REINFORCEMENT LEARNING PROJECT

Bike Rebalancing

Submitted in Fulfilment of Requirements
for the Degree of Master in Data Science
of the Northeastern University by

Ativeer Patni

Khoury College of Computer Sciences
August 25, 2021

Supervisor: **Prof. Nik Bear Brown, and**
Prof. Kylie Bemis

Table Of Contents:

Abstract

Bike sharing systems, aiming at providing the missing links in public transportation systems, are becoming popular in urban cities. A key to success for a bike sharing system is the effectiveness of rebalancing operations, that is, the effort-s of restoring the number of bikes in each station to its target value by routing vehicles through pick-up and drop-off operations. There are two major issues for this bike rebalancing problem: the determination of station inventory target level and the large-scale multiple capacitated vehicle routing optimization with outlier stations.

The key challenges include demand prediction accuracy for inventory target level determination, and an effective optimizer for vehicle routing with hundreds of stations. After researching possible machine learning techniques for such redistribution, I ultimately decided to take the approach of using Reinforcement Learning (RL), as these methods have had great success in solving strategic problems. Finally, the extensive experimental results on the NYC Citi Bike system show the advantages of the approach for bike demand prediction and large-scale bike rebalancing optimization.

The source code of this project can be found here:
https://github.com/Ativeer/Bike-Rebalancing---Citi-Bike

1. Introduction

Over the past half-decade, bike-share has become an increasingly viable option for transportation and recreation in urban areas. It offers a potentially pleasant and often convenient mode of moving about a city; bike-share also helps mitigate concerns of getting one's bike stolen, and provides the option of one-way trips. One major drawback of the largest American bike-share programs as of 2018 is an uneven flow from station-to-station, leaving some stations with an overflow of bikes, and others lacking bikes altogether. These issues inconvenience users, and have lead to a number of efforts to create ***'dockless'*** bike-share programs, which solve this problem by allowing users to park their bikes anywhere, but create a number of externalities of their own [1] , such as piles of bikes on sidewalks, and users hiding bikes for future use

1.1 Background

The company which oversees bike-share programs in New York (Citibike), Washington D.C. (Capital Bikeshare), Portland, Oregon (Downtown PDX), the Bay Area (Ford GoBike), among others, currently maintains a close eye on the issue of evenly allocating bikes among their pick-up/drop-off stations, and has a fleet of vans which drive around to redistribute the bikes; the company has also begun experimenting with incentivizing users to move bikes from full stations to empty stations with a points-based reward system in its NYC-based BikeAngels [2] program, attesting to the importance the company sees in resolving the issue.

At the present moment, the bulk of bike redistribution is done manually by a team of over 100 employees, monitoring stations in the company's headquarters and operating the vans to redistribute the bikes on the ground [3]. Regular patterns emerge in which stations fill up at which times, on which days of the week, during certain types of weather, etc. The team working bike redistribution is no doubt familiar with many of the nuances of these systems, but a problem such as this lends itself readily to the prediction techniques of machine learning, and we strongly believe that a system designed to predict and optimize the pattern of bike redistribution in New York City (and eventually trained in other cities) could greatly accelerate the successful redistribution of bikes within the network, and alleviate the inconvenience to users of winding up at a full or empty station, ultimately increasing ridership and satisfaction.

1.2 Objective

The objective of this project is to design and implement a self-Bike Rebalancing model to minimize the total cost, including the travel costs, unbalanced penalties, and incentive costs.After researching possible machine learning techniques for such redistribution, I ultimately decided to take the approach of using *Reinforcement Learning (RL)*, as these methods have had great success in solving strategic problems, and I believe that Citibike redistribution can be framed in the context of a strategic game without significant information loss. In our analysis, we examined and compared the success of a number of Reinforcement Learning methods, including Q-tables, deep Q-networks, and Q-learning with forecasting models.

1.3 Scope

The scope of this project is to optimize availability of bike stands and not overstock the bikes at one stand. The code is written in a Python environment and the model is trained against a certain period of months (from April 2019 to July 2019 and same for 2020). Also, user behaviour and incentives ideas are out of the scope of this project.

2. Literature Review

2.1 Reinforcement Learning

Reinforcement learning is one of the learning approaches for neural networks. Reinforcement learning implements differently compared to supervised learning and unsupervised learning. Reinforcement learning makes an agent to generate the optimal policy in an environment and maximises the reward. An agent can learn the rewards given by taking action in each state and subsequently learn the proper action for each state without having any predefined rules and knowledge about the environment. The agent learns by trial and error, or in other words, the actions with higher reward are reinforcement while actions with penalties are avoided in the future. It is different with supervised learning because supervised learning requires a huge number of labelled dataset to train the model. The training process is repeated with the same set of data until the model converges. It requires efforts in labelling the data and the process is error-prone. Reinforcement learning is also different from unsupervised learning because reinforcement learning aims to maximise the reward while unsupervised learning does not.

2.2 Q-Learning

[4][5][6] Q-learning is a reinforcement learning mechanism that compares the expected utility of available actions given a state. Q-learning can train a model to find the optimal policy in any finite MDP. The Bellman equation suggests the Q-value function as shown in equation below:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

The difference of the Q-value function in equation 7 with the value function in equation 3 is that the Q-value function estimates the maximum future reward for taking action a given a state *S*. The calculation of the Q-value function takes into account the maximum discounted future reward for an agent to move from state *S* to state *S'*. The Q-value function can be iteratively converged to the optimal Q-value function by the difference of the estimated utilities between current state S_t, and next state S_{t+1}, with learning rate α, at time step unit *t*, as shown below:

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha\left(r_{t+1} + \gamma \max_{a} Q_t(s_{t+1},a) - Q_t(s_t,a_t)\right)$$

Then, Q-learning function can substitute the value function, as shown below

$$V^*(s) = \max_a Q^*(s, a)$$

The optimal policy can be retrieved from the optimal value function with

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

2.3 Deep Q-Learning Networks (DQN)

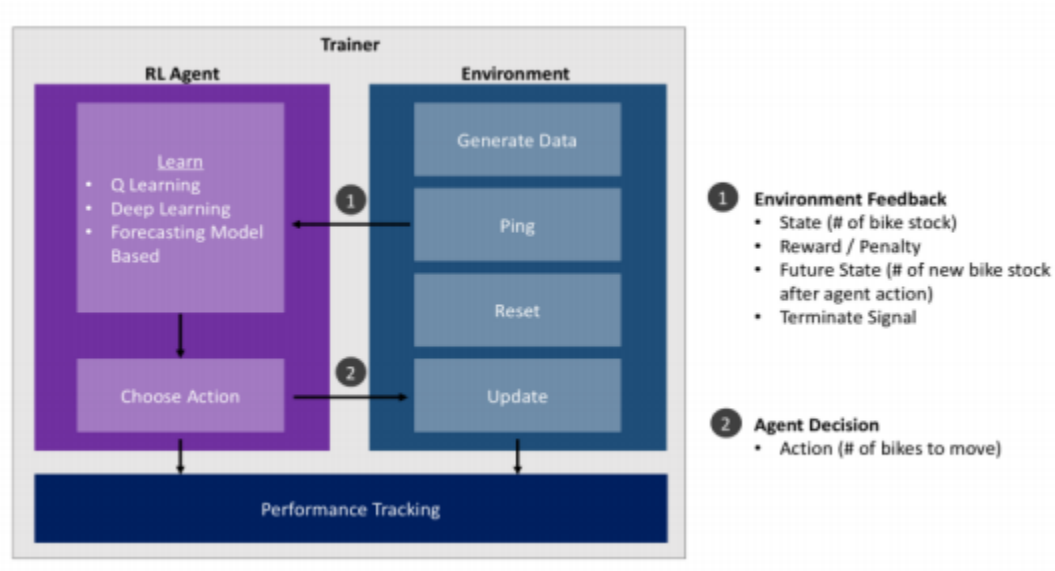
[7] Neural network is utilized as the function approximator of the Q-learning algorithm to form the Deep Q-learning network (DQN). The learning goal of DQN is to find the best settings of the parameter θof $Q(S, a; \theta)$ or the weights *w* of the function approximator. The objective of DQN is to minimise the Mean Square Error (MSE) of the Q-values. Besides that, experience replay technique uses first-in-first-out(FIFO) method to keep experience tuples in replay memory for every time step. The replay memory stores experience tuples for several episodes to ensure that the memory holds diversified experiences for different states. The experience tuples are sampled from replay memory randomly during Q-learning updates. The implementation of experience replay can remove correlations in the observation sequence and smoothing over changes in the data distribution by randomizing over the data. In practice, the most recent N experience records are sampled uniformly and randomly from replay memory. The random sampling gives equal probability to all experience tuples.

3. Data Collection

To train the RL agent, the program relies on two types of data: simulated and actual Citi Bike data. Simulated data can be 1) a simple bike stock with fixed increment of 3 additional bikes per hour and 2) an increasing bike stock of 3 additional bikes per hour with random fluctuation. The random fluctuation is produced by a random integer generator, which has a mean of zero and range between -5 and 5. The program also generates bike stock data based on real historical Citi Bike ridership. Using the Citi Bike Open Data site, an API call is made to pull a month of trip data from July, 2020. A sequence of data processing steps are performed to translate trip data to hourly bike stock by station (~600 stations) over the month based on the net flow of arrival and departure trips. Because the initial bike stock is unknown, the team made an assumption that all stations start with 20 bikes on July 1st, 2020. This is a parameter users can change dynamically as they see fit. Weather data is collected manually from the New York meteorological website.
CitiBike provides monthly related information here.

4. Solution Design

The solution includes four modules: RL agent, Environment, Performance Tracking, and Training. To better develop a robust solution with flexibility for future expansion, the code was written using an Object Oriented Programming (OOP) framework.



The Environment module generates simulated and actual Citibike data, sends feedback to the RL agent (e.g. current bike stock, reward and penalty, new bike stock, and termination signal), updates the bike stock based on the action received from the RL agent, and resets the environment for new training session. The RL Agent module mainly consists of learning and decision capabilities. One of the objectives of this project is to benchmark performances of different learning implementations, such as Q Learning, Deep Q Learning, and Model Based Learning (QLearning with Forecasting). Users can specify which method to use for learning or simply select all. The Performance Tracking module captures all the detailed metrics and visualizes and stores the results automatically. Users can find the results in their local directory for analysis and future reference. The key metrics the program tracks are action history, reward history, bike stock comparison between first and last training episodes, and success ratio of all training sessions. The Trainer module facilitates the end-to-end process of training set up, RL and environment module set up based on user inputs, and exception handling.

5. Methodology

The solution includes three types of learning methods: Q-Learning, Enhanced QLearning with Forecasting, and Deep Q-Learning using Neural Network. Each of which was discussed in detail in the literature section.

5.1 Reinforcement Learning

In the context of the Citi Bike Rebalancing problem, a state is the number of bike stock at a station. Action is the number of bikes the agent can move at each hour. The agent can choose to move 0, 1, 3, or 10 bikes at a given hour. Reward and penalty is structured as the following:

- -30 if bike stock falls outside the range [0, 50] at each hour
- -0.5 times the number of bikes removed at each hour
- +20 if bike stock in [0, 50] at 23 hours; else -20.

This reward structure encourages the agent to keep the bike stock within an acceptable range while moving as few bikes as possible.

5.2 Q Learning with Forecasting

One potential shortcoming from the Q-Learning model is that the agent chooses an action (the number of bikes to move) based only on the current bike stock. In theory, the agent should also consider the expected bike stock in the next hour. During rush hour, the number of bikes at a station might change dramatically from hour to hour. For these high-traffic periods, we thought it might be useful for the agent to consider the expected bike stock one hour in advance before making a decision.

For example, consider a scenario where we have a bike station in Midtown with 30 bikes and 20 open slots at 8:00am. Because many people are commuting to work, we expect that 40 rides will end at the station in the next hour, so the expected balance at 9:00am is 70. A hypothetical Q-Table for the particular station is below. What is the optimal action at this time?

	Action Space				
Bike Stock	-20	-10	0	10	20
30	-0.6	-0.2	1	0.4	-0.3
40	0	0.1	1	-0.2	-0.8
50	0.2	0.4	0.3	-0.5	-0.9
60	0.8	0.2	-0.6	-0.7	-1
70	0.7	-0.1	-0.7	-1	-1

If we consider only the current stock (30), the Q-table tells us that the optimal action is to do nothing. (Given row=30, the action that maximizes expected reward in one hour is "0"). But because we expect the station's bike stock to increase from 30 to 70 in the next hour, selecting an action of "0" is probably not a good strategy. Instead, we experimented with an approach that picks the best action based on an average of the current stock and expected stock in one hour. In this case, the average of the current stock (30) and expected stock (70) is 50. We then take the best action specified by bike stock=50, which is to remove 10 bikes. So at 8:00am, we would reduce the number of bikes in the Midtown station from 30 to 20 to account for the expected increase.

To compute "expected bike stock" in the next hour, we built a Random Forests model. For more details about the Random Forests model, please refer to the appendix at the end of the paper.

5.3 Deep Q Network

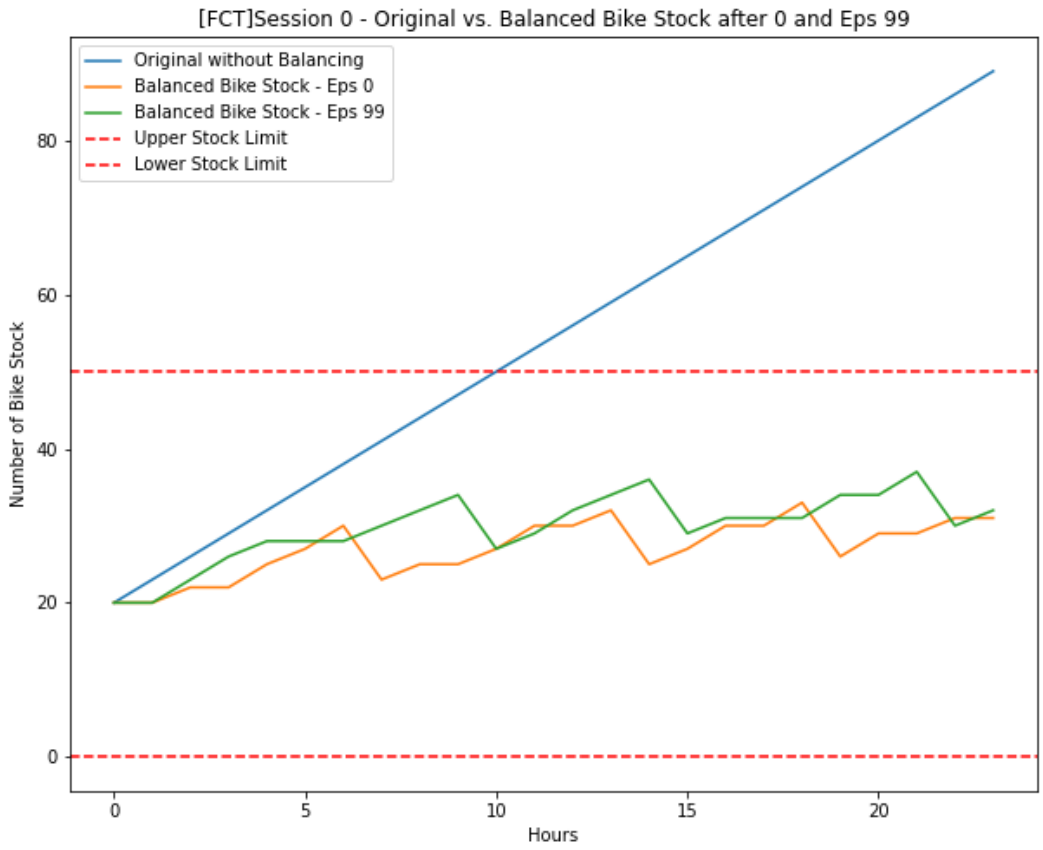
There are many times when one wants to perform reinforcement learning that a q-table will be too inefficient. A typical example of this situation is using a reinforcement learning algorithm to teach a program to play videogames. In the seminal paper by Mnih et al Human-level control through deep reinforcement learning they discuss their work teaching a program to play different Atari video games. The possible states were represented by the number of pixels in the image screen and they used 4 such screens to detect the direction and speed of objects in the game as well. Taking into consideration that the screen used a 128 color palette and the size was 84 x 84 pixels the number of possible states would equal 128^{28224}. That means that are q-table would have to have the same number of rows to represent each state. Lookups in this table would be slow and highly demanding of memory so the alternative presented was a neural network.

A neural network is very efficient in learning features from structured data. Though the reason why neural networks work so well is not fully understood, we know they have a high ability to generalize and deal with unexpected data. In order to calculate scores that we would look up in a q-table we make one forward pass through the network and

more quickly get our results. In the previous case where we mentioned the screens from a video game as the input has no missing data which allows the model to learn from data well. In the case of citibike the data is also well organized in the sense that we know the amount of bikes at a station at any particular time which makes the problem well suited for a DQN. In our model of Citibike redistribution, each state is represented by the hour of the day and number of bikes at a station. In a simple case where we are looking at 1 station that could host a maximum of 80 bikes, there are 80 x 23 states. After taking the state and possible actions as output, the model will return an action that gives the highest q-value for the possible state

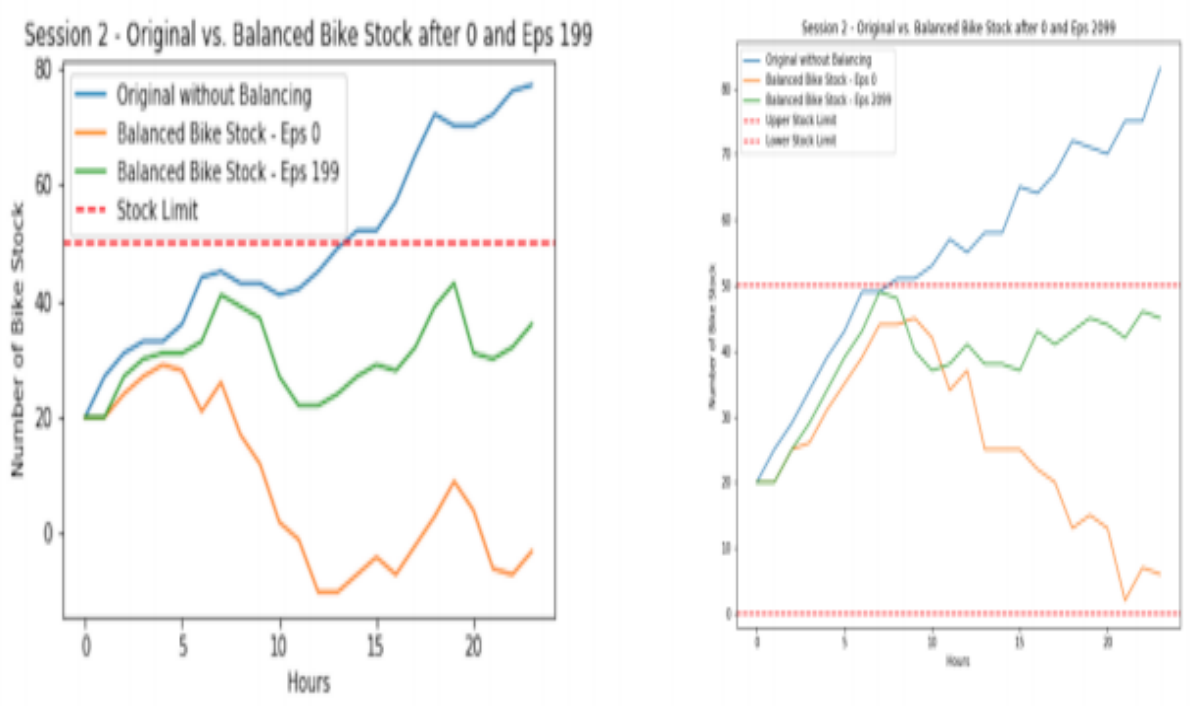
6. Results

A Simple Case with Linearly Increasing Bike Stock The team tested the RL agent with a simple case. Based on the graph below, the RL agent proved to be able to 1) recognize the bike stock limit of 50 without explicit coding and 2) learn a more cost-effective way to move bikes. The orange line represents how the RL agent moved bikes in the first interaction with the environment. It simply moved a random number of bikes at each time step. The green line represents how the RL agent moved bikes after interacting with the environment and learning after 213 rounds. It developed a smarter strategy: the agent waited until the bike stock was about to reach 50, which was the artificial constraint, before taking some actions. The agent learned to meet the objective with a cost-effective strategy that it developed by trial-and-error without having human instructions.



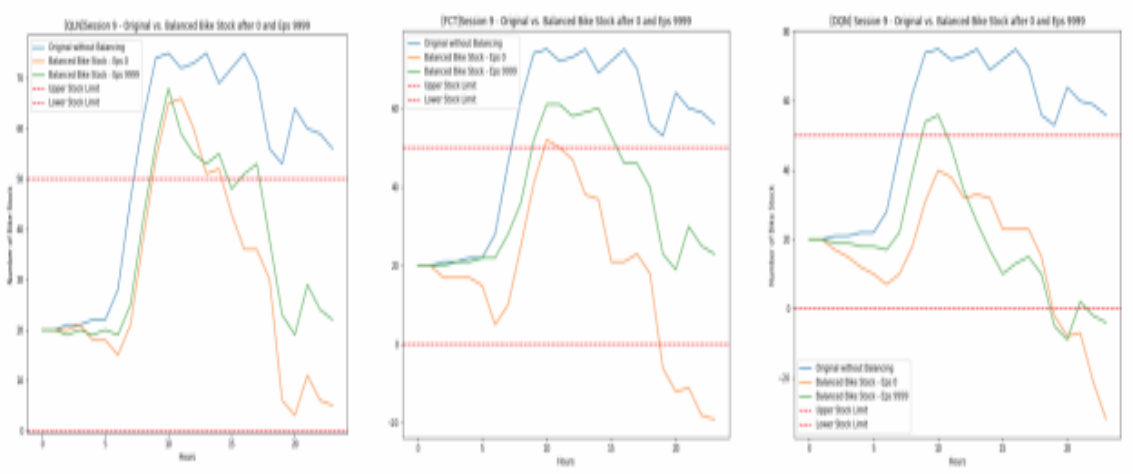
A more Realistic Simulation with Random Variation and New Constraints

Once the basic mechanics of the RL agent was proved, random dynamics was introduced to the bike stock. In addition, new constraints of non-zero bike stock were also reinforced using heavy penalties. Without changing the code or having additional human instruction, the RL agent was able to adapt and develop a better bike rebalancing strategy and recognize the non-zero boundary.

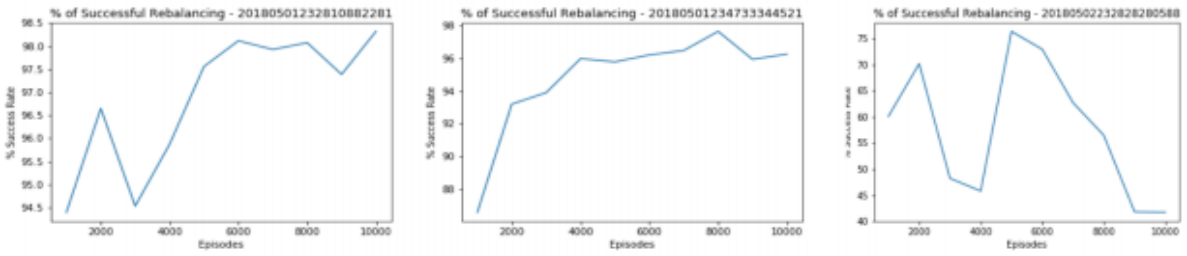


A Comparison between Different Learning Methods with Real CitiBike data

To test the real-world effectiveness of the three RL models (Q-Learning, Q-Learning with forecasting, and Deep Q Networks), we analyzed how they performed for a Citi Bike station. More specifically, we tried to balance the bike stock for the Citi Bike station at 17th and Broadway (Union Square) over a 24-hour period on July 1, 2020. Union Square is peculiar because there are two convenient north-south bike lanes (Broadway and Lafayette Street) that feed into the plaza, but no good outbound bike lanes. As a result, the station on 17th and Broadway tends to accumulate bikes over the course of the day, as more bikes are arriving than departing. The first set of plots illustrate the ability of the models to balance bikes at the 17th and Broadway station. From left to right, we have the regular Q-Learning model, the QLearning plus forecasting model, and the Deep Q Network method. Each plot contains: 1) the station's actual bike stock (blue); 2) rebalanced stock by a "dumb" untrained agent (orange); and 3) rebalanced stock by a trained agent (green). We assume that the bike stock starts at 20, and that the station has 50 total spaces for bikes.

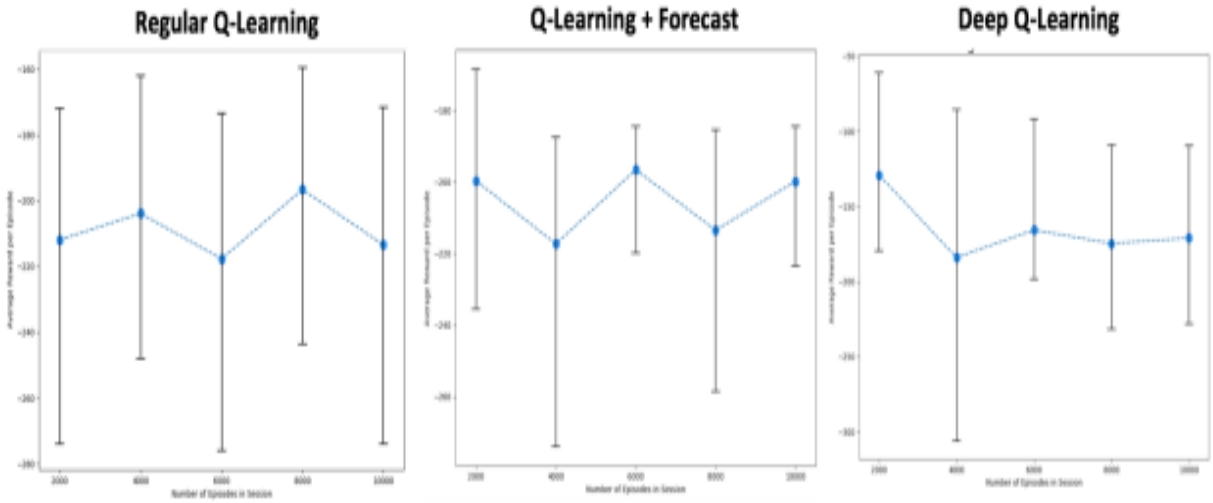


We want the agent to keep the bike stock in the range [0, 50] while minimizing the number of bikes moved. If the agent does nothing (blue line), it will suffer a huge penalty, since the bike stock remains above 50 for almost 18 hours. Conversely, an untrained agent (orange line) may pursue inefficient transfers of bicycles, sometimes even causing the stock to fall below 0. At least in the first two plots, the trained agent seems to do the best job of balancing the priorities of: i) keeping bike stock within [0, 50] and ii) moving as few bikes as possible. Our main evaluation metric for “success” was the percentage of episodes (days) where the end-of-day balance (hour 23) fell in the range [0, 50]. For each model, I trained sessions with 2000, 4000, 6000, 8000, and 10000 episodes. The results are plotted below.



After sufficient training, the two Q-Learning models were able to balance the bike stock by hour 23 over 95% of the time. However, the Deep Q Network method was not as proficient according to this evaluation metric, averaging a success rate of around 60%. We suspect that with additional parameter tuning, the performance of the DQN method would have improved.

Another evaluation metric we analyzed was “Average Reward Per Episode”. As a refresher, the agent incurs a cost of -30 for each hour it fails to keep the bike stock within [0, 50], as well as -0.5 for each bike moved. The results for 2000, 4000, 6000, 8000, and 10000 episodes are illustrated below. The y-axes are on different scales, but the most important finding is that the Deep Q-Learning method outperforms the Qlearning models by average reward. Specifically, the average reward for the QLearning methods is approximately -210. By contrast, the DQN model has an average reward of roughly -170



The gap between the DQN and Q-Learning models is about 40 points. Since each “bad” hour -- with bike stock outside [0, 50] -- results in a -30 point penalty, one could say that the DQN model, on average, keeps the station balanced for 40/30 = 1.33 hours longer than the Q-Learning models.

7. Discussion and Possible Next Steps

In this project, I focused on taking historical data from a particular station in NYC and balancing the bikes to be between 0 and 50. While the methods were generally effective for this approach, the dynamics will most likely change when we observe multiple bike stations. When bikes are removed from one station, they generally must be added to another. This means that the action space will also have to increase to represent the addition of bikes. Specifically, we would probably need to include one action space between every possible pairing of stations. Future complications occur when we decide to not only remove bikes from one station but then divide those bikes amongst multiple stations. This situation is more reflective of real world experiences when Citibike picks and retrieves their bikes for optimization. These additional requirements for our model will also mean there will be an increase in the state space. Previously, our state space was the number of possible bikes at a station * 24 hours. (In our Citi Bike station example, the bike stock peaked at roughly 80, so this would equate to 80*24 = 1920 possibilities. However, even adding one more station with the same state space could increase the total state space to over 3,000,000. Due to the size it would no longer be optimal to use a q-table and we would transition into using the DQN. Since we are using neural networks we could optimize the speed by running on a GPU.

8. Conclusions

The RL agent was able to develop, adopt, and improve bike balancing strategy autonomously in various bike stock dynamics - from simple linearly increasing, randomly generated, to actual bike stock - and new bike stock limits. In addition, we were able to benchmark the performances using different learning methods, such as Q Learning, Q Learning using Forecasting, and Deep Q Learning. Determining the best method is not trivial. By one evaluation metric (% of episodes where stock is rebalanced by hour 23), the Q-Learning methods are more successful. By another metric (average reward per episode), the DQN method achieves better performance. Ultimately, from CitiBike's perspective, the best choice of model depends on their priorities. Does CitiBike only care about balancing bikes during a 24-hour window? Then DQN is the clear winner. What if CitiBike wants to ensure that each bike station is properly balanced for the next day? Then the Q-Learning methods may warrant additional consideration.

Appendix:

Random Forests Forecasting Model

To derive a station's *"expected bike stock"* in the next hour, we used a Random Forest algorithm. Specifically, we built Random Forests models to predict the hourly net flow of bicycles, with one model per station. "Net flow" equals the number of bikes arriving minus the number of bikes departing a station. We simply added the net flow predictions to a station's current balance to predict the expected balance in one hour. Our first goal was to predict a station's hourly net flow in July 2020. To train the model, we collected Citi Bike trip data from April-July 2019 (to capture seasonal effects) and March-June 2020 (to capture more recent temporal trends), with July 2020 as our test set. Overall, for each station, we had 5112 training records (24 hours * 213 days). The variables for the model were a mix of autoregressive features and weather data scraped from Weather Underground's API. The main autoregressive features we engineered were:

- **Average Net Flow for Hour in Past Week:** average net flow for that hour over the seven previous days.
- **Average Net Flow for Hour-Weekday Combination:** average net flow for the hour/weekday combination over the three previous weeks.
- **Net Flow, hour t-1:** net flow at station in previous hour
- **Net Flow, hour t-2**
- **Cumulative Net Flow, past 12 hours:** total net flow at station over previous 12 hours

I created a similar set of autoregressive features for the number of departing bikes:

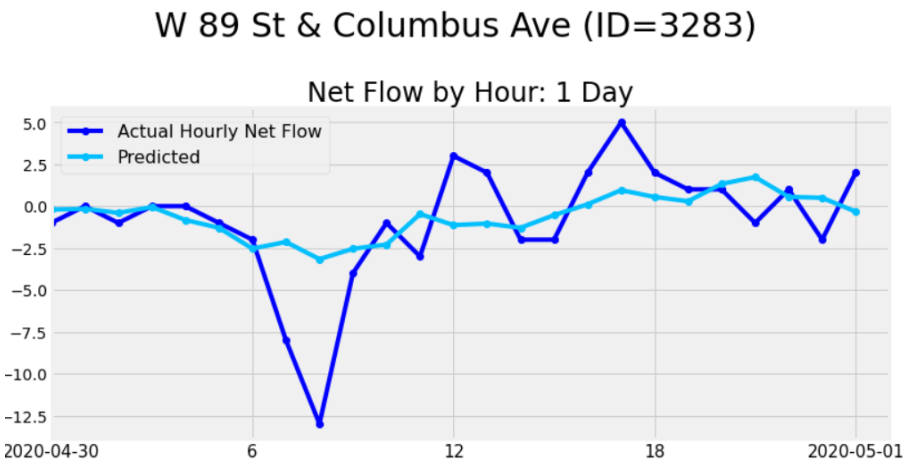
- Average Departures for Hour in Past Week
- Average Departures for Hour-Weekday Combination
- Departures, hour t-1
- Departures, hour t-2
- Cumulative Departures, past 12 hours
- Cumulative Departures, past 24 hours

I included the "departures" features because we felt that they provided information that the "net flow" variables cannot provide. For example, a "net flow" of 0 is ambiguous, as it could either mean: 1) the station has no activity or 2) the station is busy, but the departures and arrivals are balancing each other out. To avoid overfitting, we set the parameter of "minimum samples per leaf" equal to 5.

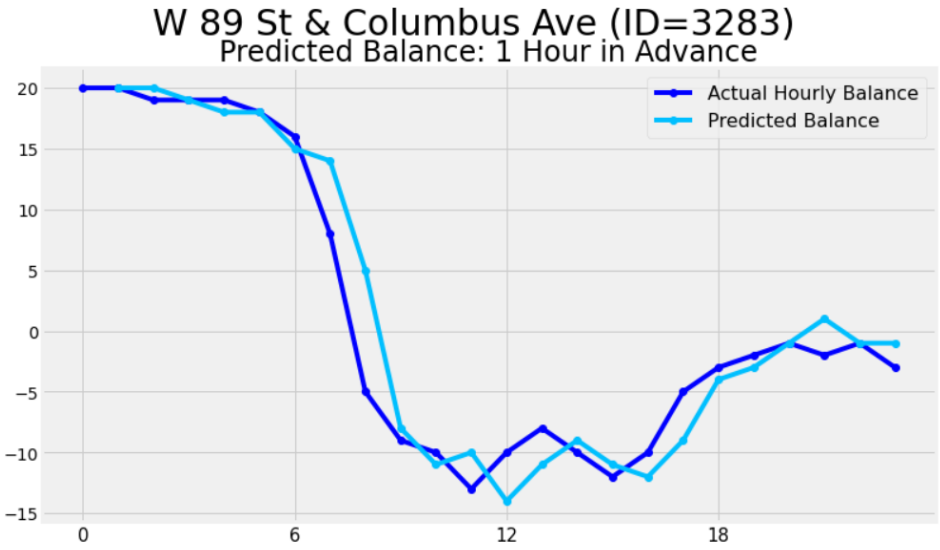
The effectiveness of the Random Forests model varied greatly from station to station. For some stations, the R2 value on the test set (July 2020) was extremely high, at over 80%. For other stations, however, net flow was far less predictable (see table below). One confounding variable in the analysis was that we lacked information on when Citi Bike moved bicycles. For some really busy stations, Citi Bike probably has a set schedule in place to move bikes. As such, it was easier to forecast net flows at busy stations. By contrast, stations with low bile flow (e.g. 3 Ave & 72 St) were tougher to predict. (Note: test set R2 calculated over all 31 days in July).

	StationID	Name	Test R^2	Train R^2
0	519	Pershing Square North	0.317	0.746
1	392	Jay St & Tech Pl	-0.02	0.632
2	426	West St & Chambers St	0.069	0.82
3	497	E 17 St & Broadway	0.352	0.802
4	3164	Columbus Ave & W 72 St	0.139	0.744
5	281	Grand Army Plaza & Central Park S	0.227	0.914
6	3443	W 52 St & 6 Ave	0.665	0.929
7	304	Broadway & Battery Pl	0.195	0.908
8	3375	3 Ave & E 72 St	-0.044	0.599
9	3283	W 89 St & Columbus Ave	0.188	0.668

The plot below illustrates the model's hour-by-hour predictions for net flows at Station 497 (W 89 St & Columbus Avenue) on July 1, 2020. This is the same station I used to test our RL models. Note that the prediction (light blue) follows a conventional trend, with peaks in the morning and evening, while the actual observations are slightly noisier.



The next plot illustrates the predicted bike stock at the same station. To generate this plot, we simply added the “net flow” predictions from the Random Forests model for each hour to the current balance.



I was concerned with the model’s accuracy rather than interpretability, but for the sake of thoroughness, a list of feature importances for the RF model is below. The autoregressive variables are quite significant. Note that the weather features have low Gini coefficients. It is likely that the autoregressive variables “deps t-1” and “nets t-1” are capturing much of the information provided by weather data.

	Feature Name	Gini Coeff.
0	hour_average_nets_past_week	0.149681
1	nets t-1	0.0987923
2	hour	0.0820759
3	hour_average_nets_weekday_past3	0.0620629
4	deps t-1	0.0512636
5	nets_12h	0.0512397
6	nets_24h	0.0490157
7	hour_average_deps_past_week	0.0465969
8	pres	0.0456881
9	hour_average_deps_weekday_past3	0.0447549
10	deps_24h	0.0405681
11	deps_12h	0.0334676
12	dwpt	0.0333159
13	deps t-2	0.0323903

References

1. https://www.washingtonpost.com/local/you-can-park-a-dockless-bike-share-bicycle-anywhere-but-you-shouldnt/2018/02/19/adbcd996-1585-11e8-8b08-027a6ccb38eb_story.html
2. https://help.citibikenyc.com/hc/en-us/articles/115000246291-What-is-Bike-Angel
3. https://help.citibikenyc.com/hc/en-us/articles/115007197887-Redistribution
4. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M.: 'Playing atari with deep reinforcement learning', arXiv preprint arXiv:1312.5602, 2013
5. Sutton, R.S.: 'Learning to predict by the methods of temporal differences', Machine learning, 1988, 3, (1), pp. 9-44
6. Boyan, J.A., and Moore, A.W.: 'Generalization in reinforcement learning: Safely approximating the value function', in Editor (Ed.)^(Eds.): 'Book Generalization in reinforcement learning: Safely approximating the value function' (1995, edn.), pp. 369- 376
7. https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf