INDIAN INSTITUTE OF TECHNOLOGY, MADRAS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

B.TECH PROJECT

Project Report

Implementation of Reinforcement Learning on Custom Railways Simulator

Isha Nayar - CS21B035
under Prof Narayanaswamy N S
Prof Chandrashekhar Lakshminarayanan

Contents

1 Acknowledgement 3

2 Abstract 3

3 Introduction 3

4 A look at OSRD 4

5 Introduction to Custom Simulator 5
- 5.1 Structure of the Simulator . . . 5
- 5.2 Event Triggers . . . 6
- 5.3 Passenger Generation . . . 6
- 5.4 Signaling System . . . 7

6 Working of the Simulator 7
- 6.1 User Input format . . . 8
- 6.2 Simulator Working . . . 9
  - 6.2.1 Section and Junction Updation . . . 9
  - 6.2.2 Logic for Bi-directional route: . . . 9
- 6.3 Output format . . . 10

7 Introduction to Reinforcement Learning 12
- 7.1 Q-Learning Algorithm . . . 13
- 7.2 Deep Q-Learning . . . 13

8 Integration of Custom Simulator with RL 14
- 8.1 Modification to initial Simulator . . . 14
- 8.2 State Space Initialization and Extraction . . . 14
- 8.3 Action Space . . . 15
- 8.4 Reward System . . . 15
- 8.5 DQN Workflow with the Train Simulator . . . 16

9 Experimenting RL implementation on our Scenario 18

10 Results for the Common Junction Network Using DQN 20

11 Extending our study to Chennai Metro 20

12 Further Work 21

13 Github Repository 21

1 Acknowledgement

This work has been done primarily because of the guidance of Prof Narayanaswamy N S and Prof Chandrashekhar Lakshminarayanan. I thank them for their constant guidance and support. I also thank Adithya and Priyanga for guiding us throughout the project. I thank Akanksha Agrawal ma'am for her coordination of the BTP course.

2 Abstract

This projects presents the integration of our custom simulator with Reinforcement Learning to aim at optimizing train scheduling. The simulator handles train transitions and passenger handling and incorporates DQN agent to minimize the overall delay. Our main scenario to focus on is a network involving a common junction shared by intersecting railway lines, allowing analysis of whether trains should wait for transferring passengers or proceed to minimize overall delay.

We will look at understanding the building of our custom simulator and analyzing RL algorithms on our model, while also keeping track of passenger activity and train time-tabling. We will also examine a real world case study of the Chennai metro and analyze how RL will help us to improve scheduling for networks with two intersecting junctions as seen in the Chennai metro model.

3 Introduction

A railway system is a complex network of interconnected tracks, where train traffic is controlled via the signaling system using signals and switches. Creating a schedule is a time-consuming process and our main goal is to assign a route to a train minimizing the total delay of the entire train system. We also wish to minimize the delay faced by passengers while they are waiting for the train. Possible delays could be stochastic delays or delays due to construction work or even train malfunction.

In order to address the issue of train scheduling and passenger handling, we have developed a custom railway simulator and integrated it with a Deep Q-Network (DQN)-based reinforcement learning (RL) framework to model intelligent and adaptive train control.

Our simulator takes care of elements such as train transitions between junctions, passenger boarding/deboarding behavior, and line switching at common junctions. In this environment, decisions regarding whether a train should halt or proceed at a junction impacts the overall passenger delay. The DQN agent uses these states to learn optimal actions—either to proceed or wait based on a reward function derived from passenger delays.

Our goal is to make our simulator based on real-life situations and integrate it with the DQN algorithm to help in automating decision making and signaling system. This is further tested out by creating a network that involves intersection of 2 lines and taking care of passenger transfers as well.

4 A look at OSRD

OSRD (Open Source Railway Designer) is a web application for designing railway infrastructures, timetabling, simulation and capacity analysis. OSRD is meant to fulfill a range of use cases related to railway planning:

- **Timetabling:** Timetabling is about designing a transport service to meet transit needs.There are three steps to follow for the same:

  1. **Creation of Infrastructure**: The Infrastructure Editor in OSRD allows users to either modify an existing infrastructure or create a new one from scratch. Certain essential elements are required to build a valid infrastructure such as Track, Buffer stop, Detectors, Signal and Route.
  2. **Creation of Rolling Stock**: The creation of rolling stock requires specifying details about the trains, such as the duration of the start, the acceleration of the start, the length of the train, the maximum speed and the inertia factor.
  3. **Adding rolling stock** to the required infrastructure to generate timetable :
     The Operational Studies in the OSRD UI helps us to add a train to our infrastructure and give the timings at the start point and the destination. Once the train is added, we can track the train at any point till it reaches the end point and also get the timetable.We have to ensure that the infrastructure does not have any error and that it consists of a block so that we can successfully do the pathfinding which is required to add a train. A block should consists of a path (including buffer stop and detectors) and a signal.

- **Operation studies:** Operation studies helps us to analyze demand and balance infrastructure capacity with transportation needs

- **Infrastructure management:** OSRD helps us in making customised map and provides infrastructure editor and a search features.

- **Short term capacity management:** This is a developing feature of OSRD to help keep trains dynamic to accommodate the demands at a station. This feature has not yet being implemented.

5 Introduction to Custom Simulator

On exploring and understanding the features of OSRD, we realized that it does not facilitate the handling of delays based on passenger capacity and it is a complex process to interact with the signals of an infrastructure. Our ultimate aim is to try to modify the train schedule based on various delays and incorporate passenger waiting times as well. We would like to try to help passengers reach their destination as soon as possible and our main aim is to look at scenarios where passengers might switch lines. For this purpose, we chose to create our own Custom simulator.

5.1 Structure of the Simulator

The custom simulator consists of five classes: Train, Section , Junction, Line and Passenger.

- **Train**: The train class has attributes to hold the current section it is residing in, section entering time and section leaving time along with basic information on the train. It also has information about the number of passengers in the train at a time and the capacity of the train.
- **Section:**The section class has attributes to hold the information on the current train resting on it, train entering and leaving time from the section along with basic information on the section like the starting and ending junction and the line it is a part of.Section tells us how much gap should be there between 2 trains. At a time, there should be a gap of atleast 1 section between them.
- **Junction:** The junction class has attributes to store its coordinates and details about the signals and their status. Our updated junction also holds the information of the train currently at the junction along with the entering and leaving time of the train. Every junction that has a signal will be considered a station. Every station has a junction queue which stores the passenger waiting at the junction for the train to arrive.
- **Line:** The line class stores the line number and the corresponding sections list and the junctions list of a particular line.
- **Passenger:** The passenger class has attributes that hold the information of a passenger such as passenger name, arrival time at a junction, boarding station and destination station. It also stores the information on the time that it deboards the destination and the delay of the passenger.

5.2 Event Triggers

The simulator works on the basis of event triggers. At any given time the infrastructure holds the information on signals, trains and which sections they are presently running. Any moment a train crosses a junction or signal changes its status is classified as an event and every time an event occurs it triggers the simulator to change its data.

The simulator takes two inputs: The infrastructure and the signal scheduler and compiles the timetable if possible. There are two errors that can happen :

1. When two trains may enter the same section
2. When two trains may enter the same junction

We need to adjust the signal scheduler so that these error can be prevented. It is also important to note that the number of signals and their placement plays a significant role in creating a successful train timetable.

5.3 Passenger Generation

The primary objective in passenger modeling is to ensure that individuals reach their destination stations in the shortest possible time, thereby minimizing the delay experienced from the moment of arrival at the origin station to the completion of their journey.

- **Initial idea:** Passenger generation was initially modeled based on a uniform arrival rate at each junction. This rate, denoted by λ, represented the average number of passengers entering a junction per unit time and was uniformly applied across all junctions. The origin and destination junctions for passengers were randomly selected from the full list of available junctions. However, this approach was not recommended, as it did not account for the varying passenger arrival rate at different stations.
- **Modified Idea:** On examining our drawbacks, we decided to have a distinct value of λ for each of the stations. Boarding junctions are now explicitly specified, with the number of boarding passengers provided in the input for each junction which has a signal. For each passenger on board, the destination junction is randomly selected from the set of downstream junctions on the same line as the boarding station.

  Considering that we will be trying to implement RL in cases where there will be cross sections and passengers will be switching lines and going to destinations that are not in the same line as the boarding junction, we made sure to include junctions that come after the common junction of all lines to be included in possible destination and that passenger can switch.

5.4 Signaling System

In our simulation, junctions equipped with signals maintain a binary signal status:

- A status of 0 indicates that the signal is green, allowing the train to proceed without halting.
- A status of 1 indicates that the signal is red, requiring the train to stop at the junction for a specific period.

Initially, the signal behavior was implemented with a hardcoded wait time of the train. This allowed us to maintain consistent logs of train arrival and departure times at each junction.

We improved on the signaling mechanism to dynamically determine the halt time based on number of passengers queued at the junction awaiting boarding and the number of passenger in the train.

When a train approaches a station:

- If the train is not at full capacity and passengers are waiting, the system computes the halt time required to allow a subset of passengers to board, based on the train's remaining capacity and their arrival times at the junction.
- If the train is full or no passengers are waiting, a default minimal halt time is enforced (typically one minute) to simulate signal response.

This dynamic signaling strategy enables the simulation to more accurately capture the delay and waiting times of passengers. It also lays the groundwork for integration with reinforcement learning (RL) systems, where the signaling policy could help in optimizing delay.

6 Working of the Simulator

Our simulator takes the input text file and the output will be the details of all the trains along with their current junctions and their arrival and leaving time at each junction.

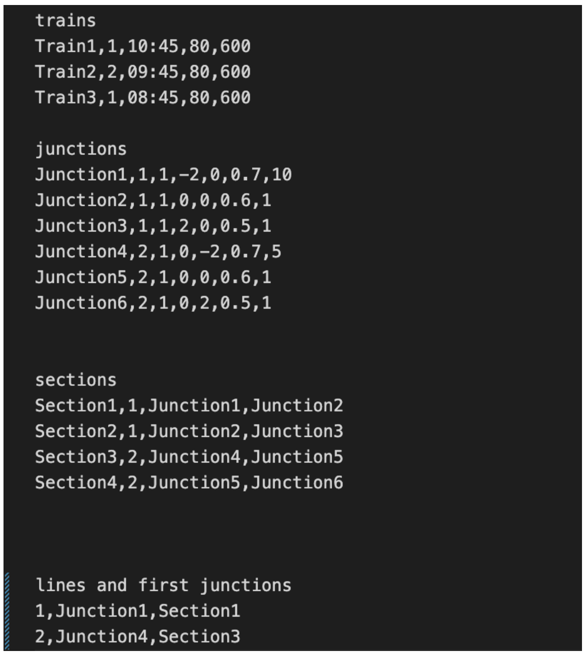
We have also generated a Final passenger list text file which gives us the information of passenger such as the name, destination junction, final line the destination is in, the arrival at boarding junction and the departure at the destination junction. We have also calculated the **Delay** of each passenger which is defined as the average time taken by the passenger to travel 1Km. The delay is updated whenever the passenger is deboarding at the final destination. In cases where a passenger switches lines, the delays accumulated from the initial deboarding junction and the final destination are summed to compute the total delay.

6.1 User Input format

The input text file consist of the following parameters:

- We have the train information in the following order: Name, Line number, Start time , Speed and Capacity.
- The data of junctions follow the train details. The Junction information is provided in the following order: Name, Line number, Signal, X coordinate, Y coordinate.
  For junctions with signal 1, we also have the Lambda value for that junction and the number of passengers.
- Next, the section details are provided. This will contain the Section name followed by the line number , the start junction and end junction of the section.
- We need to know the initial section and starting junction of a line so that we can determine the direction of the train in the line.

An example of input text file for our case of cross section scenario where 2 lines meet at a common junction (2 junctions having same coordinates) is given below.



```
trains
Train1,1,10:45,80,600
Train2,2,09:45,80,600
Train3,1,08:45,80,600

junctions
Junction1,1,1,-2,0,0.7,10
Junction2,1,1,0,0,0.6,1
Junction3,1,1,2,0,0.5,1
Junction4,2,1,0,-2,0.7,5
Junction5,2,1,0,0,0.6,1
Junction6,2,1,0,2,0.5,1


sections
Section1,1,Junction1,Junction2
Section2,1,Junction2,Junction3
Section3,2,Junction4,Junction5
Section4,2,Junction5,Junction6



lines and first junctions
1,Junction1,Section1
2,Junction4,Section3
```

Figure 1: Input file

6.2 Simulator Working

The simulation process begins with passenger generation. Passengers arriving at a junction can board a train if it arrives later and if the train has available capacity. At each train arrival event, the signaling system determines the duration for which the train halts at the junction. The simulator maintains an Asset List, which contains active trains and manages logic during event triggers.

6.2.1 Section and Junction Updation

The event trigger would happen when one train leaves a section. Since we are considering the section leaving time of a section, we have initialized a dummy section for each of the line.The logic for determining the next section and next junction on a line is based on the relationship between sections:

- The end junction of the current section matches the start junction of the next section.
- When a train transitions from one section to the next,
  1. Its arrival time in the new section is set to the departure time from the previous junction.
  2. The train's current section information is updated accordingly based on the next section function.

When a train reaches the final junction of a line, it is removed from the Asset List, signifying the end of its journey. During each event trigger, if a decision needs to be made, the leaving time of the train from its current section is updated accordingly.

6.2.2 Logic for Bi-directional route:

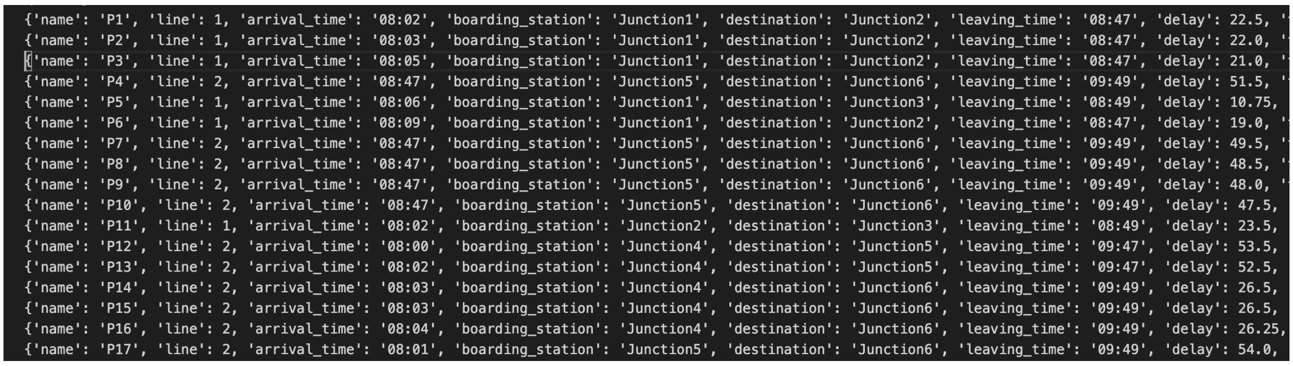
We incorporated a Line class which would help us identify the line and the section sequences in the line, helping us in finding the direction of the train. In the opposite direction scenario, the sections and junctions will be defined in the input text file. Initially we had same section name and we introduced reverse line using line class and changed the section sequence to be backwards. However this allowed two trains to be in the same section (this will occur when they are in different lines) and this is contrary to our initial assumption that there will be only one train in the section.

To now introduce train in opposite direction, we need to introduce new Lines and have distinct name for the junctions and sections in the opposite direction.

6.3 Output format

The output consists of the following:

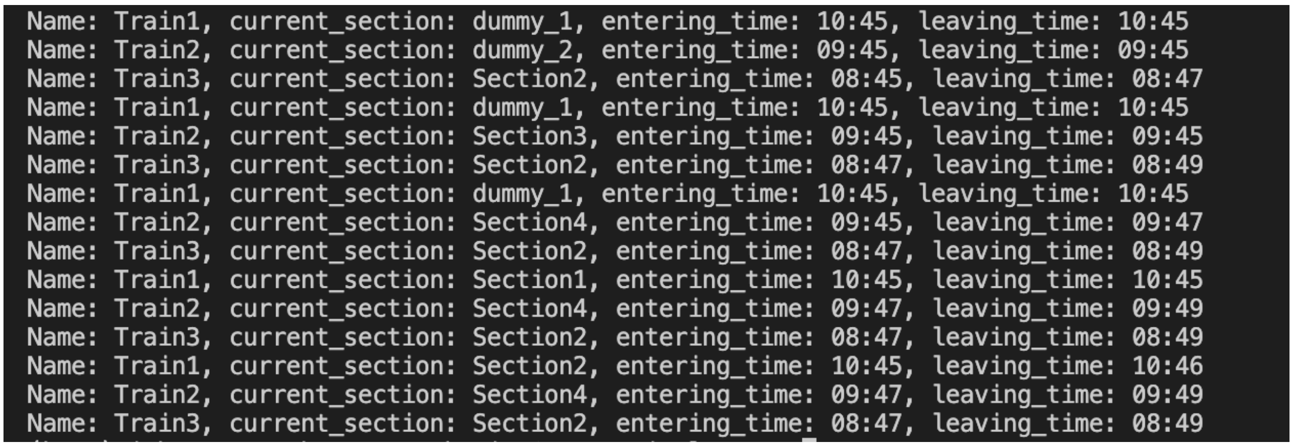
- **Final Passenger List:** The final passenger list is the list of all passengers and information about their final line number , updated boarding station(changes in case of switching), final destination,final leaving time(i.e the time that the passenger arrives at the destination junction) and delay. Here is a sample FinalPassenger.txt:



```
{'name': 'P1', 'line': 1, 'arrival_time': '08:02', 'boarding_station': 'Junction1', 'destination': 'Junction2', 'leaving_time': '08:47', 'delay': 22.5, '
{'name': 'P2', 'line': 1, 'arrival_time': '08:03', 'boarding_station': 'Junction1', 'destination': 'Junction2', 'leaving_time': '08:47', 'delay': 22.0, '
{'name': 'P3', 'line': 1, 'arrival_time': '08:05', 'boarding_station': 'Junction1', 'destination': 'Junction2', 'leaving_time': '08:47', 'delay': 21.0, '
{'name': 'P4', 'line': 2, 'arrival_time': '08:47', 'boarding_station': 'Junction5', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 51.5, '
{'name': 'P5', 'line': 1, 'arrival_time': '08:06', 'boarding_station': 'Junction1', 'destination': 'Junction3', 'leaving_time': '08:49', 'delay': 10.75,
{'name': 'P6', 'line': 1, 'arrival_time': '08:09', 'boarding_station': 'Junction1', 'destination': 'Junction2', 'leaving_time': '08:47', 'delay': 19.0, '
{'name': 'P7', 'line': 2, 'arrival_time': '08:47', 'boarding_station': 'Junction5', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 49.5, '
{'name': 'P8', 'line': 2, 'arrival_time': '08:47', 'boarding_station': 'Junction5', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 48.5, '
{'name': 'P9', 'line': 2, 'arrival_time': '08:47', 'boarding_station': 'Junction5', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 48.0, '
{'name': 'P10', 'line': 2, 'arrival_time': '08:47', 'boarding_station': 'Junction5', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 47.5,
{'name': 'P11', 'line': 1, 'arrival_time': '08:02', 'boarding_station': 'Junction2', 'destination': 'Junction3', 'leaving_time': '08:49', 'delay': 23.5,
{'name': 'P12', 'line': 2, 'arrival_time': '08:00', 'boarding_station': 'Junction4', 'destination': 'Junction5', 'leaving_time': '09:47', 'delay': 53.5,
{'name': 'P13', 'line': 2, 'arrival_time': '08:02', 'boarding_station': 'Junction4', 'destination': 'Junction5', 'leaving_time': '09:47', 'delay': 52.5,
{'name': 'P14', 'line': 2, 'arrival_time': '08:03', 'boarding_station': 'Junction4', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 26.5,
{'name': 'P15', 'line': 2, 'arrival_time': '08:03', 'boarding_station': 'Junction4', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 26.5,
{'name': 'P16', 'line': 2, 'arrival_time': '08:04', 'boarding_station': 'Junction4', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 26.25,
{'name': 'P17', 'line': 2, 'arrival_time': '08:01', 'boarding_station': 'Junction5', 'destination': 'Junction6', 'leaving_time': '09:49', 'delay': 54.0,
```

Figure 2: Final Passenger list

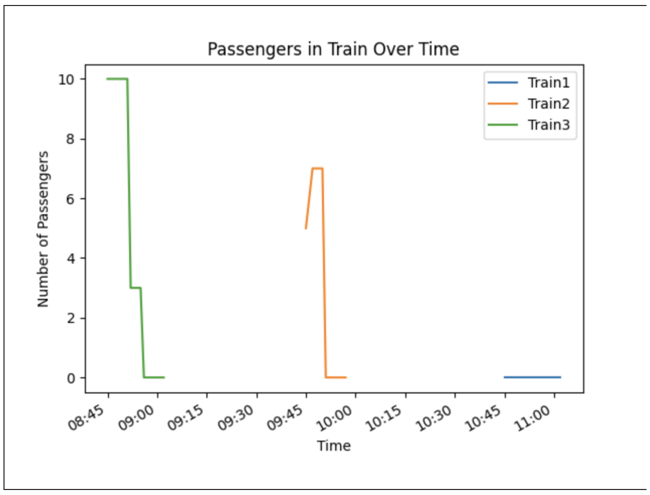
- **Time-table of the Trains:** Along with the passenger list, we'll also get the train time table which will consist of the train name, the current section it is in and the entering and leaving time at the section. This is printed every time there is an event trigger. An example of the train time-table is given below:



```
Name: Train1, current_section: dummy_1, entering_time: 10:45, leaving_time: 10:45
Name: Train2, current_section: dummy_2, entering_time: 09:45, leaving_time: 09:45
Name: Train3, current_section: Section2, entering_time: 08:45, leaving_time: 08:47
Name: Train1, current_section: dummy_1, entering_time: 10:45, leaving_time: 10:45
Name: Train2, current_section: Section3, entering_time: 09:45, leaving_time: 09:45
Name: Train3, current_section: Section2, entering_time: 08:47, leaving_time: 08:49
Name: Train1, current_section: dummy_1, entering_time: 10:45, leaving_time: 10:45
Name: Train2, current_section: Section4, entering_time: 09:45, leaving_time: 09:47
Name: Train3, current_section: Section2, entering_time: 08:47, leaving_time: 08:49
Name: Train1, current_section: Section1, entering_time: 10:45, leaving_time: 10:45
Name: Train2, current_section: Section4, entering_time: 09:47, leaving_time: 09:49
Name: Train3, current_section: Section2, entering_time: 08:47, leaving_time: 08:49
Name: Train1, current_section: Section2, entering_time: 10:45, leaving_time: 10:46
Name: Train2, current_section: Section4, entering_time: 09:47, leaving_time: 09:49
Name: Train3, current_section: Section2, entering_time: 08:47, leaving_time: 08:49
```

Figure 3: Train time-table

We have also visualized the number of passengers in each train over time to gain insights into passenger trends at any given moment. This visualization will help in our analysis when we apply reinforcement learning to our simulator, helping us understand how passenger counts evolve and how the system can optimize its actions accordingly. Below is an example plot showing the passenger count in each train over time.



7 Introduction to Reinforcement Learning

Reinforcement Learning(RL) is a type of Machine learning where an agent interacts with the environment and learn data through trial and error, unlike supervised learning where we learn from labeled data. The agent performs action and returns rewards or penalties in return. This will in turn be the input for the next state based on which the agent learns. The agent eventually learns an optimal policy that will choose the best possible action to maximize the rewards.
Here are a few important terms used in RL:

- **Environment:** This is the system with which the agent(the decision maker) interacts.
- **States:** The current situation that we are in and where an action is performed.
- **Action:** A decision that the agent makes. Based on the action, the agent will give a certain value which is the **Reward**.
- **Policy:** A strategy that the agent uses to determine an action that is to be chosen for a state.

Here is an overview of how agent interaction with the environment takes place in Reinforcement learning.

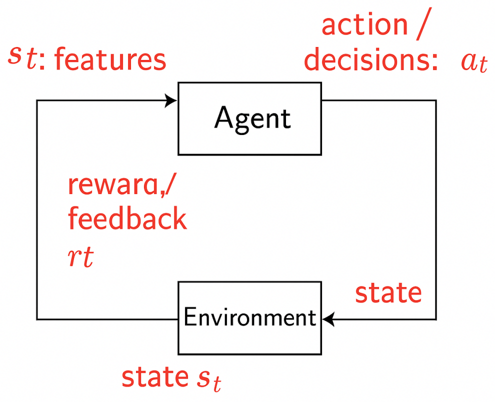


Figure 4: Interaction with Environment

We tried to understand RL and the way it works through **Gridworld** environment where we aim to go from one corner to the opposite end corner in minimum number of steps, and by maximizing the reward. The environment helped us in understanding how the decisions are made and how the agent learns from feedback.

7.1 Q-Learning Algorithm

Q-learning is a RL algorithm for learning the optimal action-value function $Q^*(s,a)$, which estimates the maximum expected cumulative reward achievable from state s by taking action a and following the optimal policy thereafter. The agent updates its Q-values using the following :

$$Q_t(s_t,a_t) \leftarrow Q_{t-1}(s_t,a_t) + \eta \left[R(s_t) + \gamma \max_{a'} Q_{t-1}(s_{t+1},a') - Q_{t-1}(s_t,a_t)\right]$$

where η is the learning rate, $R(s_t)$ is the reward received, γ is the discount factor, and $\max_{a'} Q_{t-1}(s_{t+1},a')$ is the estimated value of the best action in the next state.Q-learning finds the optimal policy by directly estimating the best possible future rewards for each action in every state.

7.2 Deep Q-Learning

Deep Q-learning (DQN) is a reinforcement learning algorithm that combines Q- learning and neural network. DQN helps to learn optimal policy in complex environment such as our custom simulator by approximating the Q function. Important terms in DQN:

- **Experience Delay:** DQN stores past experiences and sample them randomly to update the Q-network.
- **Target network:** This helps in computing the target Q value during training.
- **Loss function:**The loss function is the mean squared error between the predicted Q-values and the target Q values.

The Q value is predicted by the network and the loss is defined as:

$$\text{Loss} = \left[R(s_t) + \gamma \max_{a'} Q(s_{t+1},a';\theta_{t-1}) - Q(s_t,a_t;\theta_{t-1})\right]^2$$

8 Integration of Custom Simulator with RL

We chose to apply **Deep Q-Learning (DQN)** algorithm on our simulator.

- The DQN algorithm will help us in determining the action based on the current state and gives us the reward which will help us in determining the next state.
- DQN is useful in dynamic environment such as our simulator to try improving rewards.
- Since we use Neural network, it is easier to process large number of states (such as train name, train capacity, junction capacity) in our scenario.
- DQN considers the upcoming state also was making decision to stay or go at a signal.The decision making is sequential.

8.1 Modification to initial Simulator

Initially, our custom simulator was designed to trigger an event when a train was leaving a section. However, this approach is not suitable for our reinforcement learning integration. In our use case, we intend to apply DQN model at each decision point, based on the current state, to determine the next action and compute the corresponding reward. For this purpose, it is necessary to trigger events when a train enters a section, rather than when it leaves.

Since the simulation initializes with a dummy starting section and the existing logic relies on the section leaving time, we have revised the implementation such that it tracks the time when a train enters a junction i.e, event trigger will happen at the entering time of current junction. This will help us apply DQN on the current state and determine our action and calculate reward. We have added junction attribute to the Train class and are maintaining the entering and leaving time of a train at a junction.

8.2 State Space Initialization and Extraction

For every junction in our environment (railway simulator), the following features are collected to be a part of our state space for the DQN algorithm:

- **Junction Number:** We have extracted the junction number from the junction name given in the input file.
- **Train Number:** The train number is extracted from the train name of the train that is present in the junction at an instant. If there is no train in the junction, we have given default value to be -1.

- **Number of passengers waiting:** The number of passengers waiting at the junction queue at an instant is also considered. In case there are more number of passengers waiting, it will help us to keep the train waiting at the junction for more time.
- **Train remaining capacity:** This refers to the number of passengers that can board the train at the junction before the train gets full. This feature will help us determine whether the train should halt for capacity to be full or move.

The states are stored in a list of vectors.Each vector represents the 4 features at a junction. The vector list is converted a matrix of shape *(number_ of_junctions* ,4).

This matrix is passed to **CNN** (Convolutional neural network) model to help flatten the state space to be available to use by the DQN algorithm. The CNN model helps in maintaining symmetry across all junctions to ensure that all similar state patterns are processed consistently.

8.3 Action Space

There are 2 actions that are defined for our DQN agent. They are 0 and 1.

- **action= 0:** If the action is 0, it would mean that the train can proceed without waiting (the signal is green).
- **action= 1:** If the action is 1, the train will have to wait for a given halt time. In our scenario we have currently given a halt time of 3 minutes.

8.4 Reward System

The reward for the DQN algorithm in our simulator is calculated as the passenger delay after a state is achieved. We are calculating negative passenger delay such that we hope to minimize the delay by making efficient decision through DQN. The passenger delay was defined as the time taken by the passenger to travel 1 km. The reward is also updated whenever the passenger is switching lines. The agent receives a penalty whenever the passenger has to wait a long time to board a train and if the train leaves just a few seconds before the passenger could switch lines and board it at the new line.

$$\text{Reward} = -(\text{Passenger Delay})$$

$$\text{Passenger Delay} = \frac{\text{Total time since arriving at junction to reaching destination junction}}{\text{Distance traveled}}$$

This reward is then used by DQN agent to update the Q-value using the Bellman equation.

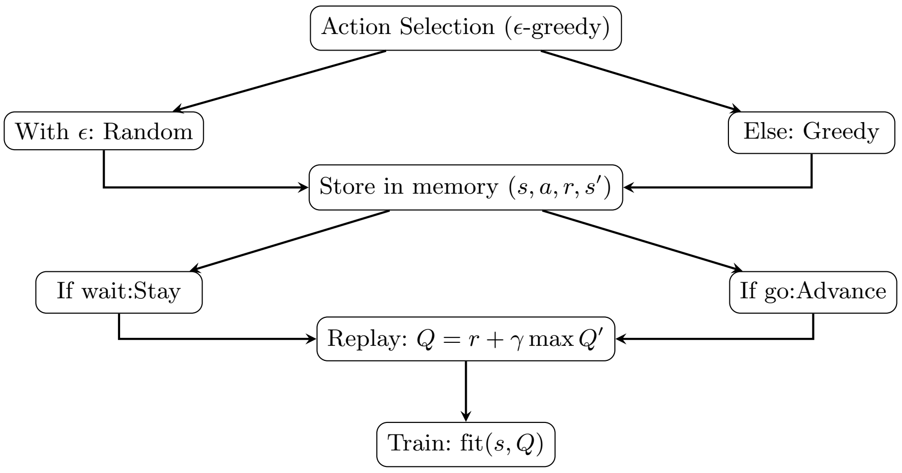
8.5 DQN Workflow with the Train Simulator

The Deep Q-Network (DQN) algorithm is integrated into our simulator by treating the simulator as a custom environment. The key steps in the DQN training loop are as follows:

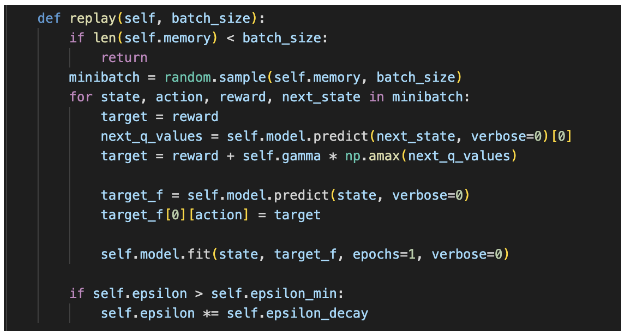
- **Action Selection:** The agent selects an action using an **epsilon-greedy** strategy:
  1. With probability ϵ (exploration), it selects a random action.
  2. Otherwise (exploitation), it selects the action with the highest predicted Q-value from the neural network.
- **Environment Interaction:**
  1. If the chosen action is wait, the train remains at its current location, and a halt time is added. The next state will be the same
  2. If the action is go, the train moves to the next junction, and the simulator updates its state using the `get_state()` function.
- **State Transition:** After the environment responds to the action, the agent observes the new state and reward. It stores the experience tuple $(\text{state}, \text{action}, \text{reward}, \text{next\_state})$ in its replay memory.
- **Experience Replay:** Periodically, a mini-batch of experiences is sampled from the replay memory. For each tuple in the batch, the agent computes the target Q-value using the Bellman equation:
  $$Q(s, a) = r + \gamma \cdot \max_{a'} Q(s', a')$$
- **Neural Network Training:** Within the `agent.replay(batch_size)` function:
  1. The target value is computed:
     $$\text{target} = \text{reward} + \gamma \cdot \max(\text{model.predict}(\text{next\_state}))$$
  2. The neural network is trained using:
     $$\text{model.fit(state, target)}$$

  This training process gradually improves the Q-value approximations, enabling the agent to make better decisions over time.

An overview of the DQN algorithm is as shown:


Action Selection (ϵ-greedy)
With ϵ: Random
Else: Greedy
Store in memory (s, a, r, s')
If wait:Stay
If go:Advance
Replay: $Q = r + \gamma \max Q'$
Train: fit(s, Q)

Here is a snippet of the DQN agent's build function(where the Neural network is defined) and the replay function where the target value is computed and model is trained:



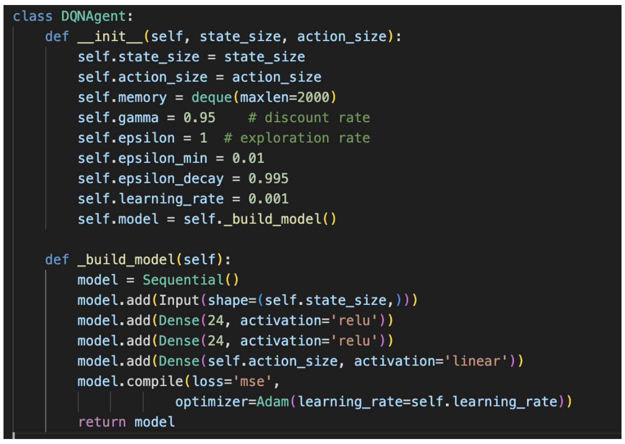
```python
def replay(self, batch_size):
    if len(self.memory) < batch_size:
        return
    minibatch = random.sample(self.memory, batch_size)
    for state, action, reward, next_state in minibatch:
        target = reward
        next_q_values = self.model.predict(next_state, verbose=0)[0]
        target = reward + self.gamma * np.amax(next_q_values)

        target_f = self.model.predict(state, verbose=0)
        target_f[0][action] = target

        self.model.fit(state, target_f, epochs=1, verbose=0)

    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
```

Figure 5: DQN agent's Replay function



```python
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Input(shape=(self.state_size,)))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse',
                      optimizer=Adam(learning_rate=self.learning_rate))
        return model
```

Figure 6: DQN agent's Build function

9 Experimenting RL implementation on our Scenario

To effectively study delay minimization while maximizing passenger accommodation, we designed a scenario where two railway lines intersect at a common junction. At this junction, passenger will be able to switch lines and board a train to a destination on the other line.

Having such a scenario will help us study how much time should a train wait for passengers that are going to board it after they have switched lines.Specifically, we evaluate whether a train should wait for transferring passengers or proceed to minimize delays for those already on board.

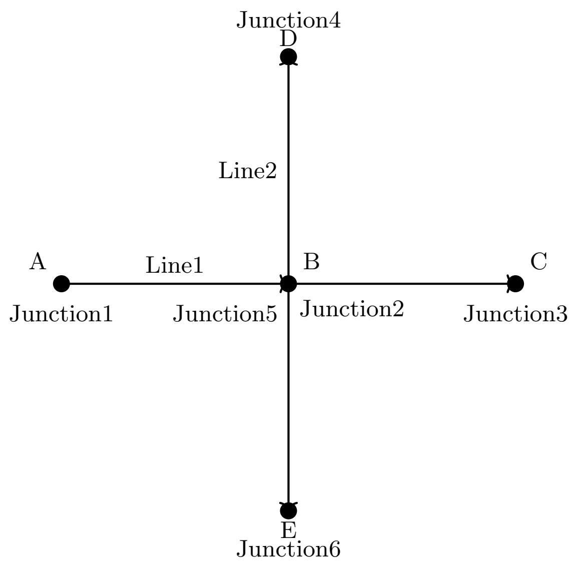
- If the number of onboard passengers significantly exceeds the number of transferring passengers, it may be preferable for the train to depart without delay.
- Conversely, if many passengers are expected to switch trains, it seems more convinient if the train waits a few minutes more for these passengers to make the switch so that the passengers don't have to wait for the next train and we minimize the delay.

Note: Common junctions have same coordinates and they do not have same name as the common junction are a part of 2 different lines . Each line treats the junction as a distinct entity. This distinction is important, as only one train is allowed at a junction at any given time.Since these are 2 different lines, we have kept the coordinates of the common junction same and distinct Junction name.

The diagram of our case of having 1 common junction is given below. Junction2 and Junction5 have the same coordinates and are common junction of 2 lines. A,B,C are a part of Line 1 where Junction1, Junction2, Junction3 represents coordinates A,B,C respectively. D,B,E are coordinates of Junction4, Junction5, Junction6 respectively of the line 2. All junctions are at equal distance from the common junction B and are station i.e they have signal. We are currently focusing in forward direction i.e

$$A \to B \to C \quad \text{and} \quad D \to B \to E$$

Passengers boarding at Junction4 can deboard at Junction5, Junction6 or switch lines and deboard at Junction3.
Similarly, passengers boarding at Junction1 can be allowed to deboard at Junction2, Junction3 and Junction6.



According to our new logic of event trigger involving junctions, suppose the train T1 enters Junction1 at 08:00 am and assuming that it halts for 3 minutes (if the action is 1) and takes 2 minutes to reach Junction2. So the leaving time of Junction1 will be 08:03 and the entering time of the train at Junction2 will be:

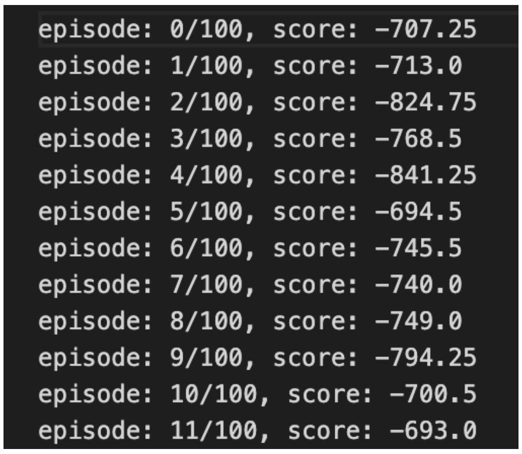
Train Leaving time of Junction1 + Time taken to travel to Junction2.

The train entering time at Junction2 will be 08:05. If the train does not halt, it's leaving time at Junction2 will be the same as the time that it entered Junction2.
For this case, passengers have been generated with first passenger coming at 08:00 am. Trains, section and junctions have been defined in the input text file.

10 Results for the Common Junction Network Using DQN

We have fed our network scenario featuring a common junction into our custom simulator. To evaluate the decision-making performance, we applied the DQN algorithm to this environment. The training loop was executed over 100 episodes, during which the agent interacted with the environment and accumulated rewards. For each episode, the total reward was recorded and saved as a text file for further analysis. In addition to that, the train time table for each episode and the final passenger list is also updated.

Here is a snippet of the rewards for the first few episodes:



```
episode: 0/100, score: -707.25
episode: 1/100, score: -713.0
episode: 2/100, score: -824.75
episode: 3/100, score: -768.5
episode: 4/100, score: -841.25
episode: 5/100, score: -694.5
episode: 6/100, score: -745.5
episode: 7/100, score: -740.0
episode: 8/100, score: -749.0
episode: 9/100, score: -794.25
episode: 10/100, score: -700.5
episode: 11/100, score: -693.0
```

Figure 7: Rewards on our network

The rewards are negative as we are considering negative passenger delay (we want to minimize the passenger delay) We can observe that the values are fluctuating. We have to do fine-tuning and sanity checks by updating and testing out different epsilon or gamma values and studying the changes to improve our results.

11 Extending our study to Chennai Metro

We aim to explore a model based on the Chennai Metro network, which features two common junctions between the Green and Blue lines. Studying this structure will provide deeper insights and help us analyze real-life optimization.
We have compiled detailed information on the junctions and sections by collecting the geographic coordinates of all stations along both lines, and prepared the necessary input data accordingly.

12 Further Work

1. Analyzing the results obtained from implementing the DQN algorithm on the one common junction network to understand its impact on train scheduling, passenger delays, and overall efficiency.
2. Running the simulation on the Chennai Metro network and experimenting with different values and configurations to observe how the model reacts .
3. Extending the RL-based signaling system to try learning the halt time and to better schedule trains and minimize delays experienced by passengers.
4. Incorporating additional real-world features such as weather conditions and train weight, which can influence delays, to make the simulation more realistic.

13 Github Repository

The codebase has been added to Github repository. The repository has a Readme file and 2 folders, one which has the custom simulator and one where we have implemented RL on our simulator.

https://github.com/ishanayar1/Custom_simulator.git

References

[1] Chennai metro. `https://chennaimetrorail.org/`.

[2] Deep q learning. `https://github.com/mingen-pan/Reinforcement-Learning-Q-learning-Gridworld-Pytorch/blob/master/DQN.py`.

[3] Osrd. `https://osrd.fr/en/about/`.

[4] Q learning on gridworld. `https://github.com/michaeltinsley/Gridworld-with-Q-Learning-Reinforcement-Learni` `tree/master`.

[5] Teodora Popescu. Reinforcement learning for train dispatching. 2022.