TEAM 29 Details and Contribution Description

1. **Harsha Dayini Akula - SE22UCSE107 :**
    - **Implemented Policy Gradient algorithm (5Q)**
    - **Explored feature engineering and tested various methods to find the optimal approach (2Q) (4Q)**
    - **Answered the theoretical section of the assignment (7Q)**
    - **Plotted graphs for Policy gradient and analysed the graphs of all the algorithms (6Q)**

2. **Harsha Biruduraju - SE22UCSE :**
    - **Implemented Epsilon-Greedy algorithm (3Q)**
    - **Answered the first question in the assignment (1Q)**
    - **Worked on tuning the hyperparameters to gain optimal rewards (4Q)**
    - **Plotted graphs for Epsilon-Greedy performance analysis (6Q)**

3. **Manasvi Boggarapu - SE22UCSE053 :**
    - **Implemented UCB algorithm (4Q)**
    - **Experimented with neural network architectures to improve Policy Gradient's rewards.**
    - **Plotted graphs for UCB performance analysis (6Q)**

1. 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.

Ans: The sources of this noise in the environment ServerAllocationEnv are:

1. **Reward based on Processing Times -** The data center knows the **estimated processing time** for each of the jobs, it does not know the **true processing time.** The same action can take **different amounts of time to complete. Latency** changes with processing time, and latency affects the reward in the episode.
   The same action can cause a **high reward** and a **low reward** (randomness associated with the reward) which can be a source of noise.

2. **Different Number of Jobs –** There can be a different number of jobs arriving for 2 different episodes, and **waiting times can differ** based on the allocated servers, affecting the reward.

   Suppose we consider two episodes where the same action is taken by the agent:

   Episode 1 – 3 Jobs arrive, and 4 servers are allocated. Therefore, waiting time is short resulting in a **higher reward**.
   Episode 2 – 7-8 Jobs arrive, and 1 server is allocated. Waiting time is **too long,** resulting in a **very low reward, causing randomness in the reward.**

3. **Distribution of Priority of Jobs –** The actions taken based on the agent's observations of the environment can have varying rewards due to **distribution of priorities** in the context space. The **more high priority jobs** that are in the **current context,** the higher the probability that **reward is higher**.

4. **Network Usage –** The amount of internet time taken to perform I/O Operations and data transfers. It can influence **waiting times** and **order of job processing.** A mix of high and low network usage jobs can very randomly, introducing **randomness to the reward.**

2 . Based on your reading of sections 4 and 5, find features from the context that will be useful for training RL agent in the subsequent parts. Your answer should address:

• How to deal with the fact that the context size is varying? This is important because most ML/DL models can't deal with varying input size.

• Can we reduce the number of features? This will eventually help you in training agents in the subsequent parts

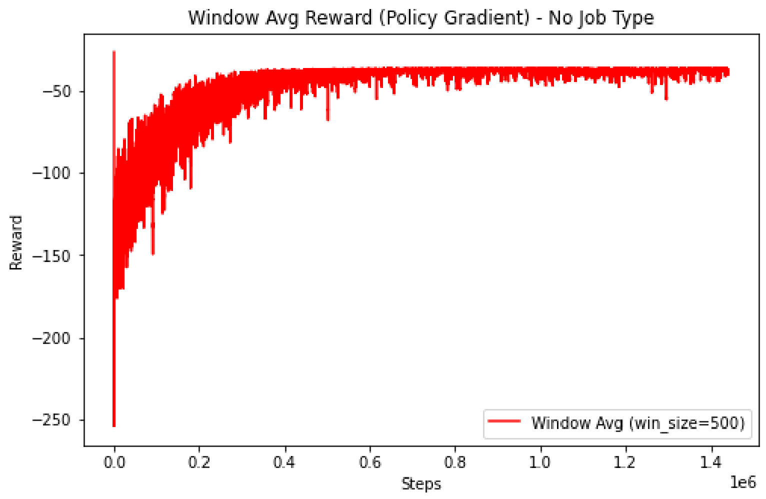
Ans: All four features in the context space are relevant because:

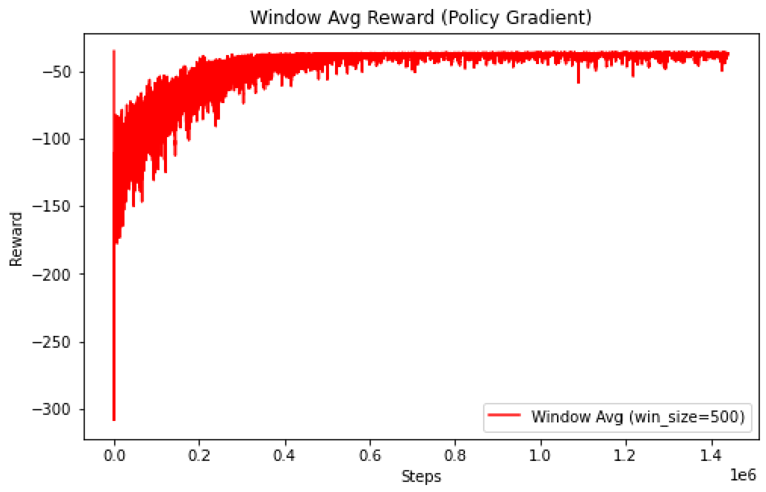
1. **Priority**: The cost function in the environment heavily depends on how high or low the priority is (lower the priority number, higher the penalty if it waits).

2. **Job type**: Different job types may correlate with how well the estimated processing time matches the real processing time, or how the network usage affects latency.

   However, we initially decided to eliminate Job type and check the agent performance. We trained two agents - one with and one without the job type. Our better performing agent, i.e; Job Type included gave a final average reward converging to -37, while the agent with no Job Type gave a final average reward that converged to -47.

   In percentage terms, this is roughly a **27% drop** in performance ((-47) - (-37) = -10; -10 / 37 = 0.27) relative to the higher-performing agent. Hence, losing the job type feature caused the agent to miss critical information about how different job categories impact processing time, leading to suboptimal server allocations and consistently lower rewards.



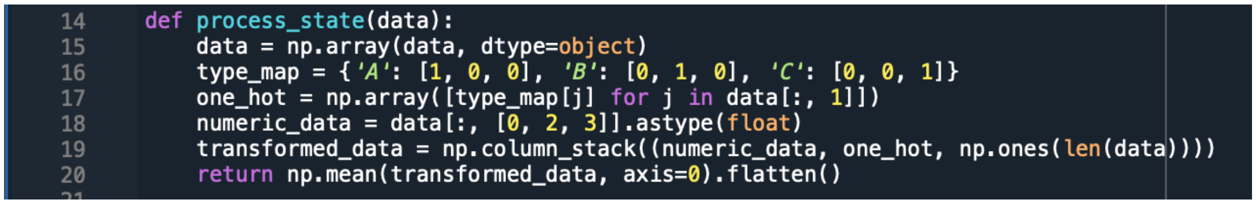


3. **Network usage**: High Network usage might indicate server load. It could affect other tasks and slow them down. This should be considered by the policy.
4. **Estimated processing tim**e: This variable directly affects the number of servers that will be allocated to the job.

Although eliminating lesser-valued features can increase computational efficiency by a considerable margin, after appropriate trial and error, we conclude that this setup should use all the four given features in the context space to achieve optimality.

After analysing the varying context size problem in this setup, we've used this function to overcome and feed a fixed length input to our neural network in Policy Gradient algorithm.

The number of jobs arriving at any time t is a variable.



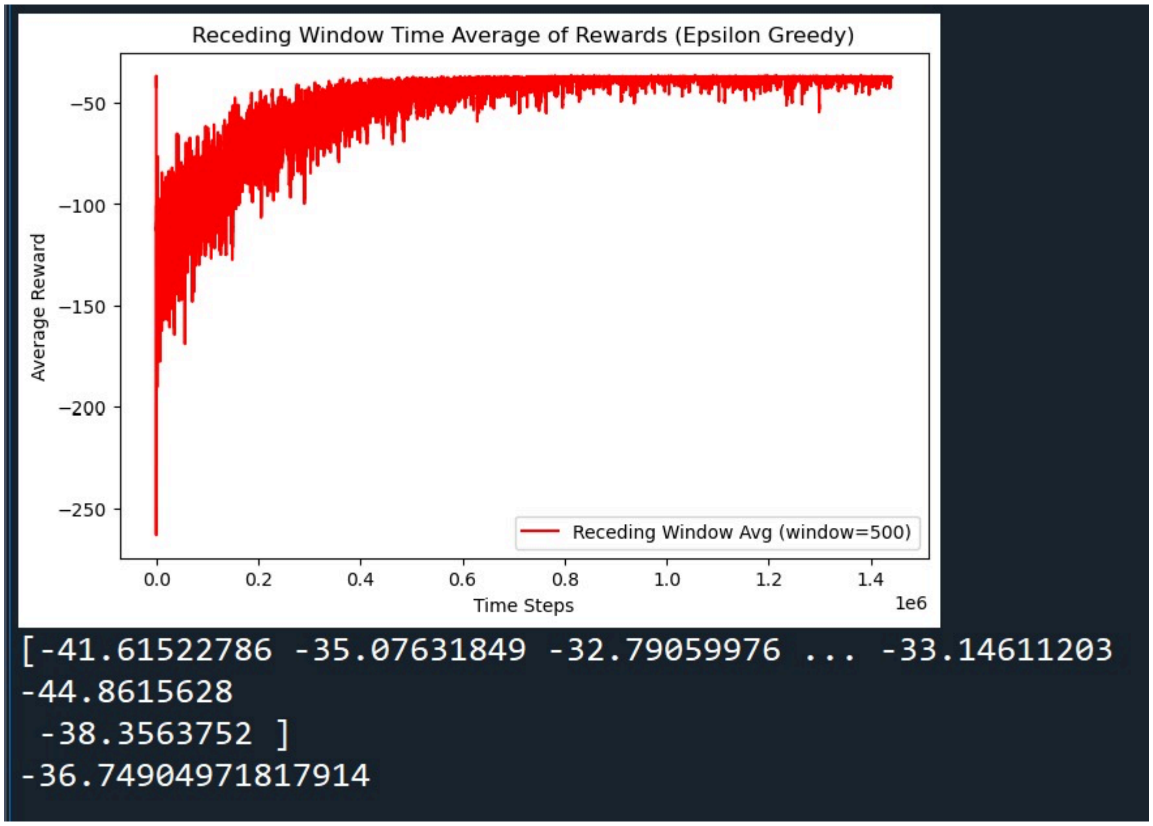
```python
def process_state(data):
    data = np.array(data, dtype=object)
    type_map = {'A': [1, 0, 0], 'B': [0, 1, 0], 'C': [0, 0, 1]}
    one_hot = np.array([type_map[j] for j in data[:, 1]])
    numeric_data = data[:, [0, 2, 3]].astype(float)
    transformed_data = np.column_stack((numeric_data, one_hot, np.ones(len(data))))
    return np.mean(transformed_data, axis=0).flatten()
```

We first converted the list of jobs into a NumPy array for easier numerical operations. The Job type variable was **One-Hot encoded** to retain the categorical value and to prevent unintentional weight assignments.

A bias term (column of ones) was added (theory/advice taken from articles and ai tools) to help in better generalisation. (Adding a bias allows the model to learn better offsets). Finally, mean across all jobs was computed, reducing variable-length input to a fixed-size feature vector.

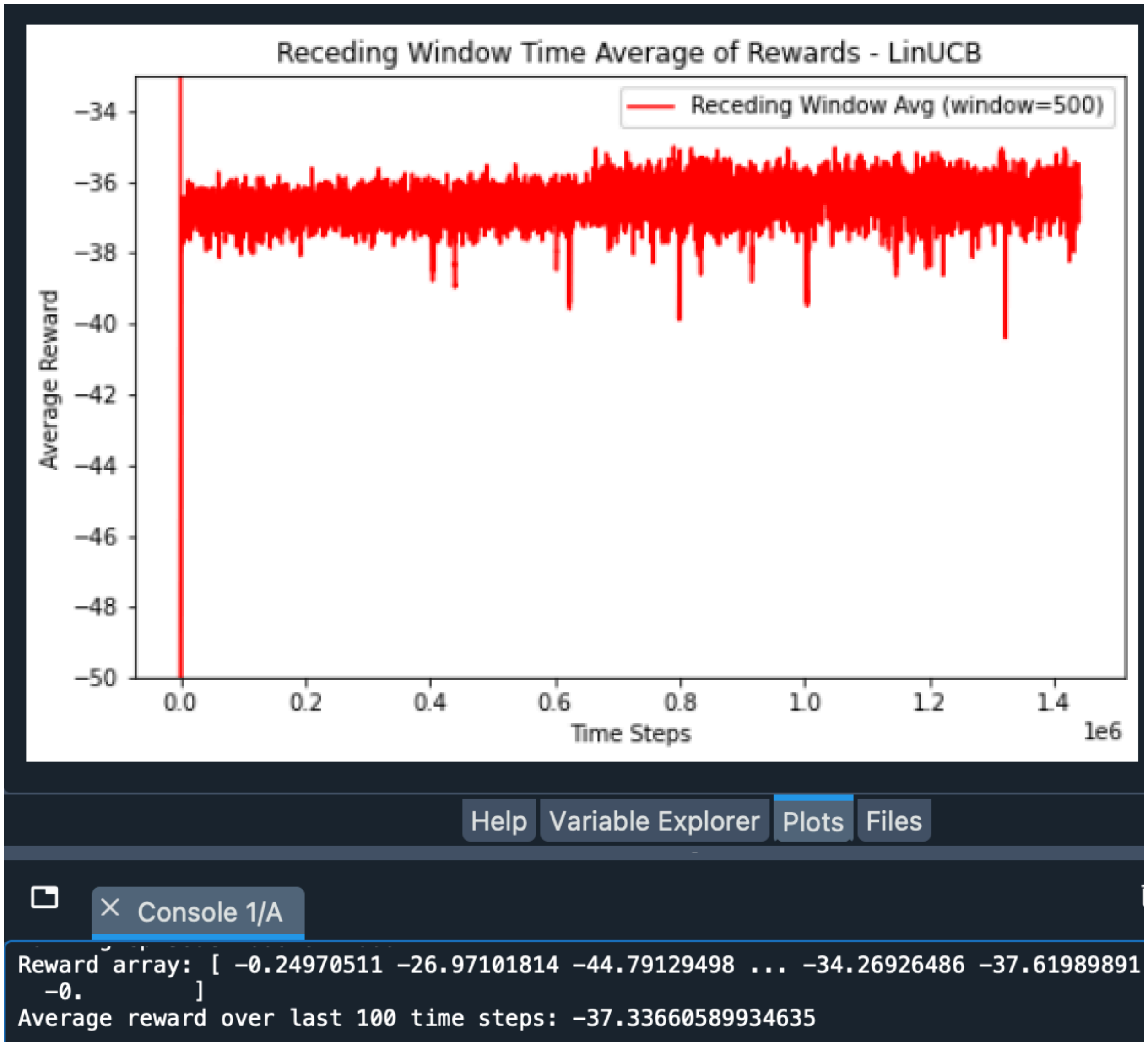
This way, we were able to retain the information pertaining to all the jobs arrived at any given instance and also pass a consistent length input.

3. ε -greedy algorithm



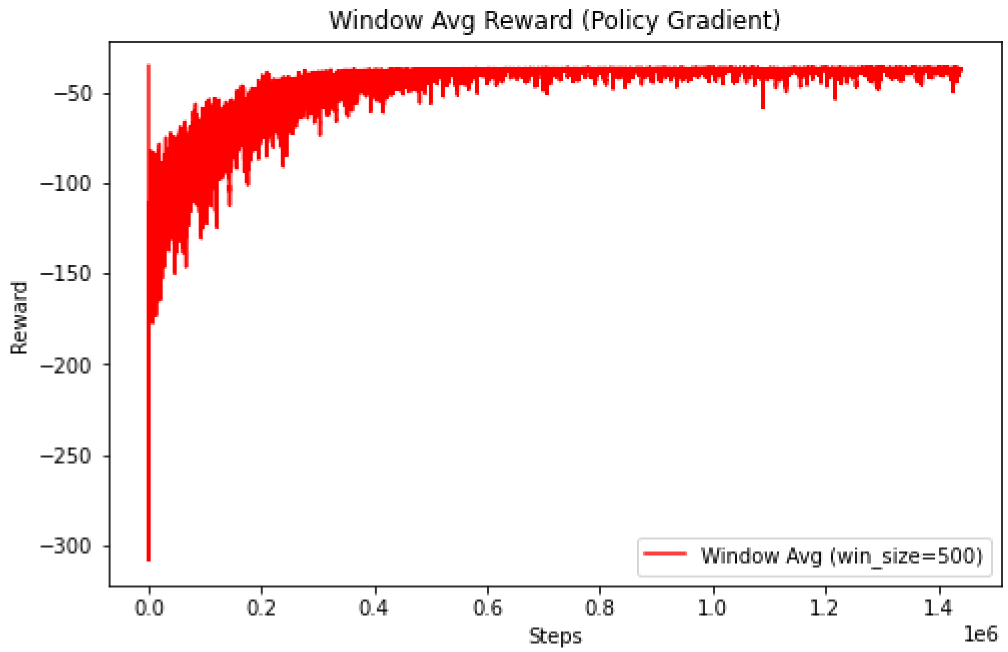
```
[-41.61522786 -35.07631849 -32.79059976 ... -33.14611203
-44.8615628
 -38.3563752 ]
-36.74904971817914
```

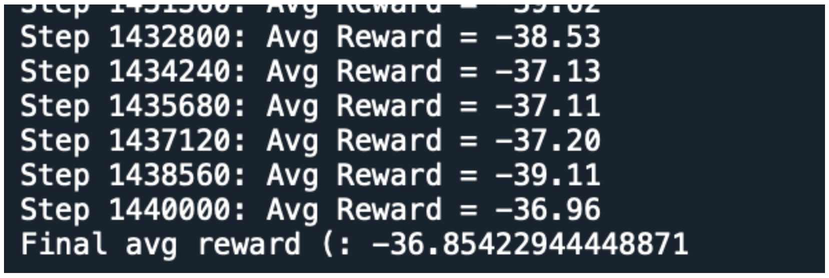
4. UCB algorithm


Receding Window Time Average of Rewards - LinUCB
Receding Window Avg (window=500)
Average Reward
Time Steps
1e6
Help
Variable Explorer
Plots
Files
Console 1/A

```
Reward array: [ -0.24970511 -26.97101814 -44.79129498 ... -34.26926486 -37.61989891
  -0.         ]
Average reward over last 100 time steps: -37.33660589934635
```

5. Policy Gradient Algorithm

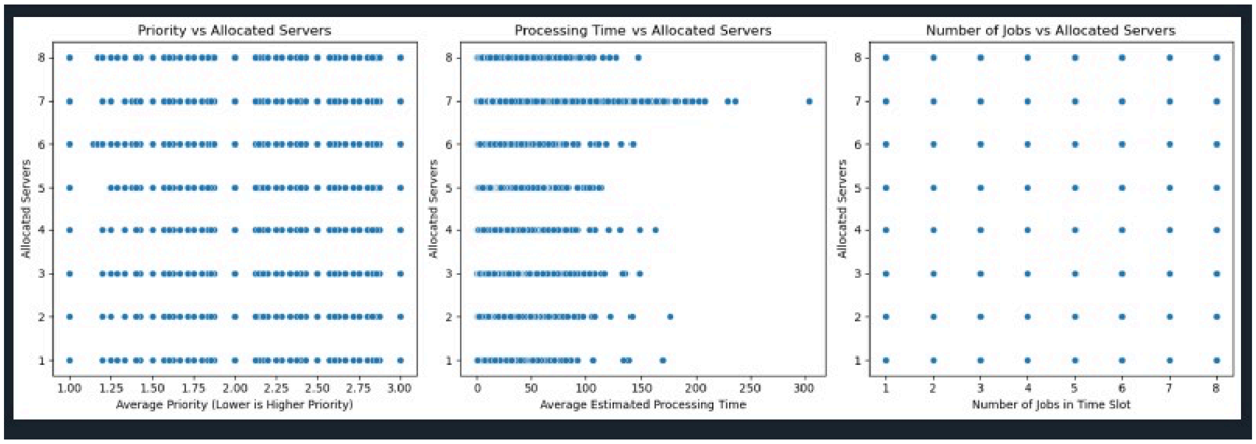




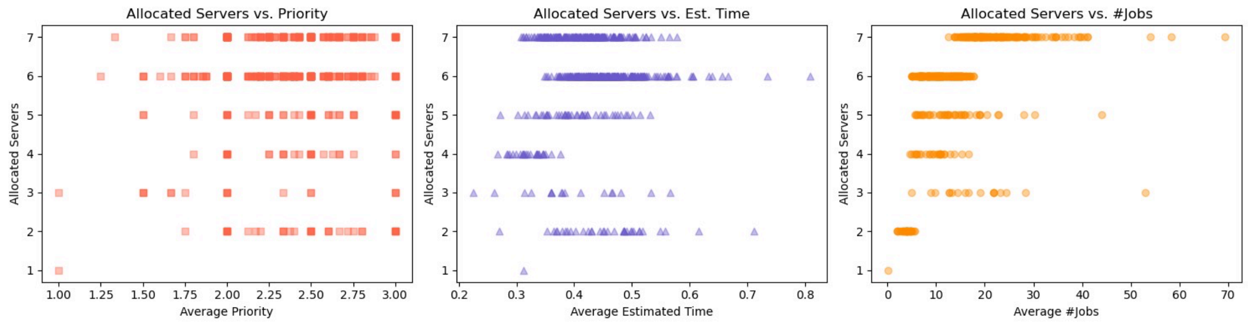
```
Step 1432800: Avg Reward = −38.53
Step 1434240: Avg Reward = −37.13
Step 1435680: Avg Reward = −37.11
Step 1437120: Avg Reward = −37.20
Step 1438560: Avg Reward = −39.11
Step 1440000: Avg Reward = −36.96
Final avg reward (: −36.85422944448871
```

6.

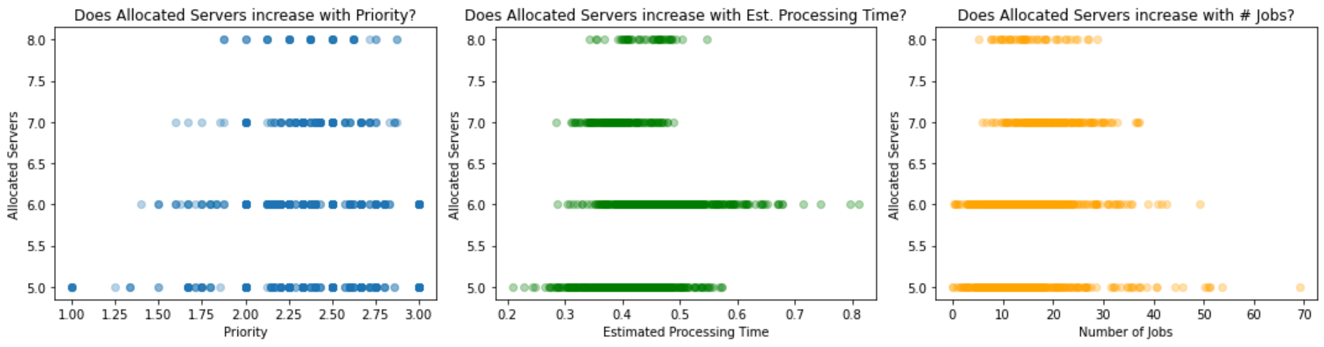
1. Epsilon-Greedy



2. UCB



3. Policy Gradient



Analysis (Policy Gradient) :

- In the Servers allocated Vs Priority graph, we plotted the priority values on the x-axis and the number of servers on the y-axis. We observe most of the allocated servers are are placed in the range (5-8). We realise that the policy rarely chooses extreme number of servers (1-2 or 8-9).
- We tried understanding why this could happen. One of the probable reasons is that, a policy can converge to a “safe” action space if it constantly yields acceptable rewards. This can be perceived as local minima/ saddle point in terms of gradient ascent.
- Continuing with the analysis, the Allocated Servers vs. Estimated Processing Time and Allocated Servers vs. Number of Jobs plots show essentially the same behavior as the Allocated Servers vs. Priority plot: the policy mostly chooses a mid-range allocation This suggests the agent has converged to a relatively “safe” action space that provides acceptable rewards across different contexts, rather than aggressively scaling servers up or down.
- There is minimal penalty (or gain) for choosing near-constant allocations, making a wide range of actions unnecessary from the agent’s perspective.

Summary :

Across all three methods, we see a pattern of “good enough” mid-range server allocations rather than a dramatic scaling based on context. It’s not that the algorithms *never* choose higher or lower servers—just that the reward signal or exploration schedules apparently don’t provide a big push to explore those extremes or heavily adapt to priority/time/jobs.

In other words, all three approaches converge to fairly safe (and presumably reward-efficient) policies. If stronger scaling is desired—for example, allocating many more servers for high-priority, long-duration, or large-job contexts—then one might consider:

- Adjusting the reward function to reward context-aware decisions more strongly,
- Increasing exploration (longer training or slower decay), or
- Tweaking hyperparameters so the agent more boldly differentiates among contexts.

Overall, the nine plots confirm that while each method has a slightly different learning mechanism, they all end up with a stable but relatively conservative server-allocation policy.

7. (a).

Python code to sample an action from Gaussian distribution with mean = theta1 and standard deviation = exp(theta2) :

```
import numpy as np

def sample_action(theta1, theta2, x):

    mean = np.dot(theta1, x)

    std = np.exp(np.dot(theta2, x))

    return np.random.normal(mean, std)
```

7. (b).

__/__/__

step 2:

7b. Gradient estimate (slide no. 85):

$$\nabla_\theta J \approx \frac{1}{N}\sum_{t=1}^{N} r_t \nabla_\theta \log \pi(a_t|x_t,\theta)$$

We have a gaussian distribution of mean $\theta_1^T x$ and standard deviation $\exp(\theta_2^T x)$. We plug this in place of $\pi(a_t|\theta, x_t)$

$$\log \pi(a_t|x_t,\theta) = -\log(\sigma\sqrt{2\pi}) - \frac{(a-\mu)^2}{2\sigma^2}$$

Simplifying: $$\log \pi(a|x,\theta) = -\theta_2^T x - \frac{(a-\theta_1^T x)^2}{2\exp(2\theta_2^T x)} + C \quad \text{(1)}$$

step 3: We will perform partial differentiation on (1) wrt θ_1

$$\frac{\partial \log \pi}{\partial \theta_1} = \frac{a-\mu}{\sigma^2}\cdot x \qquad \text{where } \mu = \theta_1^T x \text{ and } \sigma = \exp(\theta_2^T x)$$

$$\nabla_{\theta_1} J = \frac{1}{N}\cdot\sum_{t=1}^{N} r_t \cdot \frac{(a_t-\theta_1^T x_t)}{\exp(2\theta_2^T x_t)}\cdot x_t \quad \text{(2)}$$

step 4: We will perform partial differentiation on (1) wrt θ_2 and estimate $\nabla_{\theta_2} J$

$$\frac{\partial \log \pi}{\partial \theta_2} = \left(\frac{(a-\mu)^2}{\sigma^2} - 1\right)\cdot x$$

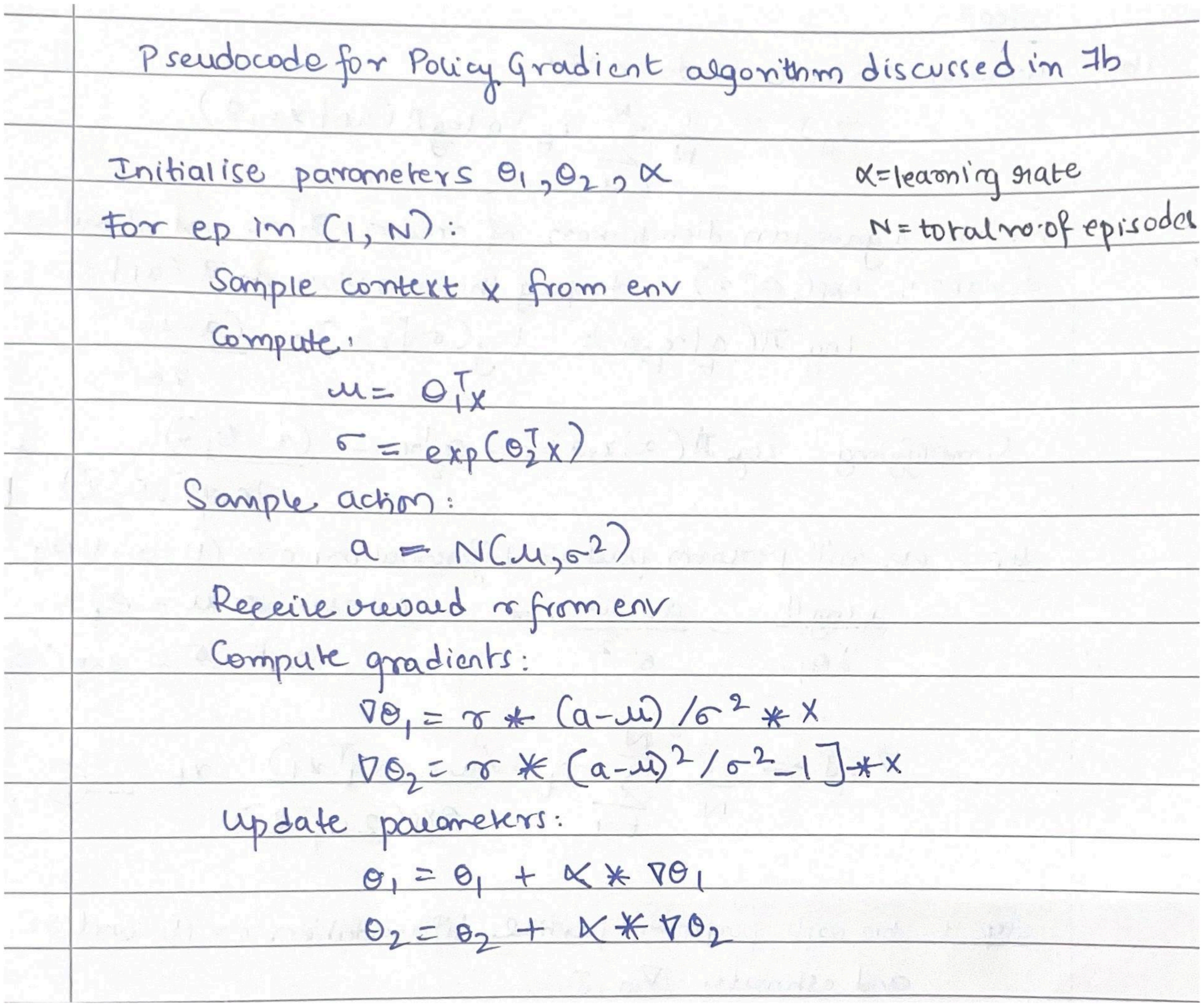
$$\nabla_{\theta_2} J = \frac{1}{N}\sum_{t=1}^{N} r_t \cdot \left[\frac{(a_t-\theta_1^T x_t)^2}{\exp(2\theta_2^T x_t)} - 1\right]\cdot x_t \quad \text{(3)}$$

step 5: We will use equations (3) and (4) to update θ_1 and θ_2 using gradient ascent. (Consider α as the learning rate)

$$\theta_1 \leftarrow \theta_1 + \alpha\cdot\frac{1}{N}\sum_{t=1}^{N} r_t \left(\frac{a_t - \theta_1^T x_t}{\exp(2\theta_2^T x_t)}\right)\cdot x_t.$$

$$\theta_2 \leftarrow \theta_2 + \alpha + \frac{1}{N}\sum_{t=1}^{N} r_t \left[\frac{(a_t-\theta_1^T x_t)^2}{\exp(2\theta_2^T x_t)} - 1\right]\cdot x_t$$

7. (c).



Pseudocode for Policy Gradient algorithm discussed in 7b

```
Initialise parameters $\theta_1, \theta_2, \alpha$                    $\alpha$=learning rate
For ep in (1, N):                                                      N = total no. of episodes
        Sample context x from env
        Compute:
                $\mu = \theta_1^T x$
                $\sigma = \exp(\theta_2^T x)$
      Sample action:
                $a = N(\mu, \sigma^2)$
        Receive reward r from env
        Compute gradients:
                $\nabla\theta_1 = r * (a-\mu) / \sigma^2 * x$
                $\nabla\theta_2 = r * (a-\mu)^2 / \sigma^2 - 1] * x$
        Update parameters:
                $\theta_1 = \theta_1 + \alpha * \nabla\theta_1$
                $\theta_2 = \theta_2 + \alpha * \nabla\theta_2$
```