# ORIE 6590 Final Report

Tao Jiang, Laurel Newman

**The link to our Github Repository is** https://github.com/taotolojiang/ORIE6590

## 1 Introduction

The Airline Revenue Management problem considers the goal of optimizing revenue for a hub which services L different destinations. The hub handles flights into and out from these L locations in both single and two leg itineraries. Every possible itinerary additionally has a high-fare and low-fare class. We use $m = 2L$ to denote the total number of single-leg itineraries and $n = 2L(L+1)$ to denote the total number of fare and itinerary combination classes that a customer could belong to.

Previous work has used dynamic programming by Talluri and van Ryzin [14], and from the dynamic program some work has been done on using linear programming to approximate it by Adelman [1]. The approximation was first considered by Schweitzer and Seidmann [12] and de Farias and Van Roy [5]. Bertsimas and Popescu [3] have also computed exact value functions while Bertsimas and de Boer [2] have worked on simulation-based methods for estimating gradients. We will specifically compare our RL approach against some of the LP approaches (including relaxation and column generation) from Table 6 in Adelman [1].

Our project used reinforcement learning to optimize the revenue gained by determining which customer classes to allow to book flights over a period while there still remains capacity for seats on some flights. The Markov decision process formulation is presented in Section 2. The specific choices of model inputs are included in Section 3. To solve the airline revenue problem, we adopted proximal policy optimization (PPO) algorithms. Section 4 gives a brief introduction of PPO and states the objective function. Our main contributions of the project are exploring other methods of advantage estimation to reduce variance and tuning hyper-paramters of PPO to boost performance. Based on the existing `Stable-Baselines3.PPO`, we added our own script for advantage estimation. We elaborate various methods of advantage estimation and report the numerical results in Section 5. In Section 6, we illustrate the effect of multiple hyper-parameters on the performance of the algorithm. Such hyper-parameters include learning rate, rollout buffer length, GAE parameter $\lambda$, discount factor and optimization algorithms. Finally, we report the numerical experiments in Section 8.

## 2 Markov Decision Process Formulation

The state space is the set of all possible available seats for every flight into and out of each location up to the full capacities. The action space is all possible binary vectors of length n which tells you whether a customer (with a specific fare and itinerary) is accepted or declined by the airline company. The one-step reward is the revenue gained from applying the predetermined action (of this time-step) to a customer who appears during this time-step (at most one will do so). The transitions are of the form as shown below. We include our choice of parameters in Section

- state $= c = [...c_i...c_j...]$;
- action$[k] = 1$;
- customerClass $= k$;
- customerDemand $= d = [0, ..., d_i, ..., d_j, ..., 0], c - d \geq 0, i \neq j, d_i, d_j$ not both 0);
- **newState** $= c - d$ (noticeably, this will only differ at indices $i$ and $j$)

All other transitions lead to **newState** = state.

## 3 Model

We specifically wanted to compare against Table 6 from [1], so we constructed our model to be the same as theirs.

We first describe the parameters which we vary to describe different problem instances.

1. The number of connected travel locations to our central hub is denoted $L$, and there are thus $2L$ different incoming and outgoing aircraft (that is, $2L$ "legs" of any possible flight itinerary). Recall that there all itineraries are either one- or two-legged and that each itinerary has a low- and high-fare version, such that we have $n = 2L(L + 1)$ possible bookings for any customer to choose from.
2. Each aircraft is given a fixed, identical number of seats, and we let $\kappa$ denote the capacity of each leg.
3. We also have a deadline by which all bookings must be made. That is, our problem instance is given a specific time limit within which all flight bookings must occur, and we denote our booking epoch by $\tau$.

Thus, a single version of this model is described by the tuple $(L, \tau, \kappa)$ and specific instances of these versions are then determined by randomly generating the following data:

- Revenue gained from any of the $n$ possible bookings: the revenue gained from the low-fare itineraries was generated by `numpy.random.randint` on the interval $[15, 50)$. The revenue for the high-fare itineraries was generated by scaling the previous vector by 5.

- Probability of each of $n$ possible booking: at each time step at most one customer arrives and makes a single booking. Firstly, we fixed the probability that no customer arrived to be 0.2. Next, we generated the probability of any specific itinerary by scaling `numpy.random.uniform` into a distribution with $n/2$ discrete values and a sum of 0.8. We then weighted the distribution by 0.75 to construct the probability for low-fare itineraries and weighted the distribution by 0.25 to construct the probability for high-fare itineraries.

It is worth noting that while Adelman was able to compare all their approximations on a specific instance of this model, we were not able to compare to this exact instance (since it is randomly generated), so instead we generated our own random instance for both the $L = 3$ and $L = 5$ version of the model, and considered the different cases over both of these instances (see Appendix A).

## 4 Policy Proximal Optimization

To solve the airline revenue management problem, we employed the proximal policy optimization algorithm algorithm (PPO) from Stable-Baselines3. The algorithm PPO combines the idea of A2C and TRPO and solves problems with multi-discrete action space.

Prior to PPO, there are two main types of methods to solve policy optimization: policy gradient methods and trust region methods. The policy gradient methods adopt a stochastic gradient ascent (SGD) algorithm and compute an estimator of the policy gradient for the SGD update. The trust region policy optimization methods (TRPO) maximize the ”surrogate objective” with the constraints that force a conservative policy update.

In the theory of TRPO, Schulman et al. [9] suggests a reformulation of the constrained problem by adding a penalty to the objective function. Moreover, unconstrained optimization problem Unfortunately, simply choosing a fixed penalty coefficients is not sufficient from various numerical experiments. Moreover, optimizing the naive unconstrained version of TRPO using SGD fails to work.

PPO was inspired by TRPO and first proposed by Schulman et al [11]. PPO attains the efficient data sampling and reliable performance that TRPO achieves, while it only uses a first-order methods. Moreover, PPO is also “simpler to implement” and “has better sample complexity” compares to TRPO. It minimizes the following objective function:

$$\max_{\theta} \hat{E}_t \left[ \min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{1}$$

## 5 Advantage Estimation

We explored different ways of estimating the advantage function, with the aim of exploring which method of variance reduction worked best for our problem. Four types of advantage estimation are considered: generalized advantage estimator, simple cumulative moving average, weighted moving average and double exponential moving average. We rewrote the function `compute_returns_and_advantage` in class `RolloutBuffer`, which is located in the file `buffers.py` in `SB3.commons`.

Recall that the estimate of advantage of the action $a_t$ is defined as

$$\delta_t^V := rt + \gamma V(s_{t+1})V(s_t),$$

where $V$ is an approximate value function. A $k$-step estimate of the returns, minus a baseline is defined as

$$\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V.$$

As $k \to \infty$, we get $A_t^{(\infty)}$ which is the empirical returns minus the value function.

### Generalized advantage estimator

The generalized advantage estimator (GAE) is the state-of-art estimator for advantage function. It was first proposed by Schulman et al. [10] as a variance reduction scheme for policy gradients. GAE was later adopted by PPO and implemented in SB3. It is the only method used to estimate the advantage function in SB3. GAE is a truncated exponentially-weighted average of $k$-step estimators defined as follows

$$\hat{A}_t = (1-\lambda)\sum_{l=0}^{k-1} \lambda^l A_t^{(1+l)} = \sum_{l=0}^{k-1} (\gamma\lambda)^l \delta_{t+l}^V$$

Observe that the average reward converges at around 300 iterations. In the specific instance below, the value converges to 715.866 after 500 iterations. Moreover, the mean of the standard deviation of the average reward is 28.967.

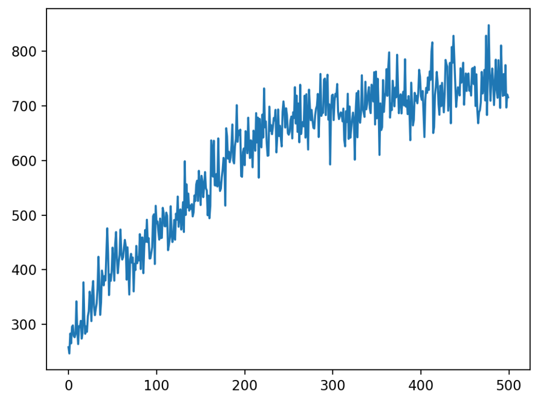

Figure 1: Mean vs. policy iterations

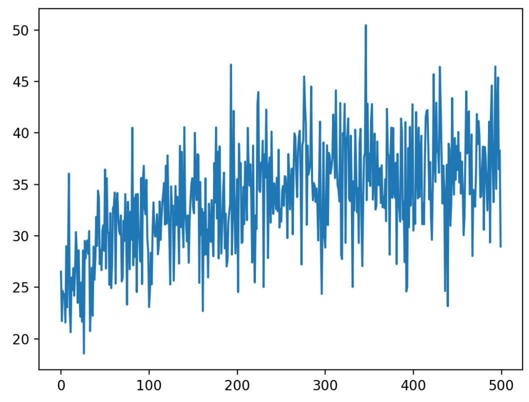

Figure 2: Std vs. policy iterations

## Simple cumulative moving average (SCMA)

We started off our own exploration from the simple cumulative moving average of $k$-step estimators, which is defined as follows

$$\hat{A}_t = \frac{1}{k}\sum_{l=0}^{k-1} A_t^{(1+l)} = \sum_{l=0}^{k-1} \frac{k-l}{k}\gamma^l \delta_{t+l}^V$$

Notice that SCMA does not converge within 300 iterations. At 500 iterations, the average reward attains a value of 573.433. The average reward first increases then decreases. Furthermore, the average reward has a average standard deviation at 35.520, increasing over time. The trend of non-monotonic average reward and increasing standard deviation indicate that the simple cumulative moving average is bad estimator for the advantage function.

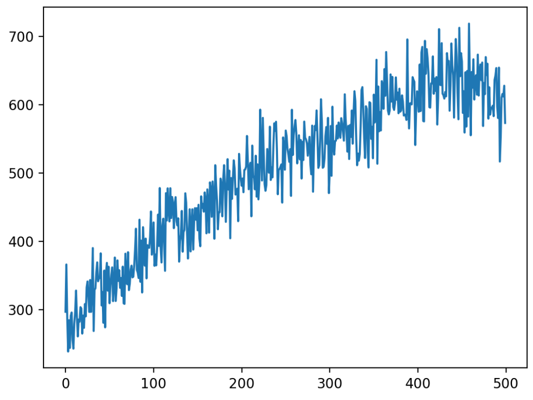

Figure 3: Mean vs. policy iterations

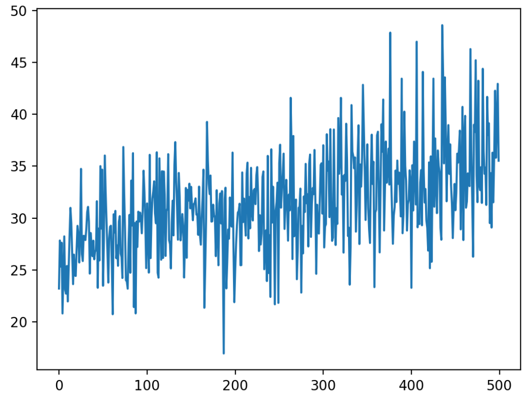

Figure 4: Std vs. policy iterations

## Weighted moving average (WMA)

We then moved on to the weighted moving average of $k$-step estimators, which is defined as follows

$$\hat{A}_t = \frac{2}{k(k+1)}\sum_{l=0}^{k-1}(k-l)A_t^{(1+l)}$$

Compared to GAE, it takes slightly longer for the PPO with WMA to converge. The average reward converges after 400 iterations. The final reward at the 500-th iteration achieves a value of 591.8, which is lower than the average reward yielded by GAE. The standard deviation is 27.416, which is slightly better than the PPO with GAE.

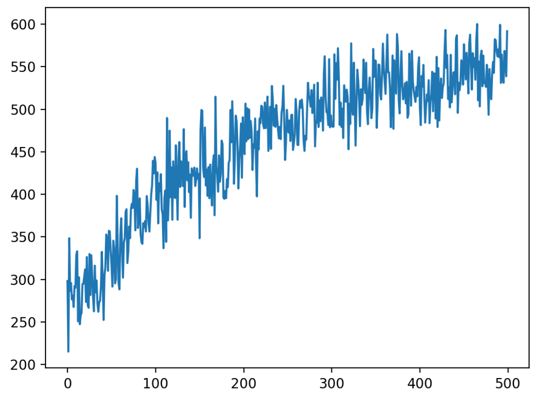
Figure 5: Mean vs. policy iterations

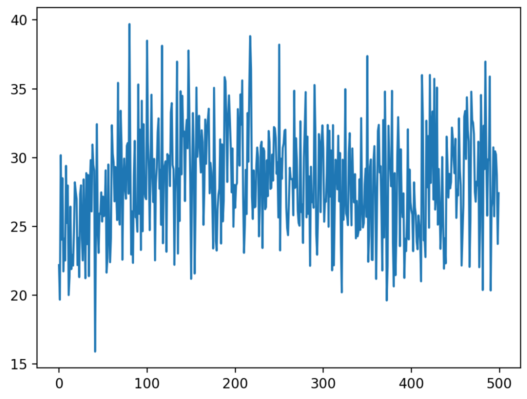
Figure 6: Std vs. policy iterations

## Double exponential smoothing (DES)

Following the spirit of GAE, we also implemented the double exponential moving average of $k$-step estimators, which is defined as below. It was first proposed by Mulloy et al. [8]. DES puts more weights on the recent values, in the hope of removing lag associated with moving average. It has been one of the best cited method to estimate and predict time-series data.

$$\begin{aligned} s_0 &= 0.0 \\ b_0 &= \delta_t^V \\ s_l &= \alpha\delta_{t+l}^V + (1-\alpha)\gamma(s_{l-1} + bl - 1), \quad l = 1, dots, k \\ b_l &= \beta(s_l - s) + (1-\beta) * b, \quad l = 1, \dots, k \\ \hat{A}_t &= s_k \end{aligned}$$

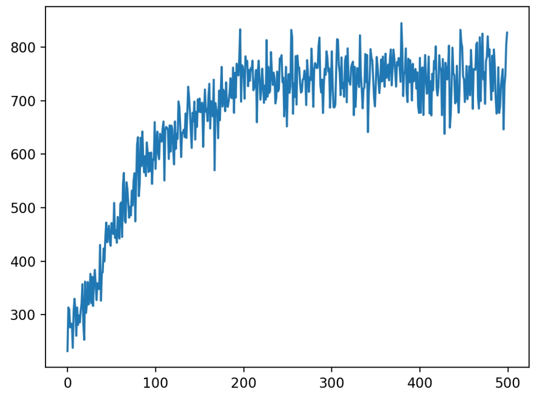
Figure 7: Mean vs. policy iterations

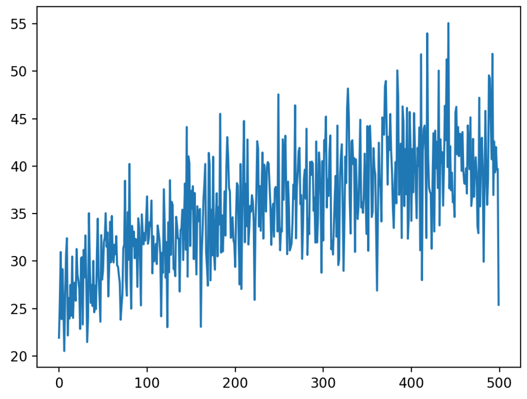
Figure 8: Std vs. policy iterations

Observe that there is a trend of improvement for the average reward with more iterations. The reward converges after about 300 iterations. The final reward value attains 827.533. The standard deviation increases quickly over iterations, and the average standard deviation is 25.386.

Among all the estimators, DES has the highest average reward and lowest standard deviation. However, the standard deviation increases much faster GAE. Thus, we continue using GAE as we explored the effect of other hyperparameters.

## 6 Hyperparameters

Once we had decided to use PPO we modified source code from the `stable-baselines3` library in order to tailor the algorithm to our problem. For all these tests, we used the $L = 3$, $\tau = 20$, and $= 2$ tuple problem instance, having generated and saved a random version of this instance, which all tests were run upon such that our comparisons were minimally affected by randomization. We ran batches of 1000 timesteps of PPO's `learn` function, and for each batch we then evaluated the current policy 10 times, averaging the results and finding their standard deviation.

### 6.1 Learning Rate

We began with learning rate, since we wanted to find a learning rate that converged to a competitive value in a reasonable amount of time (we favored around 500,000 iterations as a maximum – as when gathering data to see how the policy changed over time to help tune the rest of the hyperparameters, this many iterations took around a half hour to run for each example. Later on when actually acquiring our numerical results we were able to use longer runtimes as we had less data to gather during each iteration). The standard learning rate is 0.0003, as shown in Figure 11, so we tested a few larger learning rates, to speed up convergence. We found that by the time we were as large as 0.03 (see Figure 9) we did not learn at all since the rate was too large, so we decided upon 0.003 (see Figure 10) as an optimal compromise between speed and solution competitiveness.

**Mean revenue earned vs. number of PPO learn batches**

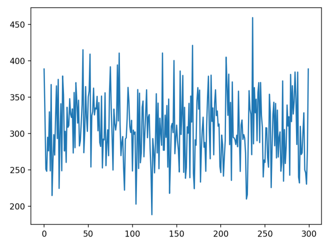

Figure 9: Learning Rate = 0.03

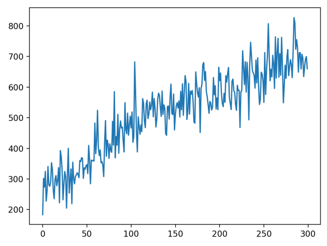

Figure 10: Learning Rate = 0.003

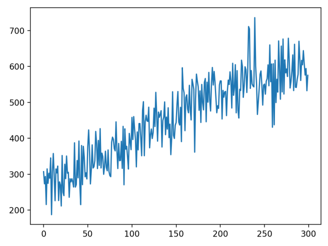

Figure 11: Learning Rate = 0.0003

### 6.2 Rollout Buffer Length

We tested scaling values for the rollout buffer length that reduced minibatches to size 2 and increased them to be the whole number of steps per each update. However, this did not ultimately affect the quality of the results, only the speed at which the algorithm ran, being slower for very small minibatch size, and requiring more total timesteps to reach a similar convergence for very large minibatch size. Thus, we left this value as its default middling value of 64 compared to 2048 steps.

### 6.3 GAE parameter

This hyperparameter weights how much we depend on the previous time steps when estimating the advantage function, and helps with variance reduction. Note that when $\lambda = 1$ we have Monte Carlo and when $\lambda = 0$ we have one-step Bellman. Examining various values of $\lambda$, we clearly saw that as lambda decreased, our standard deviation increased over more iterations. Thus, we resolved to continue with large $\lambda$ moving forwards.

**Std dev. of revenue vs. number of PPO learn batches**

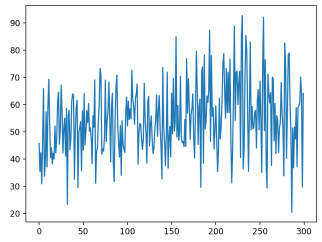
Figure 12: $\lambda = 0.8$

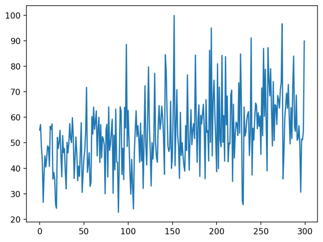
Figure 13: $\lambda = 0.5$

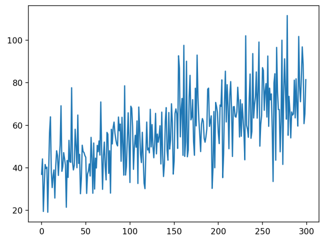
Figure 14: $\lambda = 0.2$

### 6.4 Discount Factor

The discount factor, *gamma*, was important to us to consider since its default of 1 prioritizes looking all the way into the future for the rewards. While this superficially might seem sensible for our problem, since we have fixed halt conditions and only really care about the final reward, we were concerned that it encouraged too much look ahead as the epoch drew to a close. Noticeably, we indeed found that much smaller $\gamma$ (see Figures 19 and 20) values than usually recommended by the literature worked the best – converging sooner and to higher values than $\gamma = 0.95$ (Figure 18).

**Mean revenue earned vs. number of PPO learn batches**

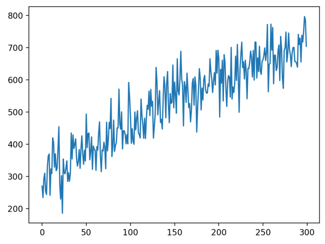
Figure 15: $\gamma = 0.95$

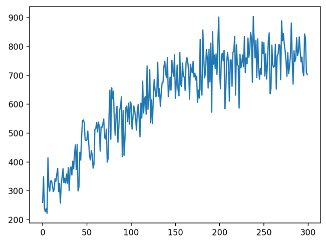
Figure 16: $\gamma = 0.5$

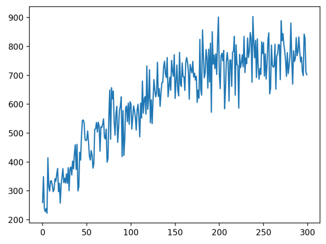
Figure 17: $\gamma = 0.1$

### 6.5 Optimization algorithms

To solve PPO, many authors such as Dai and Gluzman [4] use the `Adam` algorithm within `TensorFlow` or `Pytorch`. Besides `Adam`, we solved our PPO using other first-order methods such as `Adamax, SGD, Adagrad`. `Adamax` is a variant of Adam with infinity norm [7]. `SGD` is a state-of-art algorithm for solving optimization problems. Sutskever1 et al. [13] highlight the importance of `SGD` in the framework of deep learning. `Adagrad` was introduced by Douchi et al. [6]. While inheriting the merits from gradient-based learning, `Adagrad` includes more geometric information from earlier iterations. Among all three algorithms, `Adamax` is the best option with a similar performance as `Adam`. `Adam` is then adopted to be our primary algorithm for the numerical experiments.

**Mean revenue earned vs. number of PPO learn batches**

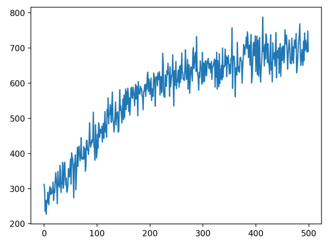
Figure 18: `Adamax`

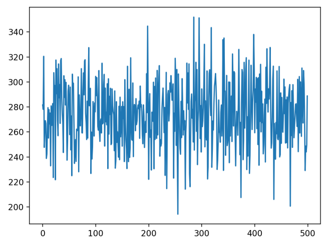
Figure 19: `SGD`

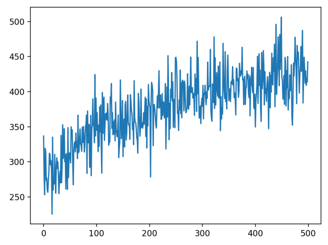
Figure 20: `Adagrad`

**Std dev. of revenue vs. number of PPO learn batches**

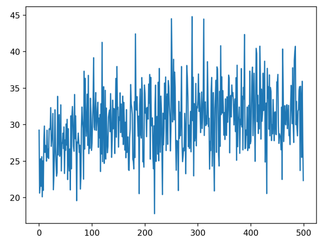
Figure 21: `Adamax`

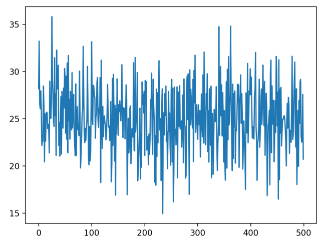
Figure 22: `SGD`

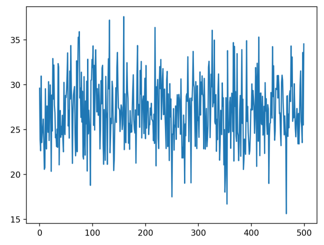
Figure 23: `Adagrad`

## 7 Insights

Besides the input data given in table (6), we also ran a few tests varying epoch length for fixed seat capacity and $L$. As expected, longer epochs shifted the policy towards high-fare seats, and very high epochs (such as 100 when $L = 3$ and $\kappa = 2$) shifted the policy towards only the single-leg high-fare seats. Thus we note that in our Numerical Results section, the near doubling of revenue each time we double both the seat capacity and the epoch length depends on preserving that ratio between epoch length and seat capacities.

With any reasonable number of PPO iterations (10,000 PPO iterations for instance was more than enough), we quickly learn a policy which ensured that for most of the instances we always sell out all the seats. Thus, while we did experiment with adding a "punishment" factor to the reward function if we should reach the end of the epoch and still have seats remaining, we found that such a factor did not improve the policy.

## 8 Numerical Results

We summarize our experiment as shown below: It is worth noting that while most of our

| L | $\tau$ | $\kappa$ | PPO | | # timesteps | # episodes | Adelman DBPC Mean (std. err.) |
|---|---|---|---|---|---|---|---|
| | | | Mean | Std | | | |
| 3 | 20 | 2 | 764.2925 | 6.989 | 800,000 | 500 | 567.78 (20.54) |
| | 50 | 6 | 1790.6675 | 13.815 | 800,000 | 500 | 1,759.91 (33.76) |
| | 100 | 12 | 3744.49875 | 20.998 | 800,000 | 500 | 3,730.04 (53.87) |
| | 200 | 24 | 7551.33625 | 33.774 | 800,000 | 500 | 7,683.04 (70.09) |
| | 500 | 61 | 20884.3325 | 65.567 | 800,000 | 500 | 19,793.50 (132.23) |
| 5 | 20 | 1 | 112.816 | 3.322 | 500,000 | 500 | 486.92 (17.86) |
| | 50 | 4 | 1233.67 | 17.022 | 1,000,000 | 500 | 1,874.34 (36.70) |
| | 100 | 8 | 3237.696 | 31.583 | 1,000,000 | 500 | 3,905.97 (50.49) |
| | 200 | 16 | 7369.488 | 50.794 | 1,000,000 | 500 | 8,109.55 (73.00) |
| | 500 | 42 | 21938.526 | 95.407 | 1,000,000 | 500 | 21,189.10 (125.73) |

results for $L = 3$ are competitive with the results in Adelman, this is not the case for $L = 5$, where we our results did not quite reach convergence (and further, that PPO performed very poorly when the seat capacities were 1). The performance for $L = 3$ suggests that if we ran PPO for longer for all instances, we would approach the LP bounds from Adelman. To demonstrate the degree to which more iterations would likely improve the performance of the $L = 5$ cases, we show in Figure 24 the results of a run with 5,000,000 time steps for $L = 5$, $\tau = 50$ and $\kappa = 4$. In this case, we improve the revenue to have mean 1619.12 (our standard deviation, 74.015, is worse since we consider only 10 episodes of policy evaluation).

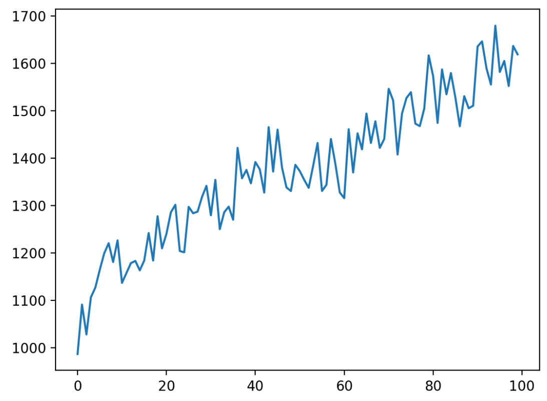

Figure 24: Mean revenue earned vs. number of PPO learn batches

Compared to our partner team who adopted A2C to solve the problem, our result is strictly better. We hypothesized that PPO is a better reinforcement learning algorithm than A2C. The other team's result is attached in Appendix B.

## References

[1] Daniel Adelman. Dynamic bid prices in revenue management. *Operations Research*, 55(4):647–661, 2007.

[2] Dimitris Bertsimas and Sanne de Boer. Simulation-based booking limits for airline revenue management. *Operations Research*, 53(1):90–106, 2005.

[3] Dimitris Bertsimas and Ioana Popescu. Revenue management in a dynamic network environment. *Transportation Science*, 37(3):257–277, 2003.

[4] JG Dai and Mark Gluzman. Queueing network controls via deep reinforcement learning. *arXiv preprint arXiv:2008.01644*, 2020.

[5] D. P. de Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Operations Research*, 51(6):850–865, 2003.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Patrick G Mulloy. Smoothing data with faster moving averages. *Stocks & Commodities*, 12(1):11–19, 1994.

[9] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[10] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[12] Paul J Schweitzer and Abraham Seidmann. Generalized polynomial approximations in markovian decision processes. *Journal of Mathematical Analysis and Applications*, 110(2):568–582, 1985.

[13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[14] Kalyan Talluri and Garrett van Ryzin. An analysis of bid-price controls for network revenue management. *Management Science*, 44(11-part-1):1577–1593, 1998.

## A Randomly Generated Instances

For $L = 3$ we randomly generated

```
self.revenue = np.array([33, 28, 36, 34, 17, 20, 39, 24, 31, 19, \
                         30, 48, 165, 140, 180, 170, 85, 100, 195, \
                         120, 155, 95, 150, 240])
self.probabilities = np.array([0.01327884, 0.02244177, 0.07923761, \
                               0.0297121,  0.02654582, 0.08408091, \
                               0.09591975, 0.00671065, 0.08147508, \
                               0.00977341, 0.02966204, 0.121162,   \
                               0.00442628, 0.00748059, 0.02641254, \
                               0.00990403, 0.00884861, 0.02802697, \
                               0.03197325, 0.00223688, 0.02715836, \
                               0.0032578,  0.00988735, 0.04038733, \
                               0.2])
```

For $L = 5$ we randomly generated

```
self.revenue = np.array([38, 34, 39, 18, 48, 29, 40, 48, 22, 39, 45, 31, \
                         42, 40, 22, 16, 27, 35, 40, 42,  15, 42, 32, 40,\
                         36, 24, 41, 33, 33, 38, 190, 170, 195, 90, 240, \
                         145, 200, 240, 110, 195, 225, 155, 210, 200, 110, \
                         80, 135, 175, 200, 210, 75, 210, 160, 200, 180, \
                         120, 205, 165, 165, 190])

self.probabilities = np.array([0.01302623, 0.00630947, 0.0193087 , 0.03749824, \
                               0.0087251 , 0.02197966, 0.0230311 , 0.0250458 , \
                               0.02696926, 0.03631881,  0.00848936, 0.0169562, \
                               0.01757013, 0.01980117, 0.03372276, 0.00092609, \
                               0.01588487, 0.01056883, 0.02438527, 0.00747704, \
                               0.00655709, 0.01516504, 0.01366724, 0.02056504, \
                               0.03065696, 0.02719751, 0.03476736, 0.03692992, \
                               0.00394042, 0.03655934, 0.00434208, 0.00210316, \
                               0.00643623, 0.01249941, 0.00290837, 0.00732655, \
                               0.00767703, 0.0083486 , 0.00898975, 0.01210627, \
                               0.00282979, 0.00565207, 0.00585671, 0.00660039, \
                               0.01124092, 0.0003087 , 0.00529496, 0.00352294, \
                               0.00812842, 0.00249235, 0.0021857 , 0.00505501, \
                               0.00455575, 0.00685501, 0.01021899, 0.00906584, \
                               0.01158912, 0.01230997, 0.00131347, 0.01218645, \
                               0.2])
```

## B Comparison with partner team

| $L$ | $\tau$ | $c$ | Base MLP | Shared | GRU | DBPC |
|---|---|---|---|---|---|---|
| 3 | 20 | 2 | 364.29(150.89) | 317.01(139.27) | 188.17(108.42) | 567.78(20.54) |
| | 50 | 6 | 845.68(247.119) | 706.30(235.96) | 737.47(228.16) | 1759.91(33.76) |
| | 100 | 12 | 1546.01(341.09) | 1556.57(342.26) | 1543.84(336.87) | 3730.04(53.87) |
| | 200 | 24 | 2913.53(488.92) | 2901.09(487.35) | 3107.65(521.16) | 7683.04(70.09) |
| | 500 | 61 | 8189.19(850.59) | 7257.78(754.45) | 8089.46(832.38) | 19793.50(132.23) |
| 5 | 20 | 1 | 116.18(72.08) | 106.38(72.87) | 93.72(69.54) | 486.92(17.86) |
| | 50 | 4 | 585.96(252.78) | 608.05 (243.24) | 654.85(250.75) | 1874.34(36.70) |
| | 100 | 8 | 1489.12(457.99) | 1641.90(449.97) | 1650.27(459.54) | 3905.97(50.49) |
| | 200 | 16 | 3660.86(792.38) | 3942.27(775.49) | 3625.08(742.68) | 8109.55(73.00) |
| | 500 | 42 | 8698.46(1266.06) | 8292.50(1140.22) | 10985.99(1419.44) | 21189.10(125.73) |