CPS2 Master: UJM · M1 Internship: EMSE

Machine Learning supported Multi-agent Resource Allocation in Autonomous Vehicle Fleets

Prepared By:
Mohamad Kassem

mohammad.kassem@etu.emse.fr
mohamad.kassem@etu.univ-st-etienne.fr

Supervised By:
Alaa DAOUD - Flavien BALBO

flavien.balbo@emse.fr
alaa.daoud@emse.fr

LIMOS UMR CNRS 6158, Institut Henri Fayol, Mines Saint-Étienne, F-42023 Saint-Étienne, France.

1 September 2021

Contents

Abstract

In this work we focus on building a machine learning model that can predict the number of taxi demands in each zone in New York city at different time slots of the day. For the purpose of the training of this model we extract data from the NYC dataset which contains real taxi trips records from different zones in the city. Having such sequential data allows our model to train through RNN, and in particular LSTM layers. The trained model predictions can be added to the agents belief base in an existing multi-agent simulation of different request allocation mechanism to test how the agent's behaviour in the system could change after learning new information from the prediction model. And if this could lead to a better allocation quality in the system. Which could result an improvement in the profit of the taxi companies and for a better passengers satisfaction.

.

Keywords:

Dial-A-Raid Problem(DARP), On-demand transport(ODT), Multi-Agent System(MAS), Autonomous agent vehicles, Taxi demand prediction, Machine Learning Model, Recurrent neural networks(RNN), Long-short term memory(LSTM).

1 Introduction

The Dial-a-Ride Problem (DARP) is a variant of the Pickup and Delivery Problem (PDP). It consists of designing vehicle routes and schedules for n users who specify pickup and delivery requests between origins and destinations to be satisfied by a set of m vehicles. Where the aim is to plan routes for the n users in a certain way that maximizes the profit of the company through minimizing the cost of the trips and at the same time keeping the clients satisfied. This must be done under a set of constraints such as taking into consideration the capacity of the vehicles, respecting client's time window...

DARP and in particular, On-Demand Transport (ODT) systems which focuses on the idea of driving passengers from their origin to their destination, have attracted increasing attention in recent years. The area of application was almost limited to road systems. The ODT system has never been considered for the replacement of public transport services but to extend it.

As mentioned in [1] we already have real-life examples of services and systems that implement different ODT solutions, such as the services provided by Uber, Lyft, and other on-demand transport platforms. Moreover, because the traditional centralized dispatching method to solve the problem has NP-Hard complexity, then a better solution for the problem is through a decentralized multi-agent system (MAS) where each vehicle is modeled as an autonomous agent that can build its own schedules, negotiate and coordinate with other agents to optimize the overall solution.

Thus, we are interested in deploying a fleet of autonomous taxi vehicles in a decentralized multi-agent system capable of communicating in a peer-to-peer manner, in order to satisfy all clients' requests in the best possible way. And here raises 2 main questions that should be taken into consideration:

1. do we know, even partially, the distribution of requests in space and time?
2. How do agent's beliefs affect the quality of their decisions?

To tackle these topics, we are going to build a machine learning (ML) model that can predict the number of taxi demands in a certain zone in the city during a certain time of the day.
But before building this model, we are going to study how it should be used in the an existing multi-agent simulation. And what could change in the system if we add such model into the agent's belief base and they started to behave differently.
After this, we will walk step by step through the process of building and training of our model.

2 State Of The Art

As many could guess, there is a lot of previous work in the domain of ODT in general. But it turns out that there are only very few previous related works in the domain of taxi demand prediction improvement using machine learning.

One of the interesting existing work is a sequence learning model proposed by [3]. In their approach they focus on the application of recurrent neural networks "RNN" (a special type of neural network designed for sequence problems) for learning a model that predicts taxi demand based on the requests in the past. A model that can learn time series data is necessary here since taxi requests in the future are related to the requests in the past. And to let the model learn time series data they used LSTM Layer (a powerful type of RNN). For this model, 3.5 years of GPS records from NYC Taxi trip dataset were used. These GPS Data needs to be encoded first. And after encoding, the entire city was divided into small areas. [3] claims that the experimental results show that the achieved prediction model outperforms the existing prediction models based on fully connected feed-forward neural networks and naive statistic average. Because the model was capable to continuously make real-time prediction of taxi demand for the entire city with a high level of accuracy.
Another approach proposed by [4] is through learning complex spatiotemporal (space - time) patterns to predict future taxi demands. For this purpose they proposes a temporal guided networks (TGNet), which is an efficient model architecture with fully convolutional networks and temporal guided embedding, to capture such patterns. To train their model they used taxi trip records in 2015 from 01/01 to 01/03 from NYC dataset and these 60 days were splitted into 40 days as a training data and 20 days for testing. After training the built model, it was compared to other existing models such as CNN with LSTM and ConvLSTM which predicts similar information and [4] claims that most of the time they had more accurate results.
In [5] they are interested in predicting a demand hot-spot (an area crowded in passengers) considering the degree of influence of past events. Through machine learning techniques, several queries generation and expansion practices they examine the prediction performance of a classifier that determines whether an upcoming event is a hot-spot or not. They also propose a framework for taxi demand prediction during non-recurrent events. They combined information extracted from the Web with time-series data to build a predictive model of taxi demand hot-spots around special event venue areas. After finalizing and testing the model [5] claims that their accurate results show that information retrieval using query expansion methods outperforms other baseline methods which only rely on the basic attributes of an event. But we need to take into consideration that in real world applications, a methodology that relies on

search engines has its shortcomings since there is limited access to their full search index and some times they doesn't contain enough details for a certain event.
Some other approaches are based on data analysis, for instance [6] propose a passenger hot-spots recommendation system. By analyzing the existing taxi trips, they extract hot-spots in each time step and assign a hotness score for each one. This score will be predicted in each time step and combined with the driver's location, the top k hot spots will be recommended for the drivers.

Most of the presented work shows interesting approaches and mechanisms to achieve the final purpose of the model. While, from what we have seen, non of them focuses on the agents in the system. However, the main difference between our work and the previous existing work is that our goal is to focus on the effect of the machine learning model on the agent's belief base. So, in our case the model is a tool for the agents that can make them learn a new information so that they could get some use of what they have learned in order to behave in a different way in the system and thus improves the overall solution of the problem.

3 MAS Coordination in ODT

In this section we are going to present different allocation mechanisms as an ODT solutions. Moreover, we are going to focus on the decentralized, Auction-based Multi-agent system approach. Then we will present an existing simulation in the domain, check how the agents behave in this simulation and give a smaller sample example of how such running simulation would looks like.

3.1 Different Allocation Mechanisms

There exist several allocation mechanism approaches to solve DARP. Some of these approaches are based on centralized dispatcher systems, like the classical centralized Mixed-Integer Linear Programming (MILP) based solvers, which focuses on the idea of building dynamical schedules by a central dispatcher that uses CPLEX solver to solve the allocation problems as MILP which maximize the objective function profit. And some other approaches are based on decentralized systems such as the Greedy approach, which is based on planning a single request in advance. And it has been mentioned by [7] as the best strategy for dynamic settings, following genetic algorithm selection.

3.1.1 Auctions Approach

The results of our work, which will be presented in the next sections, can be merged as a helpful tool in many systems, including the decentralized greedy system. However, we are mainly interested in merging the prediction results of our machine learning model with the decentralized, Auction-based Multi-agent system proposed by [1]. This new solution proposes a decentralized coordination mechanism for the exchange of requests, based on an insertion heuristic and auctions, to allocate requests to vehicles in the context of dynamic ODT with vehicle to vehicle (V2V) communication.

When we talk about auctions, there are two different types. [1] uses "First-Price Auction" to answer the question "Which vehicle will consume the request and at what cost?". In this type of auctions, several advertisers sets a fixed bid per impression, and the highest bidder wins directly. Unlike the other type of auctions "Ascending-Bid auction" (which is more popular) where the price and allocation are determined in an open competition among the bidders. The bidder is willing to pay the most win and pay a price that no other bidders are willing to top.

3.1.2 Auction Winner Determination

In the type of auctions we are dealing with in [1], when a certain demand is sent by a client, each vehicle agent will offer only one Bid for this demand, stating: the demand number, the vehicle name (his name or ID in the system), the starting time of the potential pick-up and the cost of satisfying this demand. When all the agents sets their bids for a certain demand, the auction starts, and the aim is to have only one agent as a winner who can satisfy the client's request with the minimum cost. Thus one of the bidders will win the auction through certain winning determination criteria, which is based on the idea of considering all vehicles as truthful collaborating agents. Each vehicle determines its own value v of winning an auction of a certain demand d though win(v,d) (the default value is zero). The bid is available until a predefined expiry time t. Until this time if a vehicle received a better offer than the one it had, it assigns itself as a looser in this auction (sets win(v,d) to 0). At the expiry time t if the cost of v is less than any other cost. Thus, it will have a value win(v,d) equal to 1. And it considers itself as the winner of this Auction and adds the demand d to its queue of demands.

In [1] there are also several examples of all the scenarios that could happen in the proposed system. Which shows some issues and proposes more practical solutions for these issues.

3.2 Simulation

Several ODT solutions including the greedy approach and the proposed Auction-based solution by [1] are realized as experiments within the "Plateforme Territoire". Which includes a huge running simulation composed of different Java classes for each approach and each kind of agents. For instance we have a certain set of classes that simulates a greedy system and an associated greedy agent class. We can also find another set of classes which simulates an auction-based decentralized multi-agent system which contains a request generation system that represents the client requests in the system. These requests are being satisfied by the agents through simulating auctions for these demands and determining winners through the winning determination criteria we previously discussed.

3.2.1 New Cost Equation

This existing simulation makes it easy for us to merge our machine learning model with the different approaches and benefit from the prediction results of this model in improving the existing allocation mechanisms (auctions in particular), through defining a new way of determination of the cost needed by each vehicle to satisfy a certain demand.

The existing cost equation which is built in the agent's belief and used when they want to bid in a certain auction, depends on several important attributes including the empty driving (without a client) distance that the vehicle needs to travel before arriving to the pick-up location of the client. However, beside the existing parameters in the system, we can also worry about some other important parameters that could be taken into consideration to improve our cost equation. For instance, the distance that an agent needs to travel after dropping-off a certain client and before picking-up the next one. Which is also considered an empty driving distance which need to be minimized.

Here comes the role of our model which is capable of predicting the number of potential client demands (in the near future) in the destination zones of the current available client demands. This information can be used to reduce the empty driving distance through creating a new cost equation which takes the result of our model as a new attribute.
In our new cost equation we need to keep on the previous attributes of the equation. Beside this we are going to add a negative α value to the equation:

$$Cost = OldCost - \alpha \tag{1}$$

- Old Cost: attribute represents the old equation before the model.
- α: is an attribute which represents the effect of the model in the equation. it is proportional to the predicted crowdedness of the destination zone of a certain available client.

In order to get the best use of our new cost equation, and merge it with the existing simulation. We need to understand the agent's behavior in the system.

3.2.2 Agent's Behavior

At the beginning of the execution of the simulation, a fleet of n vehicle agents is distributed randomly. The behavior of each vehicle consists of the 4 states: picking up, dropping off, going to and marauding.

- **Picking Up** Is the state where a vehicle makes a decision to satisfy the request of a certain client.
- **Dropping Off** Is the state where a vehicle is arriving to the destination of a certain client and his request is satisfied. So that, now it is ready to receive new demands.
- **Going to** Is the state of a vehicle when it has a specific destination, i.e. a request to

serve. The vehicle is either going to pickup location if the request is not yet picked up, or going to delivery location otherwise.

- **Marauding** Is the state of a vehicle when it does not have a request to serve, i.e., at the beginning of the simulation, each vehicle marauds until it decides to serve a request. Once a passenger is dropped off, the vehicle reverts to marauding. In this state, the vehicle randomly moves through its neighborhood to find requests to serve.

3.2.3 Example

Consider a decentralized, auction-based allocation multi-agent system with a set of 3 agent vehicles V1,V2,V3, a set of 3 travelers T1,T2,T3, a set of potential travelers PT1,...,PT4 and set of 2 Zones Z1,Z2. As shown in figure 1.
An agent vehicle can see a traveler's request in the system only when he can satisfy it in the requested time window of the client. For instance, V1 can see all the 3 requests of T1,T2 and T3. While V2 can see only the request of T2 but not T1 or T3 because they are far away. And V3 can't see any of the 3 requests because they are all far away. So V3 is marauding, while V1 and V2 need to take decisions. Now let's compare the expected behavior of the agent's in the system in 2 different cases: before the addition of the model, and after the addition of the model in the agent's belief base.

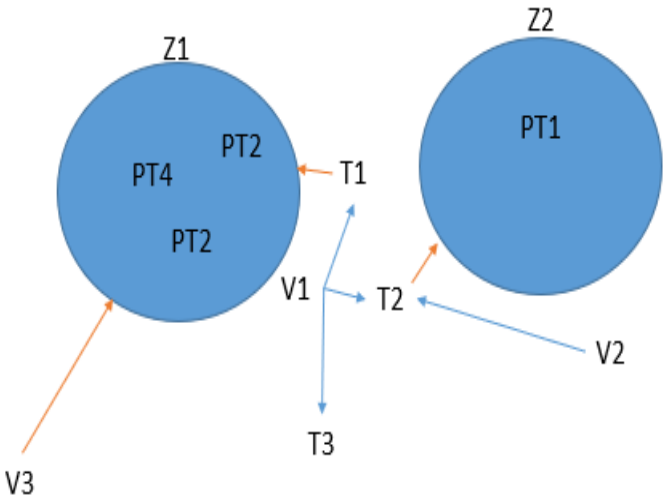


Figure 1: A Sample Running Simulation

Before adding the model:
Before the addition of the model, the agents will coordinate and react in an auction-based system:

- V1 will win the auctions of picking up travelers T1 and T3 because he is the only bidding agent.
- For the demand by traveler T2, both V1 and V2 will bid in this auction, but because cost c(V1,T2) is smaller than cost c(V2,T2), then V1 will win this auction too.
- Now V1 will have to choose among the 3 travelers, and because the cost of picking up T2 (which now depends mainly on the empty driving distance d(V1,T2) before the pick-up of T2) is the smallest among the 3 choices, then V1 will prefer to pick up T2
- V3 will keep marauding in his neighborhood.

After adding the model:
Now let's see how the agent's behavior would change after the addition of the model in the system:

- The same auctions will happen again with V1 winning all of them.
- But this time V1 will prefer to choose T1 and not T2, because now the destination zone of each of the travelers' matters, and the cost c(V1,T1) now is smaller than cost c(V1,T2) because the model predicted that Z1 (the destination zone of T1) have more potential travelers than Z2 (the destination zone of T2) and this was reflected in a change in the cost.
- V3 will stop marauding in his neighborhood and instead will go to zone Z1 (and wait for potential request) because the model predicted that Z1 is the zone with the highest amount of potential requests.

Note that we could also prepare the system to solve deeper cases for a better profit, for instance, this time V1 chose T1 instead of T2, then now maybe V2 can run another auction and win it because now he is the only bidder. And now we have 2 satisfied clients T1 and T2 in a short period of time.
But even if we ignore this case and focus only on the first 2 cases of V1: "Changing the cost value according to potential travelers in the destination zones of the current available demands", and of V3: "Marauding avoidance through going to zones with more potential travelers". These 2 cases are enough to observe the change in the agent's behavior which could improve the existing

resource allocation mechanisms leading to a better overall maximal profit.

In the next section, we are going to discuss our machine learning model to be added to the agent's belief to support our multi-agent system resource allocation process in the autonomous fleet of agent vehicles.

4 Machine Learning Model

In this section, we present our machine learning model and the dataset and environment used to train this model. We also discuss the different stages our model passed through: Data preparation stage, architecture building, training and testing. Moreover, we discuss the faced difficulties such as resources limitation. And at the end we show and interpret the obtained results.

4.1 Model Definition

The aim of this work is to build a machine learning model to support and improve the multi-agent resource allocation in autonomous vehicle fleets.

This model will predict future information that can help the agents in the system to behave in more a intelligent way, in order to reach their goals. Obviously we can't make predictions for unlimited time into the future. So in terms of time we need to take into consideration that our prediction will be for a certain t minutes horizon into the future. For instance, in [8] they were able to predict future taxi trips for the next 30 minutes in time. Similarly, for the location we need to choose a certain area to work in. For the purpose of this work, we have chosen New York City, for being famous for the wide number of taxi demands and the importance of taxi system in the city. Moreover, we are going to divide the city into a set of zones. After clarifying these points, now we can introduce the main hypothesis of our model:

*"Given the number of passengers **n** in a certain zone **z** during a time slot **t**, can we predict the number of passengers **n'** in the same zone **z** at next time slot **t+1**"*

4.2 Dataset

For the purpose of building and training of this model. Like any other model we need a dataset. In our case, we need a dataset that contains taxi trips details in New York City. So, just like [3] and [4] we have chosen NYC dataset. But in their case (before the year 2016), the dataset was a collection of GPS data (pickup and drop-off latitude & longitude) which is not very practical to use in the current working domain, so they needed to encode this data into geographical zones in the city, which is easier to deal with. But in our case we have an advantage because the data in the dataset after 2016 is directly given as geographical zones. So we can choose any year from 2017 to 2021 to take data from. Furthermore, to avoid the particular case of

COVID-19 that could affect the taxi demand in the recent years we decided to take data from 2018. This data is available on the NYC dataset website and can be extracted as CSV files where each file corresponds to certain month of the year.

The dataset contains records about different kinds of taxis: Yellow taxi and Green taxi. The difference between the two kinds is mainly in the area of activity. Where each one of them operates in a set of different zones than the other. So, we decided to extract records from both of them (12 months of csv files from each kind).

Regardless of the kind of the taxi, each one of the 24 files has a big size of rows, and each row is composed of several attributes that defines a taxi trips. Such as, the pick-up/drop-off time and zone of the trip, the payment method, the payed amount by the client, the covered distance...
Although among all these attributes, we are interested in two main attributes:

- Pick-Up Zone
- Pick-Up Time

These 2 attributes are the main essentials for our data preparation stage. But before discussing this stage, first we need to define a suitable environment to work on, during the stages of our model.

4.3 Environment

For the purpose of building our ML model we need to choose a programming language to work with, and because of all its built in libraries and packages, Python is considered as the best choice in the domain, where it makes it easy for us to perform data analysis and data preparation through libraries like Numpy, Pandas... Also to build, train and test our model using the help of libraries like Scikit-learn, Tensorflow and Keras.

To get the best use of Python, we usually need to prepare a suitable environment for the project, download the needed libraries and use a certain IDE. But since late 2017, Google decided to make this environment preparation process easier by coming up with Google Colab (or Co-laboratory) which is a free laboratory from Google Research. It is a Web IDE used to execute python code through the browser in a predefined environment (Zero Configuration required). This saves us lots of time and work that it would require if we decided to go for the traditional environment preparation process.

Colab is easy to use and makes it easy to share files because it stores user files in the cloud (Google Drive). And it is mainly used to train machine learning and deep learning models because it allows us to train our models using GPU or TPU (Tensor Processing Unit, suitable for Tensorflow projects). Because large datasets such as the one we have, produce much faster results using GPU rather than CPU. Thus taking all of these advantages into consideration we decided to use Google Colab for this project.

4.4 Data Preparation

Now that we have a ready environment to use. In this stage we are going to discuss how we prepared our data in order to reach the targeted clean and organized data that will be used as an input for the model.

4.4.1 Reading Data

After downloading the yellow and green 24 CSV files of the year 2018 from the NYC dataset. In order to work with these files in our environment, first we need to upload then to Google Drive, which is considered as the default storage of Google Colab. Thus, we uploaded the 24 files into the drive. Then we mount (connect) Colab to the Drive through a certain Python script.

Now that we have our data ready in the drive, and we have our IDE (Colab) connected to the drive. It is time to read our csv files and save them into a "Python Data Frame" (Like a Database table) After that, we use Pandas Library to extract only the two columns (Pick-up time and Pick-up zone) that we need from the data frame we have.

4.4.2 Garbage Data

Although, after extracting our data, it turns out that there exists some "Garbage Data" in the dataset. Garbage data, is a term used in the domain of data analysis to refer to some irrelevant rows of data that could appear accidentally in the dataset, due to some mistakes that could happen during the data collection that have been done by the dataset creators. And this data have to be cleaned before proceeding with the other stages of the model.
In our case we are working on data from year 2018 only. But it turns out that we have some records that corresponds to some other years. We observed this from the pick-up and drop-off time attributes. Where in some cases one of the two attributes or both of them happens in year 2014 for instance. Or any year other than 2018. Thus, such rows are considered as garbage data. And we can get rid of them using different ways. But because this garbage data is small,

they we decided to delete it from the dataset using queries (similar to SQL queries) on the data with the help of Pandas library.

4.4.3 Data Sampling

Because we have a huge amount of data more than we need, then we can make data sampling. Which is a process used to choose randomly a certain amount or percentage of data from the dataset we have. So we have extracted samples of close sizes from all the 24 files. Then we put back all the samples together into one data frame which contains the two columns (Pick-up time and zone) and a large number of rows.

Now that our data extraction and cleaning is ready and we have all the data we need to proceed in one big data frame. It is time to discuss the input of our model.

4.4.4 Model Attributes

Referring back to the hypothesis we presented in the "Data Preparation Section". We need to define three main attributes for the training of our model: the zones, time slots and number of passengers.

Each trip record has a certain pick-up zone in the second column of our final dataset. The city was divided into 263 different zones by the dataset creators. So the possible values of this attributes ranges from 1 to 263, where each number represents a certain zone in the city.

Starting from an initial time t0 being the beginning of the year 2018, we define **m** time slots (intervals of time) till the end of 2018. We kept the size of the time slot as a variable, so that we can try different values. We tried different values that lasts from few minutes into several hours. The most convenient range of value for our model turned our to be time slots of size between 10 minutes and 30 minutes. Because, the number of time slots is inversely proportional to their size. Then we can't take very small time slots such as 1 minute time slots, because if we do that then our data will be very huge to be trained and could take several days even using the best computers. We can't define very big time slots (12 hours for instance) either, because they will not be helpful for our model. As we know, we are trying to predict the number of trips in the next time slot so that the agent can benefit from this information to know how to react in the system to take near future decisions. Then if the next time slot is after 12 hours, it will not be useful for our agents.
Therefore, let's say we choose to define time slots of size 30 minutes. Then because our time

slots are consecutive, we can easily determine their number through dividing the number of minutes of the whole year (360x24x60 = 525600) by the size of each time slot (30), to get the number of time slots m equal to 17520 time slot per the whole year. The beginning of the first one is at 00:00:00-01/01/2018 and it ends at 00:30:00-01/01/2018. The beginning of the last one is at 23:30:00-31/12/2018 and it ends just before 00:00:00-01/01/2019.

Moreover, we need to generate our third and last attribute (the number of passengers), using the "pick-up time" data which is the first column in our final dataset. For this generation we also need the help of Pandas library which allows us to perform queries (Like SQL queries) on our python dataframe which contains the dataset.
Our purpose is to generate the number of passengers n during a certain time slot, in a certain zone. So, n depends on both time slot and zone number. Then the final shape of each observation would look like: {ti, zj, nij}. To achieve this, we have to perform a certain query for k iterations. Where k is the number of time slots multiplied by the number of zones. In each iteration, a certain zone is fixed and we perform our query on the data frame to count the number of records that exists between each two consecutive time slots ti and t(i+1). This counted number represents our targeted nij. We keep repeating this process until we iterate through all the zones, fixing each one of them at a time.

After defining the three main attributes for our model, now it is time to present the input and the output of the model.

4.4.5 Model Input & Output

We need the model input (X) and the model output (Y) for the training of our model.

The input X is an array of arrays, where each sub-array is of the form {ti, zj, nij} for i in {0,1...,m-1} and j in {0,1...,263}.

The output Y is an array of the form {nij} for i in {1...,m} and j in {0,1...,263}

For the input, i is in {0,1...,m-1}. While for the output, i is in {1...,m}. Because for each time slot ti in the input, the model need to learn how to predict the number of trips for time slot t(i+1) in the output. Thus, we need to have this difference of one time slot between the input and the output. But at the same time we need to keep them of the same size. And for this reason we decided to ignore the last time slot (i=m) in X and the first time slot (i=0) in Y.

As a result of this data preparation stage, now we have X and Y ready to be used in the model training stage. But first we need to discuss the architecture of our model.

4.5 Model Architecture

In this part, we are going to describe the shape of our model. And in order to build build it we first need to define the following terms: RNN, LSTM, Tensorflow and Keras.

4.5.1 RNN

Recurrent Neural Networks (RNN) are a special type of neural network designed for sequence data problems. Simplest example of RNN model can be:
Input: Sequence of integers {4,5,6,7,8}.
Output: Next Digit in the sequence {9}.
Given a standard feed-forward, multi-layer perceptron network, an RNN can be thought of as the addition of loops to the architecture. For example, in a given layer, each neuron may pass its signal sideways in addition to forward to the next layer. The output of the network may feedback as an input to the network with the next input vector. And so on... This can be illustrated in the following figure:

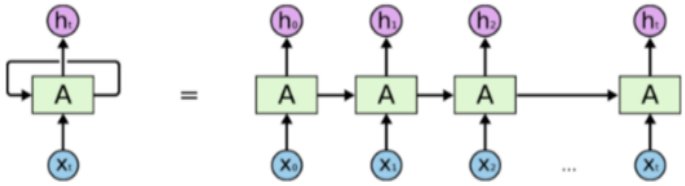


Figure 2: An unrolled RNN

RNNs has a vanishing gradient problem which occurs when the weight update value is almost zero (the update in the weight is negligible). This is a very bad situation because we will be running our machine (training) but our model will not get any better.

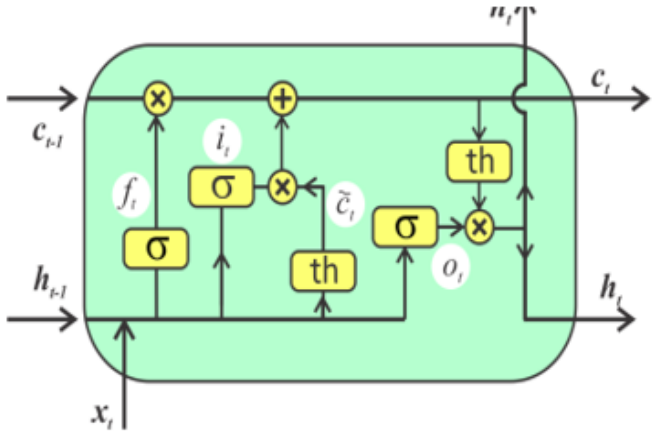
4.5.2 LSTM

A powerful type of RNN called the Long Short-Term Memory (LSTM) Network has been shown to be particularly effective when stacked into a deep configuration, achieving good results on several sequence problems such as, language translation and video classification.

LSTM network is an RNN that is trained using Back propagation through time and overcomes the vanishing gradient problem. It can be used to create large (stacked) RNNs, that can be used to address difficult sequence problems in machine learning and still achieve good results. Instead of neurons, LSTM networks have memory blocks that are connected into layers.

A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A unit operates upon an input sequence and each gate within a unit uses the sigmoid activation function to control whether they are triggered or not, making the change of state and addition of information flowing through the unit conditional. There are four types of gates within a memory unit:

- Forget Gate: conditionally decides what information to discard from the unit.
- Input Gate: conditionally decides which values from the input to update the memory state.
- Cell Update: conditionally decides what information to keep.
- Output Gate: conditionally decides what to output based on input and the memory of the unit.



$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_t\right)$$
$$\mathbf{i}_t = \sigma\left(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i\right)$$
$$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c\right)$$
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o\right)$$

Figure 3: LSTM Memory Block Architecture

4.5.3 Tensorflow

TensorFlow is an open-source Python library for fast numerical computing created and released by Google. It is a foundation library that can be used to create machine learning and deep learning models directly or by using wrapper libraries that simplify the process built on top of TensorFlow.

Unlike other numerical libraries intended for use in deep learning like Theano, TensorFlow was designed for use both in research and development and in production systems, it can run on single CPU systems, GPUs as well as mobile devices and large-scale distributed systems of hundreds of machines. But for best performance, using Tensorflow with GPU is suggested.

4.5.4 Keras

Keras is a powerful and easy-to-use free open source python library created by Google to make developing and evaluating machine learning and deep learning models as fast and easy as possible. It is a high-level interface and it is built on top of Tensorflow (it uses Tensorflow in the back-end). Just like Tensorflow, Keras can operate on different machines but GPU is suggested for best performance. Keras allows us to define and train neural network models in just few lines of code. And it is widely used for both research and development purposes.

4.5.5 Model Design

We can easily build our RNN - LSTM sequence model in python, using Tensorflow and Keras. Our model is composed of three stacked LSTM layers. Between each two of them, we add a dropout layer which is used to drop a small percentage of the data to avoid "Overfitting". Overfitting is a concept in data science, occurs when a certain model fits exactly against its training data. i.e, when the model memorizes the data instead of learning from it. In this case the model is unable to generalize well to new data.

After building our model, now it is time to train it using X and Y.

4.6 Model Training

In order to train our designed model on X and Y, we are going to split our dataset into "Train Dataset" and "Test Dataset". Then we will use Keras which allows us to compile and fit the model using K-Fold cross validation technique.

Train - Test Split is the process of dividing our dataset into two subsets. The first subset is used to fit our model and is referred to as the "Train Dataset". The second subset is used instead to evaluate our model, and it is referred to as "Test Dataset". The objective of this process is to estimate the performance of the model on new data that it haven't seen before (it helps to avoid overfitting).

K-Fold Cross Validation is a re-sampling procedure also used to evaluate the model skills on unseen data. This procedure has a single parameter **k** that refers to the number of groups that a given data sample is to be split into. This procedure uses the train-test split process in k iterations. At each iteration one of the samples will act as test dataset and all the other samples together will act as train dataset. Then we keep iterating until each data sample have acted as a test data exactly once.

We can compile our model using Keras, which provides us with the opportunity to use a pre-defined optimizer and loss function. Also, to adapt a certain leaning rate for the training.

- "Loss Function" is the function that computes the difference between the current output of the algorithm and the expected output. For our model we are using "Mean Square Error" (MSE) loss function.
- "Optimizer" is an algorithm used to change the attributes of the neural network such as weights and learning rate to reduce the losses. In our model we are using Adam optimizer.
- Adapting a "Learning Rate" for our Adam optimizer could increase the performance and reduce the training time. After trying different values for the learning rate, we have chosen 0.01 as the best one for our model.

After the compilation of our model, now we need to fit it and this involves several hyper parameters that can be taken into consideration:

- The number of epochs which defines the number of times that the learning algorithm will work through the entire training dataset. The choice of this parameter depends on how fast a certain model can converge to a final accuracy. In our case we tried different values between 5 and 20 epochs. But most of the time the model was able to converge into a fixed accuracy using only 5 epochs.
- The batch size which is a parameter of the gradient descent that controls the number of training samples to work through before the model's internal parameters are updated. For example, 16 batches means that after every 16 data inputs one forward/backward

propagation will occur. In our case we tried different values for the batch size. Among the tested values, both 8 and 16 were good choices.

At this point we had everything ready to perform our first run to train our model. But when we tried to do this we failed several times because of certain limitations in the working environment. In the following subsection, we are going to discuss these limitations and how we were able to overcome them and succeed on running the training process of our model.

4.7 Resources Limitations

It is well known that working with big machine learning/deep learning models takes a lot of processing time (up to several days in some cases) especially if we try to train the model using CPU. Because a better option is training our model using GPU as they have proven to be much better in the domain, as they can process multiple computations simultaneously. Also, the GPU memory bandwidth is more suitable for the computation of such artificial intelligence models which has huge amounts of data to be handled.

Due to this fact, and because of a previous very bad experience of working with machine learning model using CPU on my computer. Google Colab came to be a solution for this problem as it provides GPU opportunity and it is easier and faster to deal with (No env preparation needed, faster runs...) then we decided to use it.

But there were still many limitations and difficulties to deal with during working with Colab because: Google Colab notebooks have an idle timeout of 90 minutes and an absolute timeout of 12 hours. This means, if a user does not interact with his Google Colab notebook for more than 90 minutes, its instance is automatically terminated. Also, the maximum lifetime of a Colab instance is 12 hours (And from my experience, most of the time an absolute timeout will happen way much before the maximum lifetime).

And in order to proceed with all the stages (Data preparation, Model Training/Testing...) with a big dataset, it takes much more time (such models can take up to 24 or 48 hours), then it was not possible with the currently available tools to proceed with the initial plan. And this caused a lot of additional work to try to avoid these limitations as much as possible and come up with the best possible result.

For instance, we followed some techniques. Such as, taking data samples of different sizes and

perform different experiments to determine what is the maximum size that the environment can handle before timing out. We also had to save every data (dataframes, arrays, results...) we generate into files organized into folders in the drive and load them back whenever we need them, instead of generating them again. Which have saved a lot of environment resources, but took some time and effort to achieve.

4.8 Results

After dealing with the limitations, we were able proceed and perform many training attempts before reaching a final succeeding attempt.

In our first set of runs we have a huge loss, which made us obtain a model with null accuracy. i.e the model wasn't able to learn anything. This was expected because at the first run, many hyper parameters had to be arbitrary chosen.

Then after gradually working on the hyper parameters and all the attributes in the system, we were able to make the loss decrease gradually in each run (figure 4). And this resulted an increase in the accuracy of the model. And in the final run we succeeded to train our model to predict future taxi demands with an accuracy of **89%**.

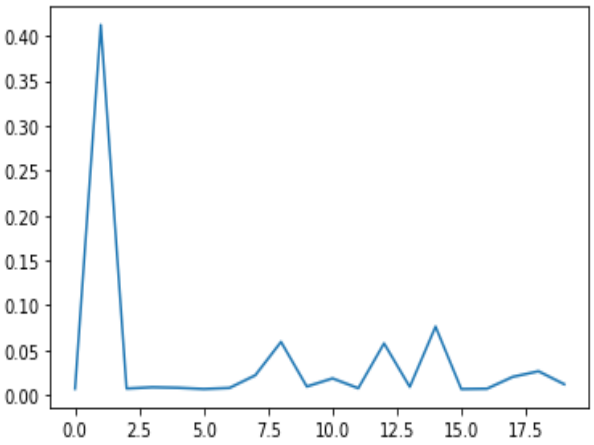


Figure 4: This Graph shows that variation of the loss with respect to time in our final run.

5 Conclusion

Consequently, we were able to overcome all the limitations and succeed to build a machine learning model that can predict future taxi demands, in 263 different zones in New York City, at a certain time slot. We also succeed to train this model and evaluate it with 89% accuracy. But building this model was only a part of our goal. We are also interested in adding this model the the agent's belief base to see how it could improve the quality of agent's planning decision, through comparing their decisions before and after the insertion of the model using different request allocation mechanisms already existing in an multi-agent simulation.

But because of resources limitations, time limit and other difficulties we were only able to achieve a theoretical study (in section 3) of how the agents would behave after the insertion of the model (section 3.2.3) without having the opportunity to preform this part of the work in the simulation as a real experiment. Therefore, this can be considered as a future related work where we could test the effect of this model through performing 2 different experiments in the multi-agent simulation. The first experiment is to observe the default behavior of the agents in the system and the allocation results before adding the model. And for the second experiment we need to use our prediction model in a new cost equation (section 3.2.1) in the agent's belief base and to study their new behavior in the system.

References

[1] Alaa Daoud, Flavien Balbo, Paolo Gianessi, Gauthier Picard. ORNInA: A Decentralized, Auctionbased Multi-agent Coordination in ODT Systems: Online Rescheduling with Neighborhood exchange, based on Insertion heuristic and Auctions. AI Communications, IOS Press, 2021, 34 (1), pp.37–53. 10.3233/AIC-201579. hal-03037353

[2] Alaa Daoud, Flavien Balbo, Paolo Gianessi, Gauthier Picard. A Generic Multi-Agent Model for Resource Allocation Strategies in Online On-Demand Transport with Autonomous Vehicles. Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), May 2021, London, United Kingdom. pp.3. hal-03093017

[3] Jun Xu, Rouhollah Rahmatizadeh, Ladislau B¨ol¨oni and Damla Turgut Department of Electrical Engineering and Computer Science University of Central Florida, Orlando, FL "A Sequence Learning Model with Recurrent Neural Networks for Taxi Demand Prediction" 2017 IEEE 42nd Conference on Local Computer Networks

[4] Doyup Lee1, Suehun Jung2, Yeongjae Cheon, Dongil Kim, Seungil You "forecasting taxi demands with fully convolutional networks and temporal guided embedding" at openreview

[5] Ioulia Markou∗, Kevin Kaiser, Francisco C. Pereira "Predicting taxi demand hotspots using automated Internet Search Queries" Transportation Research Part C 102 (2019) 73–86

[6] K. Zhang, Z. Feng, S. Chen, K. Huang, and G. Wang, "A framework for passengers demand prediction and recommendation," in Proc. of IEEE SCC'16, June 2016, pp. 340–347.

[7] R.R.S. van Lon, T. Holvoet, G. Vanden Berghe, T. Wenseleers and J. Branke, Evolutionary synthesis of multi-agent systems for dynamic dial-a-ride problems, in: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion - GECCO Companion '12, ACM Press, Philadelphia, Pennsylvania, USA, 2012, p. 331. ISBN 978-1-4503-1178-6. doi:10.1145/2330784.2330832.

[8] Luis Moreira-Matias, João Gama, Michel Ferreira, João Mendes-Moreira, Luis Damas "Predicting Taxi-Passenger Demand Using Streaming Data" IEEE Transactions on Intelligent Transportation Systems (Volume: 14, Issue: 3, Sept. 2013)