_N\}$, the observation $\mathcal{O}$ is the gradient of the objective function $\frac{\partial U}{\partial \xi}$. The action $\mathcal{A}$ is the update step $\Delta\xi$ for optimization and state transition $\mathcal{P}$ is $\xi = \xi + \Delta\xi$. The reward $\mathcal{R}$ is set as $-U(\xi)$ when a feasible solution is not found and $-\mu U(\xi)$ when a feasible solution is found. We set μ= 0.1 and μ= 1 for comparison. We hope we could use a small reward factor $\mu < 1$ to encourage the RL agent to find a feasible trajectory. Lastly, we set the discount factor as 1.

From reinforcement learning's perspective, general optimization algorithms like gradient descent can also be considered as an agent which is in state ξ, observes the gradient $\frac{\partial U}{\partial \xi}$, takes the action $\Delta\xi = -\gamma \frac{\partial U}{\partial \xi}$ and translates to $\xi + \Delta\xi$. With help of deep reinforcement learning, the final action $\Delta\xi$ is no more a linear transformation of gradient. Instead, it's a nonlinear transformation learned by the policy network, which is specialized in optimizing a certain class of objectives.

We also tried giving the agent more information with observation space including multi-step gradient and the image of workspace, hoping to achieve a higher planning success rate. Results are summarized in section III.

II. Implementation [1]

A. Training Data

We first generate 100 motion planning environments and there corresponding objectives are used as training objectives for reinforcement learning. We generate 3 training sets: *easy*, *hard* and *mix* training set. The *easy* set are generated by the strategy described in section I.A. The *hard* training set additionally needs to fail 200-step gradient descent. The *mix* training set consists 50 samples from *easy* and 50 from *hard*. For each objective function in the training set, we generate 100 supervision datapoints to pretrain the policy network(see next section II.B). The training datapoints are pairs of observations and actions $(\frac{\partial U}{\partial \xi}, -\gamma \frac{\partial U}{\partial \xi})$, used to clone the gradient descent behavior to the policy network.

1 Code can be found at https://github.com/DDD26/tum-adlr-ws20-02

B. Training

First, we perform supervised pretraining to clone the gradient descent behavior to the policy network. We use 10k training data points to train the policy network for 1 epoch. Then we further train the pretrained agent with standard RL algorithm PPO. We set the buffer size as 40, hoping the agent should find a feasible solution within 40 steps. The value and policy networks are 3-layer fully connected networks with no shared layer. The hidden dimension is set as 64 as the default value for both networks. The activation for the first 2 layers is tanh and the output layer has no activation. Weights are initialized with identity matrix and bias are initialized as zero.

C. Testing

We set up 3 test benchmarks called *easy*, *hard* and *time* test benchmarks. The *easy* test benchmark consists of 1000 motion planning environments, which are generated in the same way as *easy* training environments. The *hard* test benchmark consists of 100 motion environments generated in the same way as *hard* training environments. On both test benchmarks, we compare gradient descent and the learned algorithm's success rate. The *time* test benchmark consists of 100 environments, in which both gradient descent and the learned algorithm should find a feasible solution. Moreover, in these environments, gradient descent should use at least 10 steps to find a feasible solution. This requirement is used to rule out very simple planning tasks, to which the straight line initialization is the solution. On this benchmark, we compare the averaged iteration needed to find a feasible solution. We use seed 0 to generate training environments, seed 1 to generate environments on *time* benchmark and seed 2 to generate environments on the *easy* and *hard* benchmark.

III. Results

A. Performance on Easy and Hard Test Benchmarks

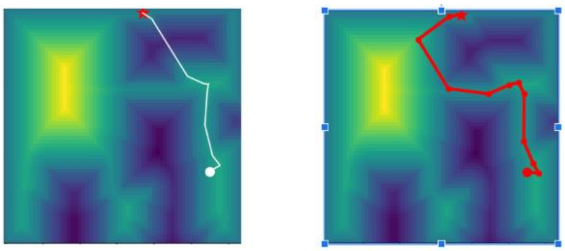
The reinforcement learning agent is trained based on different hyperparameters. We use different latent dimensions of the policy network (64, 128 and 256), reward factor (μ= 0.1 and 1) as well as several training epochs(10k and 80k). The performance of the motion planners are evaluated based on the success rate on easy and hard benchmarks, the test results are shown in Table I.

We can see that the learned algorithm outperforms gradient descent on both easy and hard benchmarks. When we set the training epoch to 80k, the reinforcement learning agent shows the best result. In this case, the success rate increased by about 5 percentage points on the easy benchmark; On the hard benchmark, more than 30% of failure cases of gradient descent are solved by the reinforcement learning agent. Additionally, among different settings, the policy network with latent dimension 64 and reward factor 0.1 has the best performance in general. In the following, we also use them as our first choice. Figure 1 shows one case that the gradient descent falls to the local optimum while the learned optimization algorithm finds a feasible solution.

TABLE I. Performance on Easy Test Benchmark

Test Benchmark	**Easy Benchmark**					**Hard Benchmark**				
Hyperparams	**Dim=64 μ = 1**	**Dim=64 μ = 0.1**		**Dim=128 μ = 0.1**	**Dim=256 μ = 0.1**	**Dim=64 μ = 1**	**Dim=64 μ = 0.1**		**Dim=128 μ = 0.1**	**Dim=256 μ = 0.1**
Training Epochs	**10k**	**10k**	**80k**	**10k**	**10k**	**10k**	**10k**	**80k**	**10k**	**10k**
Easy Training	81.4%	82.7%	**84.3%**	81.4%	81.5%	20%	23%	**30%**	22%	24%
Mix Training	80.9%	83.2%	**84.8%**	81.7%	82.6%	24%	27%	**33%**	25%	25%
Hard Training	80.1%	82.0%	**83.3%**	81.7%	81.8%	27%	25%	**37%**	25%	25%
Gradient Descent	79.7%					0%				

Fig. 1. Behavior of GD (left) and RL (right) agent



B. Runtime Performance Comparison of Both Planners

To evaluate the run time of the learned algorithm, we train RL agents with *easy* environments. The hidden dimension of the policy network is set as 64 and the reward factor is set as 0.1. The learning rate of the gradient descent algorithm here is 0.1. We trained the RL agent for 10k, 30k, 50k, and 80k epochs and compare their performance with gradient descent respectively (results are shown in Table II). From the table we can see, the RL-80k agent can increase planning speed with a factor of 2.4.

TABLE II. Runtime Performance of Both Motion Planners

Algorithm & Training Epochs	**Time per iteration**	**Iterations to find a solution**	**Average time**
Gradient Descent	1.31 ms	26.43	34.62ms
RL – 10k epochs	1.71 ms	24.82	42.44ms
RL – 30k epochs		17.83	30.49ms
RL – 50k epochs		10.94	18.71ms
RL – 80k epochs		8.28	14.66ms

C. Multiple Random-Start Initialization

We also evaluated the learned optimization algorithm with multiple initializations. Previously, the initial trajectory is

simply a straight line that connects the start point and endpoint. Now multiple random initializations are used.

In our experiment, we use N=5 and N=10 and compare their performance with straight-line initialization as well as the gradient descent algorithm (results are shown in Table III). Here N means that we use 1 straight line initialization and N-1 random initializations for the learned algorithm. From the table, we can see that with multiple random initializations, the learned algorithm successfully finds feasible solutions on nearly 90% randomly generated environments (easy benchmark) and 50% hard cases.

TABLE III. PERFORMANCE ON MULTIPLE RANDOM STARTS

Benchmark	**Easy Benchmark**			**Hard Benchmark**		
Random Start	**N=1**	**N=5**	**N=10**	**N=1**	**N=5**	**N=10**
Easy Training	82.7	87.3	88.8	22	38	44
Mix Training	82.7	88.0	**89.1**	26	40	47
Hard Training	82.3	86.7	88.2	21	40	**48**
Gradient Descent	79.7	84.4	86.2	0	32	41

D. Larger Observation Space

To further improve the performance of the policy network, we also tried to expand our observation space, by providing more information to the agent. Previously, the observation space only involves the gradient of the current step. Now it also includes obstacle cost, multistep gradient, or obstacle sizes and positions.

For multistep gradient agent, the performance of the learned algorithm degrades, which is the same case if we add the obstacle position and size information into the observation space. On the other hand, the expanded observation space with obstacle cost leads to better performance on the *hard* test benchmark as Table IV shows. The RL agents are trained in easy environments with the reward factor of 0.1 and evaluated with 5 random initializations. By adding obstacle cost, with just 5 random initializations, agents outperform gradient agents with 10 initializations as Table III shows.

TABLE IV. PERFORMANCE ON EXPANDED OBSERVATION SPACE

Benchmark	**Easy Benchmark**			**Hard Benchmark**		
Policy Network Dimension	**64**	**128**	**256**	**64**	**128**	**256**
Gradient	87.3	87.9	88.8	38	45	41
Grad+Obstacle Cost	88.5	87.6	88.3	**55**	**57**	**53**
Gradient Descent	84.4			32		

E. Image-based agents

We also tried to use the image of the workspace as the agent's observation. The image encodes obstacle locations, sizes and current trajectory. We first use a 2-layer CNN with ReLu activation to extract features from the workspace image. The CNN is shared by the policy and value network. The extracted feature then goes through 3 unshared fully connected layers to output value and action mean and standard deviation.

We also tried setting both the image and gradient as observation. The gradient goes through a skip connection from the input layer to the last layer. Both variants did not produce very good results. For the pure image agent, after reinforcement learning, extreme behavior is observed. However, if the gradient is added, the agent has more reasonable behavior but is still a bad planner. We discuss the reason for the failure in the next section.

IV. DISCUSSIONS

We attribute the failed experiments, i.e. image-based agents and multistep gradient agents to not enough pretraining. According to the curse of dimension, pretraining data needed grows exponentially as the observation space gets larger. With insufficient training data, the policy network cannot be trained well, leading to the distribution mismatch problem [4]. The observations in expert data are assumed to lie in a low dimensional submanifold in a high dimensional observation space. Through pretraining, the policy network can take good action when seeing observations close to the manifold. However, when conducting the policy, the agent is going to leave the manifold since this is a noise process. In such case, the agent would face more strange observations, making him take more strange actions, which further push him out of the manifold. The resulting rollout for reinforcement learning is a bad one. Thus, with a vanilla reinforcement learning algorithm, the agent should be initialized well to produce good trajectories for policy gradient estimation.

To make our agent working with larger observation space. We may borrow the idea of [4] to do the pretraining recursively, mitigating the distribution mismatch problem. Or we may borrow [5]'s idea to incorporate supervision mechanism into the reinforcement learning framework. [5] proposes to fix any failed rollouts in replay buffer by expert motion planning algorithm. At the very beginning, when conducting the policy, almost all rollouts generated by the current policy failed to be successful and are fixed by an expert motion planning algorithm. Performing reinforcement learning on those fixed rollouts is very close to doing supervised learning.

In summary, to train an agent with a very large observation space, a stronger reinforcement learning framework is needed. Since the limitation of time, we did not have time to incorporate [5]'s TensorFlow implementation into our PyTorch framework.

REFERENCES

[1] Zucker, M., Ratliff, N., Dragan, A. D., Pivtoraiko, M., Klingensmith, M., Dellin, C. M., Bagnell, J. A., and Srinivasa, S. S. (2013). Chomp: Covariant hamiltonian optimization for motion planning. International Journal of Robotics Research, 32(9-10):1164-1193.

[2] Li, K. and Malik, J. (2016). Learning to optimize. CoRR, abs/1606.01885.

[3] Li, K., & Malik, J. (2017). Learning to optimize neural nets. arXiv preprint arXiv:1703.0

[4] Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning." *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2011

[5] Jurgenson, Tom, and Aviv Tamar. "Harnessing reinforcement learning for neural motion planning." *arXiv preprint arXiv:1906.00214* (2019