IE 3186 ADP Project Writeup
Dynamic Pricing for N-Stations Vehicle Sharing System

Ibrahim El Shar

1 Introduction and Model

Suppose that a vehicle sharing manager is responsible for setting the rental price for the vehicles at the beginning of each period in a finite planning horizon consisting of T periods of equal length. We study an N-stations car sharing system. The goal is to optimize the prices to set for renting a car at each of the N stations (p_{it} for all $i \in [N] = \{1, \dots, N\}$). Demands in each period t are nonnegative, independent and depends on the vehicle renting price according to a stochastic demand function

$$D_{it}(p_{it}, \epsilon_{it}) := \kappa_{it}(p_{it}) + \epsilon_{it},$$

where $D_{it}(p_{it}, \epsilon_{it})$ is the demand in period t, ϵ_{it} are random perturbations and $\kappa_{it}(p_t)$ is a deterministic demand function of the price p_{it} at station $i \in [N]$. The random variables ϵ_{it} are independent over time with $E[\epsilon_{it}] = 0$ without loss of generality. Furthermore, we assume that the expected demand $E[D_t(p_{it}, \epsilon_{it})] = \kappa_{it}(p_{it}) < \infty$ is strictly decreasing in the renting price p_{it} which is restricted to a set of feasible price levels $[\underline{p}_{it}, \overline{p}_{it}]$ for all $i \in [N]$, where $\underline{p}_{it}, \overline{p}_{it}$ are the minimum and the maximum prices that can be set at station i during time period t, respectively. This assumption implies a one-to-one correspondence between the renting price p_{it} and the expected demand $d_{it} \in \mathfrak{D} := [\underline{d}_{it}, \overline{d}_{it}]$ for all $p_{it} \in [\underline{p}_{it}, \overline{p}_{it}]$ where $\underline{d}_{it} = \kappa_{it}(\overline{p}_{it})$ and $\overline{d}_{it} = \kappa_{it}(\underline{p}_{it})$. The problem can be formulated as a Markov Decision Process (MDP) with state $\mathbf{x}_t$ which is a vector whose components is the number of available cars at each of the N stations, at beginning of period t. The state space is $\mathcal{X}^N = \{0, \dots, \bar{x}\}$, where $\bar{x}$ is the maximum number of cars in the vehicle sharing system. We assume that a customer at station i and time period t goes to station j with probability α_{ijt} for all $i, j \in [N]$. We penalize unmet demands by a lost sales unit cost ρ_{it}, $i \in [N]$. The decision vector is $\mathbf{p}_t = \{p_{it} \in [\underline{p}_{it}, \overline{p}_{it}], \forall i \in [N]\}$. Let $v_t^*(\mathbf{x}_t)$ be the revenue-to-go function at the beginning of period t with number of available vehicles $\mathbf{x}_t$. Let there be no end-of-horizon profit, i.e., $v_{T+1}^*(\mathbf{x}) = 0$ for all $\mathbf{x}$. Then, for each $t = 1, 2, \dots, T$, we have the Bellman recursion,

$$\begin{aligned}
v_t^*(\mathbf{x}_t) = \max_{\mathbf{p}_t \in [\underline{\mathbf{p}}_t, \overline{\mathbf{p}}_t]} \mathbf{E}\Big[\sum_{i \in [N]} p_{it} \sum_{j \in [N]} l_{ij}\, w_{ijt}(\epsilon_{it}) - \sum_{i \in [N]} \rho_{it} \Big(\kappa_{it}(p_{it}) + \epsilon_{it} - w_{it}(\epsilon_{it}) \Big) \\
+ \gamma v_{t+1}(\mathbf{x_{t+1}}) \Big] \\
x_{i,t+1} = x_{it} + \sum_{j \in [N]} \alpha_{jit} w_{jt}(\epsilon_{jt}) - w_{it}(\epsilon_{it}), \quad \forall\, i \in [N],\ t \in [T] \qquad (1.1) \\
w_{it}(\epsilon_{it}) = \min\left(\kappa_{it}(p_{it}) + \epsilon_{it}, x_{it}\right) \quad \forall\, i \in [N],\ t \in [T] \\
w_{ijt}(\epsilon_{it}) = \alpha_{ijt}\, w_{it}(\epsilon_{it}) \quad \forall\, i, j \in [N],\ t \in [T]
\end{aligned}$$

where l_{ij} is the distance from station i to j and $\gamma \in (0, 1)$ is a discount factor.

2 Car Sharing Environment

We create a simulator of our environment to generate training episodes. The environment was created following popular openai gym environment simulator. The details of the environment are shown in Table 1:

Table 1: Environment details

Simulator	
Settings	**value**
Environment Name	‘AdpCarsharing-v0’
Number of stations	5
Number of cars	50
Demand model	Additive
Deterministic demand function	Stationary, Linear: $\kappa(p) = a - bp$ where a= 17., 19., 16., 22., 23. and b= 5., 5., 4., 3., 4. for each station respectively.
ϵ support for each station	$\{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$
Minimum price $\underline{p}_t$	1 for all stations
lost sales cost ρ	3., 3., 4., 3., 4. for each station respectively.
Minimum demand $\underline{d}$	5, 5, 5, 5, 5 for each station respectively.
Maximum demand $\overline{d}$	12, 14, 12, 19, 19 for each station respectively.
Probabilities α_{ij}	Sampled from a Dirichlet distribution with parameters $k_i = 1 \, \forall \, i = 1, \ldots, 5$. Random seed = 2
Distance l_{ij}	Sampled from a Uniform(1,20) distribution. Random seed = 2

* discount factor was set to 1 in all implementations. Output activations were all linear.

3 Algorithms

Due to both the well known curse of dimensionality and the fact that the actions are continuous, the MDP in (1.1) cannot be solved exactly by Dynamic Programming (DP). This motivates us to work with Approximate Dynamic Programming (ADP) algorithms to find a good solution to this pricing problem. In this writeup, we consider two different state-of-the-art algorithms to find the optimal policy of this problem. We first consider the recent Deep Deterministic Policy Gradients (DDPG) Reinforcement Learning (RL) algorithm, proposed by Lillicrap et al. (2015) to solve RL problems with continuous actions. Next, we propose an algorithm that approximates the N-Station problem by a N two-stations problems and solves each of them exactly by dynamic programming. To mitigate the problematic continuous decision vector $\mathbf{p}_t$ in DP, we use the expected demand ($\mathbf{d}_t = \kappa(\mathbf{p}_t)$) instead of the price as the decision vector and consider only its discrete values in $\{\underline{\mathbf{d}}_t, \ldots, \overline{\mathbf{d}}_t\}$. This discretization helps us to solve the DP for each of the N problems exactly by backward induction. The optimal policy for each station in the two-station problems is then used to generate a policy for the N-station problem. Preliminary experimental results show that the resulting policy is good by itself. We try however to improve it by an Evolution Strategy (ES) algorithm that takes the DP policies as input to come up with a better one. In the next sections, we discuss the mentioned algorithms and show some results.

3.1 DDPG

DDPG is an off-policy actor-critic approach for continuous actions that combines 3 techniques: Deterministic Policy-Gradient Algorithms, Actor-Critic Methods and Deep Q-Network. DDPG can be thought of as being deep Q-learning for continuous action spaces. The difference is that in addition to learning the action value it also learns a deterministic policy. DDPG is similar to Q-learning since similarly it tries to minimize the mean square Bellman error (MSBE):

$$L(\theta, \mathcal{D}) = \mathop{\mathbf{E}}_{s,a,r,s' \sim \mathcal{D}} \Big[\Big(Q_\theta(s,a) - \big(r + \gamma \max_{a'} Q'_{\theta'}(s', a') \big) \Big)^2 \Big]$$

which tells us roughly how closely our approximator Q_θ with parameter θ satisfies the Bellman equation. Moreover, as is the case of most standard algorithms for training a deep neural network to approximate $Q^*(s,a)$ DDPG make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences. DDPG also makes use of a second network, called the target network $Q'_{\theta'}(s', a')$, which lags the main Q-network in order to stabilize the MSBE minimization. Since our action space is continuous computing the maximum over the actions in the target is problematic. To mitigate this problem DDPG uses a target policy network $\mu'_{\theta^{\mu'}}$ to compute an action which approximately maximizes $Q'_{\theta'}$. Accordingly, Q-learning is performed in DDPG by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\theta, \mathcal{D}) = \mathop{\mathbf{E}}_{s,a,r,s' \sim \mathcal{D}} \Big[\Big(Q_\theta(s,a) - \big(r + \gamma Q'_{\theta'}(s', \mu'_{\theta^{\mu'}}(s')) \big) \Big)^2 \Big]$$

In the policy learning part, we need to learn a deterministic policy $\mu_{\theta^\mu}(s)$ which outputs the action that maximizes $Q_\theta(s,a)$. To learn this policy, we perform a gradient ascent with respect to policy parameters only to solve $\max_{\theta^\mu} \mathop{\mathrm{E}}_{s \sim \mathcal{D}} [Q_\theta(s, \mu_{\theta^\mu}(s))]$. That is,

$$\frac{\partial \mathrm{E}\left[Q_\theta(s,a)\right]}{\partial \theta^\mu} = \mathrm{E}\Big[\frac{\partial Q_\theta(s,a)}{\partial a} \frac{\partial a}{\partial \theta^\mu} \Big] = \mathrm{E}\Big[\frac{\partial Q_\theta(s, \mu_{\theta^\mu}(s))}{\partial \theta^\mu} \Big].$$

Silver et al. (2014) proved that this is the policy gradient that has to be followed in updating the policy parameters to get the maximum expected reward. Note that the action-value function parameters are treated as constants here.

Since the policy trained by DDPG is deterministic and off-policy the agent will not try a wide enough variety of actions to find useful learning signals. To treat this issue, we add noise to the deterministic policy's actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise which we follow in our implementation.
The complete algorithm is shown below.

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$
    Update the target networks:
        $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
        $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
  end for
end for
```

3.1.1 DDPG Implementation Details

The implementation details of DDPG algorithm are shown in Table 2.

Table 2: DDPG Implementation details

DDPG	
Settings	**Value**
Actor NN Architecture	MLP, 2 hidden layers each of size 64 with Relu activation
Critiic NN Architecture	MLP, 2 hidden layers each of size 64 with Relu activation
Optimizer	Adam
Burn-in transitions	1200
Random process for exploration	Ornstein-Uhlenbeck process: $\theta = 0.15$, $\mu = 0$, $\sigma = 0.2$
Buffer size	1000000
Minibatch size	32
τ for updating networks parameters	0.001
Steps before minibatch	1
Actor learning rate	0.0001
Critic learning rate	0.001
Discount factor	1

3.2 DP Policy Mixing

The idea behind DP Policy Mixing (DPPM) is inspired from revenue management literature. In DPPM, the N-Station problem is approximated by an N two-stations problems that are solved exactly by dynamic programming. As mentioned earlier, we use the expected demand ($\mathbf{d}_t = \kappa(\mathbf{p}_t)$) instead of the price as the decision vector. The DP for the two-station problems are solved by discretizing the demand in $\{\underline{\mathbf{d}}_t, \ldots, \overline{\mathbf{d}}_t\}$. The two-station problems are obtained by averaging $N-1$ stations into one station. This averaging means that we approximate the demand functions, distances between the stations, and the probabilities of going from one station to the other. The station resulting from averaging these $N-1$ stations is referred to as the aggregated station. The resulting optimal policies for each station in the two-station problems is then used to generate an approximate policy for the N-station problem.

$$\pi_{approx}(x_1, x_2, \ldots, x_N) = (\pi_1^*(x_1), \pi_2^*(x_2), \ldots, \pi_N^*(x_N))$$

The N-station policy is then improved by an Evolution Strategy (ES) algorithm as in Salimans et al. (2017), also shown below.

Algorithm 2 Parallelized Evolution Strategies

```
1:  Input: Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
2:  Initialize: $n$ workers with known random seeds, and initial parameters $\theta_0$
3:  for $t = 0, 1, 2, \ldots$ do
4:      for each worker $i = 1, \ldots, n$ do
5:          Sample $\epsilon_i \sim \mathcal{N}(0, I)$
6:          Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
7:      end for
8:      Send all scalar returns $F_i$ from each worker to every other worker
9:      for each worker $i = 1, \ldots, n$ do
10:         Reconstruct all perturbations $\epsilon_j$ for $j = 1, \ldots, n$ using known random seeds
11:         Set $\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{j=1}^{n} F_j\epsilon_j$
12:     end for
13: end for
```

The ES-algorithm takes the approximate DP policy as input instead of the state to maximize the objective. Preliminary experimental results showed however that the policy mixing for our instance problem did not improve significantly or showed no improve at all from the approximate DP solution. This shows that our DP-policy estimate of the N-stations problem is near optimal. This is also confirmed by comparing to the DDPG policy which performed poorly relative to the approximate DP-policy.

One essential difference between DPPM and DDPG is that in DPPM we assume that we know the dynamics of the environment to solve the two-station problems by DP. DDPG on the other hand, is a pure RL algorithm and does not require such information. This may explain at least partially the difference in the obtained polices.

One way to transform DPPM into a pure RL algorithm would be to solve the two-station problems by an RL algorithm. Model based reinforcement learning algorithms could be a good replacement of DP.

3.2.1 DPPM Implementation Details

The implementation details of DPPM algorithm are shown in Table 3.

Table 3: DPPM ES Implementation details

DPPM	
Settings	**Value**
NN Architecture	MLP, 1 hidden layers each of size 20 with tanh activation
ES population size	50
σ	0.03
Learning rate	0.003
Discount factor	1

4 Results

The below plots show both the training and performance curves obtained by training our agent using DDPG and DPPM on our environment. For the performance curve, every k=50 episodes the policy being trained is tested on 20 episodes. The plot shows the mean and standard deviation of each of this tests. The training curve shows that the both agents were able to learn reasonable results with substantial learning progress. It took both agent around 7 epochs to learn a reasonable policy, where they stabilize there after. The DPPM agent however significantly outperforms the DDPG one. We believe that the policy produced by combining the two-stations DP policies is near optimal.
The performance curves shown in Figure. 2 show that the DDPG agent has a higher variance than that of DPPM.
Table. 4 compares performance of different policies. One can see that the policy obtained directly from solving the DP of the two-station problems outperforms all the other polices and that ES algorithm could not enhance the DP policy after 5000 training episodes.

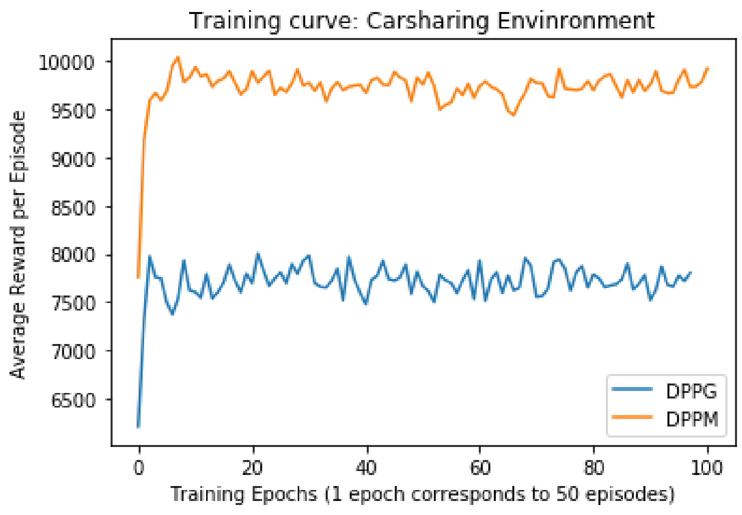

Figure 1: DPPM & DDPG Algorithms: Training curves.

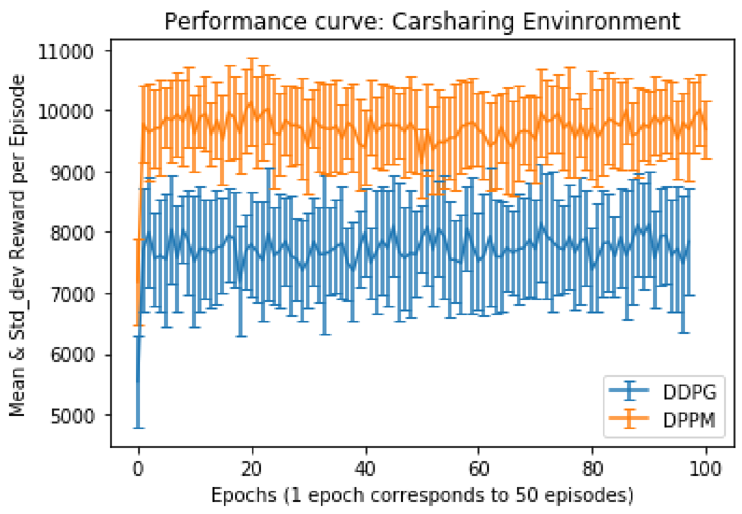

Figure 2: DPPM & DDPG Algorithms: Performance curves.

Table 4: Performance after training DDPG and DPPM for 5000 episodes. We report the mean expected total reward of 1000 episodes. We compare against the policy obtained directly from solving the DP of the two-station problems and a static policy that keeps the price fixed at $(\underline{p}_t + \overline{p}_t)/2$.

Algorithm	**Reward**
DDPG	7712
DPPM	9788
N station approx. DP policy	9914
Static policy	7202

References

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *ICML*.