REINFORCEMENT LEARNING FOR NETWORK EDGE CALIBRATION

James Baker, Alex Sim (Advisor), John Wu (Advisor)
Lawrence Berkeley National Lab

ABSTRACT

Predicting traffic patterns is a complicated problem. Traditionally, simulations use an agent-based approach, with commuters trying to find the most efficient routes while also adjusting their travel patterns to avoid congestion and crowding. Given an efficient method to do this, we would like to study how the graph can be optimized, specifically so we can find parts of the graph where the properties of the nodes and edges of the graph are incorrect and need to be adjusted. Given real data of flow, we want to see how we can adjust the parameters using a reinforcement learning framework in order to match the expected traffic outcome. All code is at https://github.com/jamesBaker361/rlib-frankwolfe-beam

1. INTRODUCTION

Predicting traffic patterns is a complicated problem. Traditionally, simulations use an agent-based approach, with commuters trying to find the most efficient routes while also adjusting their travel patterns to avoid congestion and crowding. The hope is to be able to produce simulations that tell us realistic, approximately correct estimates of travel times for commuters and congestion for roads. Works such as [1, 2, 3, 4] have managed to develop an efficient algorithm to do so, based off og the frank-wolfe algorithm for traffic assignment [5], using contraction hierarchies to find shortest paths. This function takes as inputs the area being studied (such as a city or metropolitan area), represented as a directed graph, where edges have properties head, tail, length, capacity and speed (As in Table 2), and the travel demand, represented as a matrix of origin-destination pairs, each with a volume to represent the amount of commuters who want to travel from the origin to the destination (As in Table 1).

Origin	Destination	Volume
141285	141322	1515
140371	140377	1501
140409	140411	1501
140531	140532	1501
...	...	...

Table 1: OD Matrix

Given this, it is important that we have accurate inputs to the routing algorithm. The edges of the graph (which correspond to roads) have parameters such as capacity, speed limit and length, which affect the output of the routing algorithm. If any of these parameters are wrong, the routing algorithm will not produce accurate results. In some cases, we have real world data regarding traffic flow, but it is unknown whether our graph has accurate parameters. If the graph is incorrect, then we cannot model new scenarios accurately, like more commuters or a change in infrastructure.

edge tail	edge head	length	capacity	speed
139084	139183	11	800	15
139085	139192	12	800	15
139092	139093	5	800	15
139092	139200	7	800	15
...	...	...	...	...

Table 2: Edge Network

Reinforcement learning (RL) is a type of machine learning where an agent performs actions, augmenting its environment in some way, and the receiving a reward. The agent is then trained to maximize the reward like how a classifier is trained to minimize loss. In this context, Reinforcement Learning is ideal for optimizing the parameters of an incorrect network in order to bring them closer to the actual parameters of the real network. This work attempts to train an RL agent to adjust the parameters of the network with the reward function.

2. MATERIALS

As this project requires simulations, it requires mainly computing power. For that, we are running our code on the National Energy Research Scientific Computing Center (NERSC) high-performance computing cluster.

3. METHODS

To actually try this out, we are using a network that represents Austin, Texas. The Nodes in the network are seen on 1 The physical network data came from Open Street Map and the demand matrix came from the Behavior, Energy, Autonomy, and Mobility (BEAM) team at Berkeley Lab.

Given the massive size of this network, we will be shrinking the network and only using a small subset of nodes and

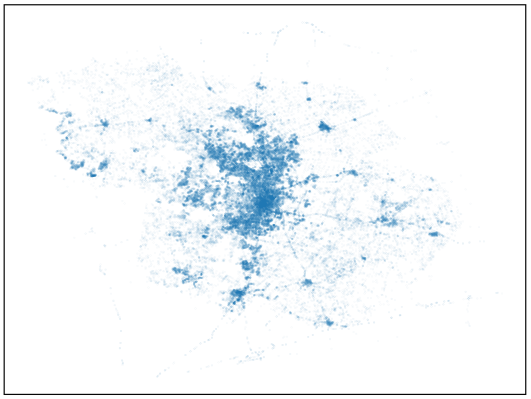


Fig. 1: The entirety of Austin

demand pairs. First, we isolated only the largest connected component. Then, we trimmed 99.5% of all nodes by repeatedly pruning nodes that had only one outgoing edge. This can be seen (on a much smaller level) in Figure 2. Subfigure 2a shows the initial state of the network at T=0. Subfigure 2b shows the same network, but the nodes that will be deleted have been colored red. Subfigure 2c shows the network at T=1, after the red nodes in Subfigure 2b have been deleted. Subfigure 2d shows the same network as in subfigure 2c, but the nodes to be deleted are in red.

We reduced the demand matrix to only include paths that were in the pruned network. Since this would lead to an extremely small amount of demand, we artificially boosted the volume of commuters of each demand pair.

A Reinforcement Learning problem requires a state space, an observation space (which is the state space in some scenarios), an action space and a reward function. Starting with an initially perturbed copy of the original network, where the network has e quantity of edges, we copied it and randomly selected a fraction of the edges to have a capacity=100. This perturbed network at any timestep s_t is the state space $\mathcal{S}$.

$$s_t \in \mathcal{S} : \mathbb{R}^e$$

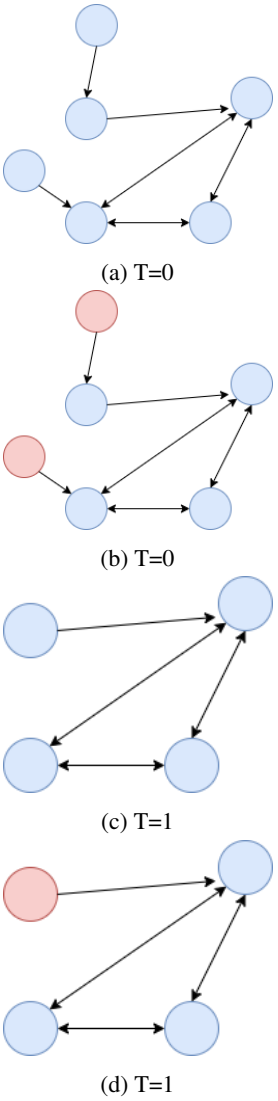
The observation o_t at any timestep was in the space Ω, the traffic flow predicted by the routing algorithm using the perturbed network.

$$o_t \in \Omega : \mathbb{R}^e$$

The action at any timestep a_t was in the space $\mathcal{A}$ were values between 0 and 1 for each edge; for the agent to choose a value $a_{t,i}$, for edge i, the capacity of edge i would be set to $base + (a_{t,i} * range)$.

$$a_t \in \mathcal{A} : \mathbb{R}^e; a_{t,i} \in [0, 1]$$

We obtained an actual "ground truth" flow using the actual, unperturbed network as an input to the routing algorithm. The



(a) T=0

(b) T=0

(c) T=1

(d) T=1

Fig. 2: Pruning

instantaneous reward function r_t was then calculated by finding the L_2 norm of the "ground truth" flow, o' minus the flow predicted from the perturbed network, $o_t \in \Omega$, minus a regularization term equal to $\beta*$ the L_2 norm of the perturbed network, s_t, as to penalize the algorithm for setting the network parameters too large, where β is a hyperparameter = 0.1.

$$r_t = -||o' - o_t||_2 - \beta||s_t||_2 \in \mathbb{R}$$

While having the reward function depend on the observation space may look unconventional, o_t is a function of a_t, s_t, so this is standard. With each episode lasting T steps, where at each step the agent takes an action, thus updating the state of the world, we define the return function:

$$R = \sum_{t}^{T-1} r_t$$

Thus the return is only contingent on the reward at the end of the episode. We used two state of the art frameworks: Twin Delayed Deep Deterministic (TD3) [6], and Soft Actor Critic (SAC) [7].

3.1. TD3

The TD3 algorithm trains an actor network to map states to actions, and two critic methods to map states and actions to rewards, and trains them in parallel, but it only updates the actor half as frequently as the critic. First we make the Critic networks $Q_{\theta_1}, Q_{\theta_2}$, distinguished by two different (initially random) sets of parameters θ_1, θ_2. These critic networks return a value given a state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$

$$Q_\theta(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

We also have the actor network μ_ϕ, aka the policy, that maps states to the next action to take, with parameters ϕ, or equivalently is a probability distribution parameterized by by ϕ

$$\mu_\phi(s) : \mathcal{S} \rightarrow \mathcal{A}$$

For all of the actor and critic networks $Q_{\theta_1}, Q_{\theta_2}, \mu_\phi$, we also make target networks $Q_{\theta'_1}, Q_{\theta'_2}, \mu_{\phi'}$, with parameters $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$, initially set equal to the non-target network parameters.

We also have a replay buffer $\mathcal{B}$, which is like our dataset, it consists of tuples (s, a, r, s'), representing a state, an action, the reward from that action, and the state of the world following that action, respectively.

Then for each episode, of length T, we do one iteration.

First, we fill up the replay buffer by sampling from the distribution μ_ϕ and add random white noise and use that to determine the action

$$a \sim \mu_\phi(s) + \mathcal{N}(0, \sigma)$$

We can then find the reward function$r$, and the state of the world after the action s', and produce a tuple (s, a, r, s'), which we can add to $\mathcal{B}$. We may do this multiple times to fill up the buffer.

We get a minibatch N from $\mathcal{B}$. Using this minibatch, for each element of the minibatch, we can calculate the next action $\tilde{a}$ using the target actor network $\mu_{\phi'}(s')$ and random noise ϵ clipped to be between $[-c, c]$

$$\tilde{a} = \mu_{\phi'}(s') + \epsilon; \epsilon \sim -c \leq \mathcal{N}(0, \sigma) \leq c$$

Using this we can find the learning target $y(r, s')$, which is the current reward and the discount factor γ times the minimum target Q value from the target critic networks. This is done to prevent overestimation of the reward function.

$$y(r, s') = r + \gamma min_{i=1,2} Q_{\theta'_i}(\tilde{a}, s')$$

We can then update the Q function parameters ϕ_i using gradient descent, to minimize the squared distance between the learning target and the Q value of the current state action pair.

$$\nabla_{\theta_i} \frac{1}{|N|} \sum_{(s,a,r,s') \in N} (Q_{\theta_i}(s, a) - y(r, s'))^2$$

Every d iterations, where d is an integer hyperparameter, we then update the actor network and all the target networks. Holding the critic network constant, we update the actor network μ_ϕ using gradient ascent

$$\nabla_\phi \frac{1}{|N|} \sum_{s \in N} Q_{\theta_1}(s, \mu_\phi(s))$$

Then we update the target network parameters, using the update rate $0 < \tau < 1$

$$\theta'_1 \leftarrow \tau\theta'_1 + (1 - \tau)\theta_1$$

$$\theta'_2 \leftarrow \tau\theta'_2 + (1 - \tau)\theta_2$$

$$\phi' \leftarrow \tau\phi' + (1 - \tau)\phi$$

3.2. SAC

The Soft-Actor Critic method is similar to the TD3 method. Key differences are a lack of a target actor network and the use of an entropy, which represents the randomness or unpredictability of the system being processed.

First, like for TD3, we make the critic networks $Q_{\theta_1}, Q_{\theta_2}$, and the actor network μ_ϕ, parameterized by θ_1, θ_2, ϕ, and target networks for the critics $Q_{\theta'_1}, Q_{\theta'_2}$, with parameters θ'_1, θ'_2. We also have the replay buffer $\mathcal{B}$.

Then for each episode, of length T, we do one iteration.

First, we fill up the replay buffer by sampling from the distribution μ_ϕ. We do not add noise, as we did in TD3.

$$a \sim \mu_\phi(s)$$

We can then find the reward function$r$, and the state of the world after the action s', and produce a tuple (s, a, r, s'), which we can add to $\mathcal{B}$. We may do this multiple times to fill up the buffer.

We get a minibatch N from $\mathcal{B}$. Using this minibatch, for each element of the minibatch, we can calculate the next action a' using the actor network $\mu_\phi(s')$

$$a' = \mu_\phi(s')$$

Using this we can find the learning target $y(r, s')$, which is the current reward and the discount factor γ times the minimum target Q value from the target critic networks, minus an entropy term $\alpha \log \mu_\phi(a'|s')$, where α is a hyperparameter.

$$y(r, s') = r + \gamma min_{i=1,2} Q_{\theta'_i}(a', s') - \alpha \log \mu_\phi(a'|s')$$

We can then update the Q function parameters ϕ_i using gradient descent, to minimize the squared distance between the learning target and the Q value of the current state action pair.

$$\nabla_{\theta_i} \frac{1}{|N|} \sum_{(s,a,r,s') \in N} (Q_{\theta_i}(s,a) - y(r,s'))^2$$

Then we update the actor network and all the target networks. Holding the critic network constant, we update the actor network μ_ϕ using gradient ascent

$$\nabla_\phi \frac{1}{|N|} \sum_{s \in N} min_{i=1,2} Q_{\theta_i}(s, \mu_\phi(s)) - \alpha \log(\mu_\phi(s)|s)$$

Then we update the target network parameters, using the update rate $0 < \tau < 1$

$$\theta_1' \leftarrow \tau\theta_1' + (1-\tau)\theta_1$$

$$\theta_2' \leftarrow \tau\theta_2' + (1-\tau)\theta_2$$

4. RESULTS

Results have been good, but strange. The algorithm improves sharply, and then plateaus, for both SAC and TD3. Note that the rewards do not start out the same, as both algorithms start out with different randomly chosen actions.

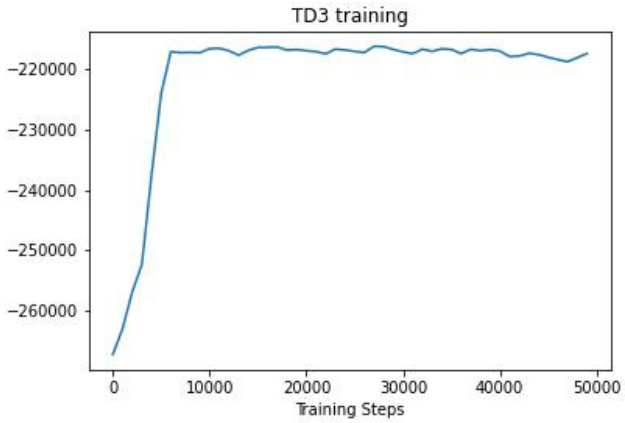


Fig. 3: TD3: Episode Return (y axis) vs Steps (x axis)

5. CONCLUSION

In this work, we showed that Reinforcement Learning can be applied to an optimization problem for validating inputs for transportation modeling problems. This will be useful in real-world scenarios, where we want to use the routing algorithm on new cities represented as new networks. By having a method to validate the road networks, we are removing a major bottleneck to doing so. Further work would be to use this

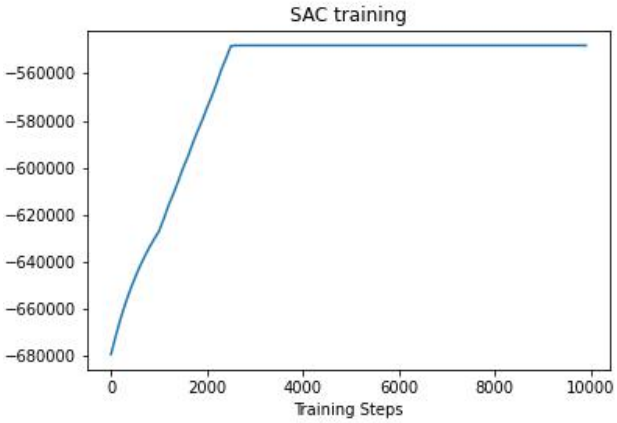


Fig. 4: SAC: Episode Return (y axis) vs Steps (x axis)

methodology to validate demand input, while holding the parameters of the road network constant. Additionally, scaling up this process so it can handle larger sets of nodes and edges at once may also prove to be worthwhile. The current pruning algorithm breaks the road network into one small road network. A different pruning algorithm may produce more interesting or more realistic results. Further work could also experiment with scheduling or cycling learning rates to deal with the plateauing reward problem.

6. ACKNOWLEDGEMENTS

Zach Needell, C. Anna Spurlock, Xiaodan Xu and Ling Jin also contributed to this work. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and also used resources of the National Energy Research Scientific Computing Center (NERSC). This work was also supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internship (SULI) program.

7. REFERENCES

[1] Valentin Buchhold, Peter Sanders, and Dorothea Wagner, "Real-Time Traffic Assignment Using Fast Queries in Customizable Contraction Hierarchies," in *17th International Symposium on Experimental Algorithms (SEA 2018)*, Gianlorenzo D'Angelo, Ed., Dagstuhl, Germany, 2018, vol. 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 27:1–27:15, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] Valentin Buchhold, Peter Sanders, and Dorothea Wagner, "Efficient calculation of microscopic travel demand

data with low calibration effort," in *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, New York, NY, USA, 2019, SIGSPATIAL '19, p. 379–388, Association for Computing Machinery.

[3] Valentin Buchhold, Peter Sanders, and Dorothea Wagner, "Real-time traffic assignment using engineered customizable contraction hierarchies," *ACM J. Exp. Algorithmics*, vol. 24, dec 2019.

[4] Valentin Buchhold and Dorothea Wagner, "Nearest-Neighbor Queries in Customizable Contraction Hierarchies and Applications," in *19th International Symposium on Experimental Algorithms (SEA 2021)*, David Coudert and Emanuele Natale, Eds., Dagstuhl, Germany, 2021, vol. 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 18:1–18:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Masao Fukushima, "A modified frank-wolfe algorithm for solving the traffic assignment problem," *Transportation Research Part B: Methodological*, vol. 18, no. 2, pp. 169–177, 1984.

[6] Scott Fujimoto, Herke van Hoof, and David Meger, "Addressing function approximation error in actor-critic methods," *CoRR*, vol. abs/1802.09477, 2018.

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *CoRR*, vol. abs/1801.01290, 2018.