CS 4122: Reinforcement Learning and Autonomous Systems

Programming Assignment #1 (for Module 1)

Release date: 6th September, 2024, 2:00 am (IST)

Due date: 29th September, 2024, 11:59 pm (IST)

Maximum Score: 100 marks (this assignment is graded)

Read the following before you move forward:

1. This programming assignment is worth **10% of the total marks** of this course.

2. **Late policy:** You can be late by a maximum of 3 days. Your assignment will not be accepted more than 3 days after the due date. The **actual score** and **received score** (received score is actual score minus penalty for late submission) are related as follows:

$$received\ score = \begin{cases} actual\ score & ;late\ by\ 0\ days \\ 0.9 \cdot actual\ score & ;late\ by\ 1\ day \\ 0.7 \cdot actual\ score & ;late\ by\ 2\ days \\ 0.5 \cdot actual\ score & ;late\ by\ 3\ days \\ 0 & ;late\ by\ more\ than\ 3\ days \end{cases}$$

3. Plagiarism, if detected, no matter how big or small, will lead to a score of ZERO for all the team member; no exceptions! It is ok to take help from AI tools or talk to other teams. But, you cannot copy from another team or any AI tools.
    a. It is down right stupid to copy-paste from AI tools and think that is acceptable.
    b. Also, it is a bad idea to keep open another teams work in front of you while you do the assignment. While you may think you are just taking help, your reports and your codes will get heavily influenced by the other team to the point that it will qualify as being plagiarized.

4. This is a team project. You have to do the project with the team that you selected in the excel sheet that was shared with you.
    a. **Only one submission per team**.
    b. Submission must be made using the google drive link that was shared with you.
    c. **Your submission in the google drive link must contain ONLY FOUR files:**
        i. **report.pdf**
        ii. **lin_greedy.py**
        iii. **lin_ucb.py**
        iv. **policy_gradient.py**

    You must read section 5 carefully to understand what needs to be included in these four files. **DON'T submit any other files.** You MUST NOT zip/compress these four files. **You MUST NOT put these four files in a subfolder inside the google drive link.** Directly drop these four files in the google drive folder corresponding to your team.

Contextual Bandits for Millimeter Wave Vehicular Communication

In the first part of this programming assignment you will be implementing a contextual bandit based agent to improve millimeter wave communication.

Section 1: Installation

If you are using **Spyder** that comes with **Anaconda**, execute the pip command in **Anaconda prompt** to install the libraries required for this assignment (for **Colab**, pip should be replaced with !pip):

pip install gymnasium

pip install swig

Please Note: You need to install **OpenAI Gymnasium** and NOT OpenAI Gym *(**Gym** and **Gymnasium** are two different libraries that contains environments for reinforcement learning. Gymnasium is the newer version of Gym and supports more environments than Gym. Hence, install Gymnasium)*. Also note that my installation instructions assume the following:

1. You already have a setup to code in Python and run that code. Typically, this would mean that you have **Anaconda** installed in your system. Anaconda has both **Spyder** and **Jupyter Notebook** that can be used to code in Python. NOTE: Even if you are using Jupyter notebook for coding, the final codes that you submit must be Python scripts (.py files) as mentioned in the 1st page of this document.
2. You already have either **Keras, Tensorflow, or Pytorch** (these are all **Deep Learning libraries** that will be required to implement **policy gradient algorithm**) installed in your system.
    a. Preliminary survey indicates that some of you know Tensorflow but not Keras. Don't worry… Keras is a more user-friendly version of Tensorflow. You can learn it quite fast.
    b. You can code in Pytorch as well. But, I am not well versed with Pytorch. So, **partial grading will be difficult for me if you use Pytorch**; either your code runs or it does not. That said, I will still go through the code to ensure that there is no funny business!

Section 2: Introduction

Millimeter wave (mmWave) communication refers to the wireless communication using the electromagnetic spectrum in the range 30 GHz to 300 GHz (there are some variations in this range). This range of spectrum provides much higher bandwidth. However, mmWave can get absorbed by various entities like humans, concrete structures, trees, etc. One way to combat the absorption issue of mmWave is to use **highly directional transmission** instead of omnidirectional transmission. By using highly directional transmission, the entire energy of the electromagnetic wave is concentrated in a very small area. Hence, even if there is absorption, the signal received by the receiver is strong enough.

However, the problem with highly directional transmission is that if the receiver moves out of the transmission zone (which can be quite small), the communication link will get disconnected. This is specially a problem if the receiver is inside a vehicle travelling at a significant speed. For e.x., in Fig. 1, if the vehicle moves from position 1 to position 2, the communication link will get disconnected because position 2 is outside the transmission zone of the mmWave beam. Sometimes the absorption is so large that even if the mmWave beam is pointing correctly, the communication link can get disconnected. This is called **blockage** and is shown in Fig. 2 where the mmWave beam in yellow gets blocked by a tree even though it is pointing in the correct direction. In this situation, another base station has a better chance of establishing a communication link because there is no blockage in its path (like base station 2 in Fig. 2).

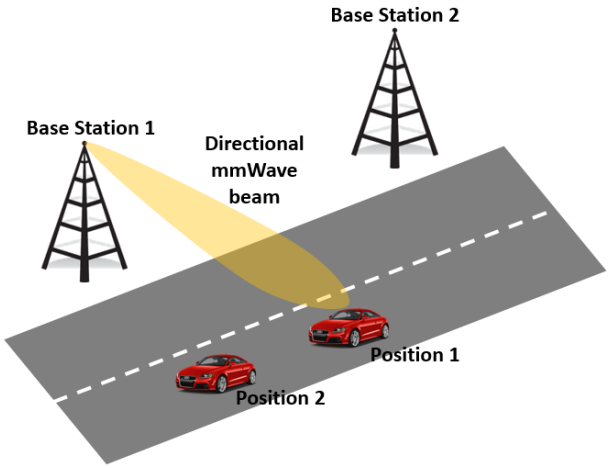

Fig. 1

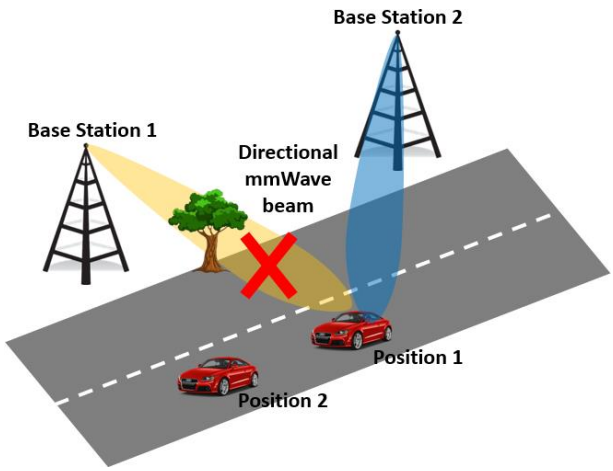

Fig. 2

So essentially, based on the position and velocity of the vehicles **(the context)**, the base stations has to decide which base station will transmit and in which direction to point the mmWave beam **(the action)** with the objective of maximizing the net received power[1] **(the reward)** over a time horizon. Hence, **base station selection** and **beam alignment** for mmWave communication can be modeled as a contextual bandit setup. An obvious question arises: **Why are we using contextual bandit approach to solve this problem instead of the conventional supervised learning approach?**

Answer: We can definitely use supervised learning for this problem. To do so, we need to collect data relating the best beam direction for different position and velocity of vehicles. Then we can train a supervised learning model for each location. The problem is that this data will be widely different for different location because of different sources of blockages (both moving and stationary) and the overall radio-frequency environment in general. So, **we need to collect data and train a separate supervised learning model for each location** which is cumbersome. In comparison, a contextual bandit approach is more suitable here because it is an **online learning** approach. And being an online learning approach, it will collect data online and customize itself according to the location.

[1] The received power is a viable candidate for the reward because it determines the data rate and the strength of the wireless communication (the bars that you see on top of your mobile screen).

Section 3: The Gym Environment

(NOTE: Even though we will keep saying "Gym" we actually mean "Gymnasium".)

You are given a Python file **mmWave_bandits.py** that has a class **mmWaveEnv**. This class implements a simplified version of **"the environment"** that simulates base station selection and beam alignment for mmWave communication. mmWaveEnv implements a 2D version of the setup discussed in the introduction. This setup is shown in the figures below.

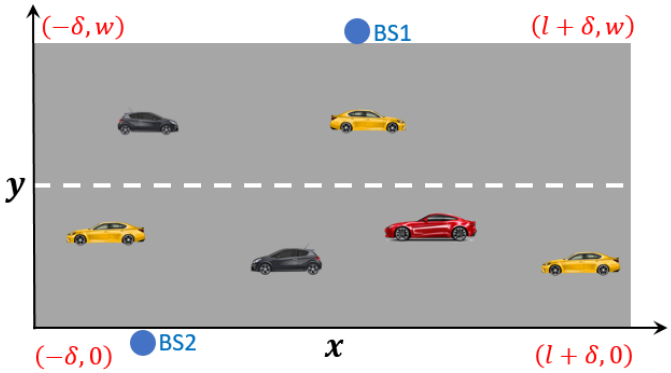

Fig. 3

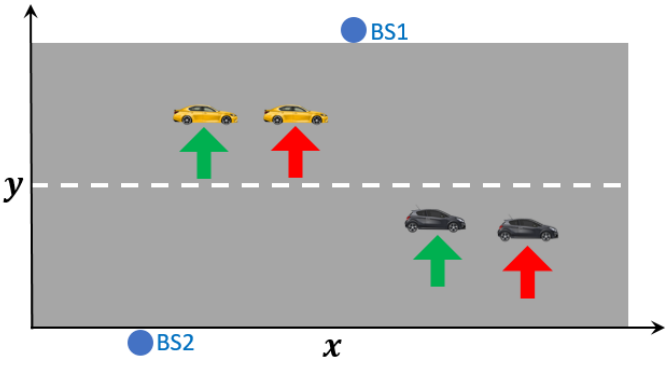

Fig. 4

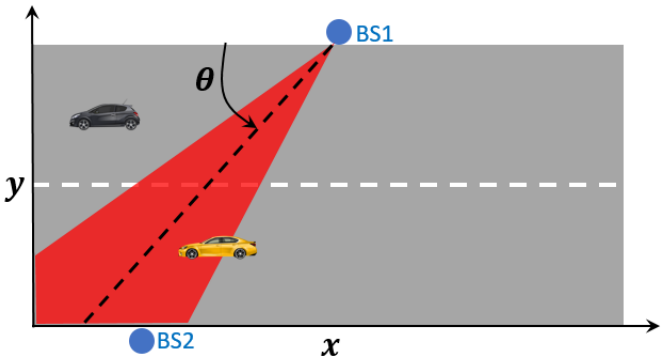

Fig. 5

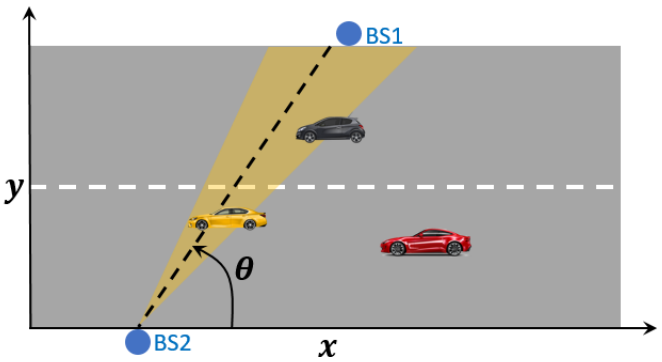

Fig. 6

Fig. 3 shows the overall geometry of the setup. There are two base stations, BS1 and BS2 (shown using the blue circle), that have to handle all the communications for a certain section of a road. This section of the road is a rectangle whose corner coordinates are shown in red in Fig. 3. For the given setup, $l = 100\ m$, $w = 7\ m$, $\delta \approx 10\ m$ (this selection of parameters are based on some real-life data). BS1's and BS2's coordinates are $(50\ m, 7\ m)$ and $(20\ m, 0\ m)$ respectively. This road has two lanes, one for each direction. Apart from what is shown in Fig. 3 there are also blockages, both moving and stationary, that can block communication between the base stations and the receivers.

Any environment for RL is associated with the concept of **episode**. An episode is a sequence of consecutive time slots associated with a certain task. An episode starts from time slot $t = 0$ and continues till a prespecified **optimization horizon** H. After $t = H$, the episode ends. After an episode ends, we have to **reset the environment** in order to start a new episode back from time slot $t = 0$. In our setup, the task is mmWave vehicular communication, a time slot is of 5 seconds duration, and an episode is of $1\ hour$ duration, i.e. $H = 720$ ($1\ hour$ divided into 5 seconds time slot).

In a given time slot, the vehicles (or the mobile phone inside the vehicles) that wants to communicate measure its position and velocity using its GPS and sends it to both the base stations. The number of vehicles that wants to communicate at any given time slot is a random number between 1 to 4. To elaborate on this, we use Fig. 4. In Fig. 4, there are two vehicles that wants to communicate. The current position of both the vehicles are shown using the green arrow. These vehicle measures its current position and velocity and sends it to both the base stations. Needless to say, these measurements are subjected to random noise. So, the **context c_t** of the base stations at time slot t is a tuple of the form,

$$c_t = \left(\left[x_{1,t}, y_{1,t}, v_{1,t}\right], \left[x_{2,t}, y_{2,t}, v_{2,t}\right], \cdots, \left[x_{n,t}, y_{n,t}, v_{n,t}\right]\right)$$

where n is the total number of vehicles that wants to communicate in time slot t (in Fig. 4, $n = 2$), and $\left[x_{i,t}, y_{i,t}, v_{i,t}\right]$ consists of the x-position, y-position, and velocity of the i^{th} vehicle, where $1 \leq i \leq n$. Note that:

1. n is a random number between one and four. n changes every time slot. Hence, the size of contexts changes every time slot. This is a challenge for most ML models because they assume that the input size is fixed.
2. Velocity $v_{i,t}$ contains direction as well. Minus if going from right-to-left (bottom lane in Fig. 4) and plus if going from left-to-right (top lane in Fig. 4).

After the base stations receives the context, it has to make its decision. The time taken by the base stations to take an action is not negligible. By the time the base stations take an action, the vehicles would have moved to another location (shown using red arrows in Fig. 4). After the base stations receives the context, both base stations have to **collectively** decide:

1. Which base station will transmit. Only one base station can transmit in a time slot.
2. The base station that transmits have to choose the beam direction.

To this end, **the action** for this setup is a tuple,

$$(base\ station, beam\ direction)$$

where, $base\ station \in \{0,1\}$ and $beam\ direction \in \{0,1,\cdots,9\}$. $base\ station = 0$ means BS1 and $base\ station = 1$ means BS2. In our setup, there are 10 beam directions indexed 0 to 9. Fig. 5 and Fig. 6 shows one such beam direction for BS1 and BS2 respectively. The beam directions are marked using angle θ and the dotted-black line. θ is between 0° and 180°. A higher value of θ implies higher value of $beam\ direction$. This essentially means that for:

1. BS1 shown in Fig. 5, $beam\ direction = 0$ is the one that points to left and $beam\ direction = 9$ is the one that points to right.
2. BS2 shown in Fig. 6, $beam\ direction = 0$ is the one that points to right and $beam\ direction = 9$ is the one that points to left.

Finally, the **reward** in a particular time slot is the **sum of the received transmission power** of all the vehicles who wants to communicate. For a given beam and base station selection, the received transmission power is a positive value if and only if *(i)* the vehicle is covered by the beam, and *(ii)* there is no blockage between the base station and the vehicle. For e.x., in Fig. 5, the black car's received transmission power is zero while the yellow car's received transmission power is a positive value provided there is no blockage. The

beam directions and the beam width (the angular spread of the broad beam) are such that every point of the road is covered by at least one beam. It is possible that for a given beam and base station selection, a base station can communicate with multiple cars at once. For e.x., in Fig. 6, BS2 can communicate with the yellow and the black car (provided that there is no blockage). In this case, the reward will be the sum of the received transmission power of the yellow and the black car.

Section 4: Using the Gym Environment

Suppose your Python script is there in the same folder as the **mmWave_bandits.py** file. Then you can simply import and use the **mmWaveEnv** as follows:

```
import gym

from mmWave_bandits import mmWaveEnv

env = mmWaveEnv()          # env is "the environment" your agent will interact with.

# Your code goes here.

env.close()                # Close the environment after using it.
```

You don't have to understand the code in mmWave_bandits.py. You just have to know how to use a Gym environment. I will release a set of three videos by the end of this weekend about using Gym environments. Also, you can get ample tutorials online. One such tutorial is:

https://www.youtube.com/watch?v=cO5g5qLrLSo&t=581s&ab_channel=NicholasRenotte

You can only call a few attributes and functions of **mmWaveEnv**. The following are the attributes and functions that you can access:

1. env.Horizon. This is the number of time slots H.
2. env.road_width. This is the width of the road w (in meters) as shown in Fig. 2.
3. env.road_length. This is the length of the road l (in meters) as shown in Fig. 2.
4. env.delta. This is the variable δ (in meters) as shown in Fig. 2.
5. env.Nbeams. This is the total number of beam directions.
6. env.Ngps. This is the number of GPS samples per context/observation.
7. env.bs_location. The location of the two base stations.
8. env.observation_space. The is the context space associated with this environment.
9. env.action_space. This is the action space associated with this environment.
10. env.reset(). This function is required to start a new episode.
11. env.step(action). This function takes an action and returns (i) the **context for the next time slot**, (ii) the **reward** associated with the current context and current action, (iii) a value called **terminated** which is NOT USEFUL for this setup, (iv) a value called **truncated** this is **False** when an episode is not over and **True** when an episode is over, and (v) a value called **information** that is NOT USEFUL for this setup.

VERY IMPORTANT: In your code, you must NOT access any other attributes and functions of mmWaveEnv other than the ones mentioned above. This is because in contextual bandits, various components of the environment must remain hidden (because contextual bandit is a learning setup).

Section 5: The Deliverables

1. (5 marks) In every contextual bandits, there must be randomness in the reward associated with an action. Based on your reading of what is discussed above, what are the sources of this noise? The answer to this question must be given in **report.pdf**. The answer not more than half a page.

2. (10 marks) As mentioned in section 3, the number of cars that wants to communicate in a given time slot may vary. Hence, the size of the context vector varies in this setup. Most ML/DL models can't deal with inputs (context is the input) with varying size. How will you address this issue? Justify your answer. The answer to this question must be given in **report.pdf**.

3. (15 marks) In the blank Python script **lin_greedy.py** that is provided to you, implement an RL agent for the **mmWaveEnv** environment that uses **ε-greedy algorithm for linear bandit**. This Python script should also plot a **receding window time-averaged reward** ($window\ size = 100\ time\ slots$) incurred by the RL agent. You MUST NOT include anything in **report.pdf** for this part.
    - What is receding window time averaged reward? Check this small 9 minutes video.
    - The number of episodes you train your RL agent is up to you. The only condition is that the plot of the receding window time averaged reward should "plateau out".
    - You don't have to do any feature engineering. Just use the context directly.
    - You may choose to use a time varying exploration rate. How you vary the exploration rate is completely up to you but it must satisfy basic rules of exploration-exploitation tradeoff.

4. (15 marks) In the blank Python script **lin_ucb.py** that is provided to you, implement an RL agent for the **mmWaveEnv** environment that uses **UCB algorithm for linear bandit**. This Python script should also plot a **receding window time-averaged reward** ($window\ size = 100\ time\ slots$) incurred by the RL agent. You MUST NOT include anything in **report.pdf** for this part.
    - The first three sub-points same as lin_greedy.py.
    - You have to tune a variable ϵ (refer slide 48 of lectures 6 to 9). ϵ can be chosen anywhere between 0.01 to 0.2.

5. (30 marks) In the blank Python script **policy_gradient.py** that is provided to you, implement an RL agent for the **mmWaveEnv** environment that uses **policy gradient algorithm for contextual bandits**. This Python script should also plot a **receding window time-averaged reward** ($window\ size = 100\ time\ slots$) incurred by the RL agent. You MUST NOT include anything in **report.pdf** for this part.
    - The first two sub-points same as lin_greedy.py.
    - You can choose to implement it in Keras or Pytorch. I prefer Keras.
    - During lecture we discussed several variants of the policy gradient pseudocode. All those variants are allowed.
    - The structure of the neural network model is up to you. My advice is to keep it simple.
    - In RL we tend to use a smaller learning rate compared to supervised learning. The smaller the batch size, the smaller should be the learning rate.

6. (5 marks) Make **one plot** that contains the **receding window time-averaged reward** of the three RL agents that you have code till now. Include ONLY this plot in **report.pdf**. Any code that you write to **consolidate** the receding window time-averaged reward for the three RL agents MUST NOT be there in lin_greedy.py, lin_ucb.py, policy_gradient.py, or report.pdf.

7. (20 marks) For contextual bandits, we learned **"value function" based policy** only for linear bandits. However linear bandits is very restrictive for the same reason linear regression is restrictive. In this question, you will develop a very general value function-based policy for contextual bandits (not specifically mmWave vehicular communication).

   We will be referring to slides of lectures 6 to 9 for this question. The context-action value for linear bandits is given by equation (L.1) in slide 29. We then try to learn the weights and bias in order to **minimize the mean squared error** (refer to first equation in slide 37) between the **predicted reward of an action** and the **reward of that action the agent received from the environment**. In general, the context-action value need not be linear with respect to the context. We can use any parametric function of the form $f(x, a; \theta)$, where θ is the parameter to be tuned, to estimate the context-action value. The most obvious choice of $f(x, a; \theta)$ is a neural network whose inputs are the context-action pair, whose output is the estimate of the mean reward corresponding to the input context-action pair, and θ are the weights and biases of the neural network. To tune θ in time t, we find a θ that minimizes the mean squared error,

$$L(\theta) = \frac{1}{t+1}\sum_{k=0}^{t}\left(r_k - f(x_k, a_k; \theta)\right)^2$$

   where x_k, a_k, and r_k are the context, action taken, and reward received at time k. However, this approach to tune θ is **not iterative**, i.e. we need to optimize over the entire history.

   An iterative approach will be to update θ using gradient descent in order to minimize the squared error just for the current time slot t,

$$\hat{L}(\theta) = \left(r_t - f(x_t, a_t; \theta)\right)^2$$

   We can think of this as stochastic gradient descent with batch size 1. Based on this idea, answer the following questions about this general value function-based policy,

   a) (4 marks) Suppose we were to use this approach for mmWave vehicular communication, what will be the number of inputs and outputs of the neural network.

   b) (2 marks) Is training this neural network a regression or a classification problem? Why?

   c) (4 marks) What is the gradient of $\hat{L}(\theta)$ with respect to θ? Use this write the update equation of θ using gradient descent.

   d) (10 marks) Just like ε-greedy algorithm for linear bandits, we can have an ε-greedy algorithm for this general value function-based policy. Write a pseudocode for such a policy. Your pseudocode should demonstrate (very briefly) how you will train the neural network using Keras/Pytorch.