90

ECE597AA Project assignment

Bradford Gill, Robin Dehler, Siddhanta Shrestha

Link to Github: https://github.com/electricdarb/597AA.git

Problem 1

Assume the network provider aims to maximize the immediate reward defined in [equation (16), ref2]. Formulate the network provider's resource allocation problem as a centralized optimization problem. This problem includes the allocation of radio, computing and storage resources to network slices (by the network provider) to meet the slice requests from tenants.

Answer:
The optimization problem needs to be designed in a way such that the revenue of the InP is maximized. We do this in the following equation:

$$\max_{n_c} r(s, a_s) = \begin{cases} r_c & \text{if} \quad e_c = 1, a_s = 1, s' \in S \\ o & \text{otherwise} \end{cases}$$

$$\text{s.t. } \Theta, \Omega \text{ and } \Delta \text{ (the maximum available fog and InP resources)}$$

The dependency of the maximization to radio (ϴ), computing (Ω) and storage resources (Δ) means, that the algorithm for solving the optimization problem has to check if the next state (s') is a valid state, which basically means that it needs to check whether the accepting of a slice request with a_s = 1 exceeds the available resources (InP + fog). To ensure that allocated resources do not exceed the available resources, the following constraints are placed on ϴ, Ω, Δ where number of slices (n_c) = 3:

- Maximum radio $(\theta) \geq \sum_{c=1}^{C} r_c^{re} n_c$
  - $\theta = \theta_{\text{cloud}} + \theta_{\text{fog}} \geq \sum_{c=1}^{C} r_c^{re} n_c$
- Maximum computing $(\omega) \geq \sum_{c=1}^{C} \omega_c^{re} n_c$
  - $\omega = \omega_{\text{cloud}} + \omega_{\text{fog}} \geq \sum_{c=1}^{C} \omega_c^{re} n_c$
- Maximum storage resources $(\Delta) \geq \sum_{c=1}^{C} \delta_c^{re} n_c$
  - $\Delta = \Delta_{\text{cloud}} + \Delta_{\text{fog}} \geq \sum_{c=1}^{C} \delta_c^{re} n_c$

Additionally, the utility = reward (r_c) - cost, where the cost is included to make the utility concave. As such, the eMMB slices would have a higher number of fog nodes and URLLC would have the least amount of fog nodes. In the event that the arrival of the slice is rejected or if no slice request has arrived at the system, then the immediate reward is 0. If the slice is accepted then the system moves to the next state (s') and the network provider reciences an immediate reward rc, where its value is the amount of money paid by the tenant based on resources and additional services required.

Maximize $r(s, a_s)$ - $c(s, a_s)$

Subject to

- Maximum radio $(\theta) \geq \sum_{c=1}^{C} r_c^{re} n_c$
    - $\theta = \theta_{\text{cloud}} + \theta_{\text{fog}} \geq \sum_{c=1}^{C} r_c^{re} n_c$
- Maximum computing $(\omega) \geq \sum_{c=1}^{C} \omega_c^{re} n_c$
    - $\omega = \omega_{\text{cloud}} + \omega_{\text{fog}} \geq \sum_{c=1}^{C} \omega_c^{re} n_c$
- Maximum storage resources $(\Delta) \geq \sum_{c=1}^{C} \delta_c^{re} n_c$
    - $\Delta = \Delta_{\text{cloud}} + \Delta_{\text{fog}} \geq \sum_{c=1}^{C} \delta_c^{re} n_c$

Problem 2

Solve the optimization problem and represent the number of requests versus the optimum reward for different numbers of available fog nodes. You can use Matlab and the simulation parameters described in [ref2].

Answer:
The code that we used to solve this problem is provided in file **prob2.py**.
The maximum number of resources that can be allocated to the InP and each fog node is only dependent on a bottleneck resource that is fully exhausted before the other two. Therefore, we chose to not compare the 3 different resources (radio, computation and storage resources), but to set a maximum number of requests that can be allocated by the InP and the fog nodes, respectively.
For the maximum number of requests that can be allocated to the InP we chose the number inp_res = 4. For the slices we chose different available resources for each slice

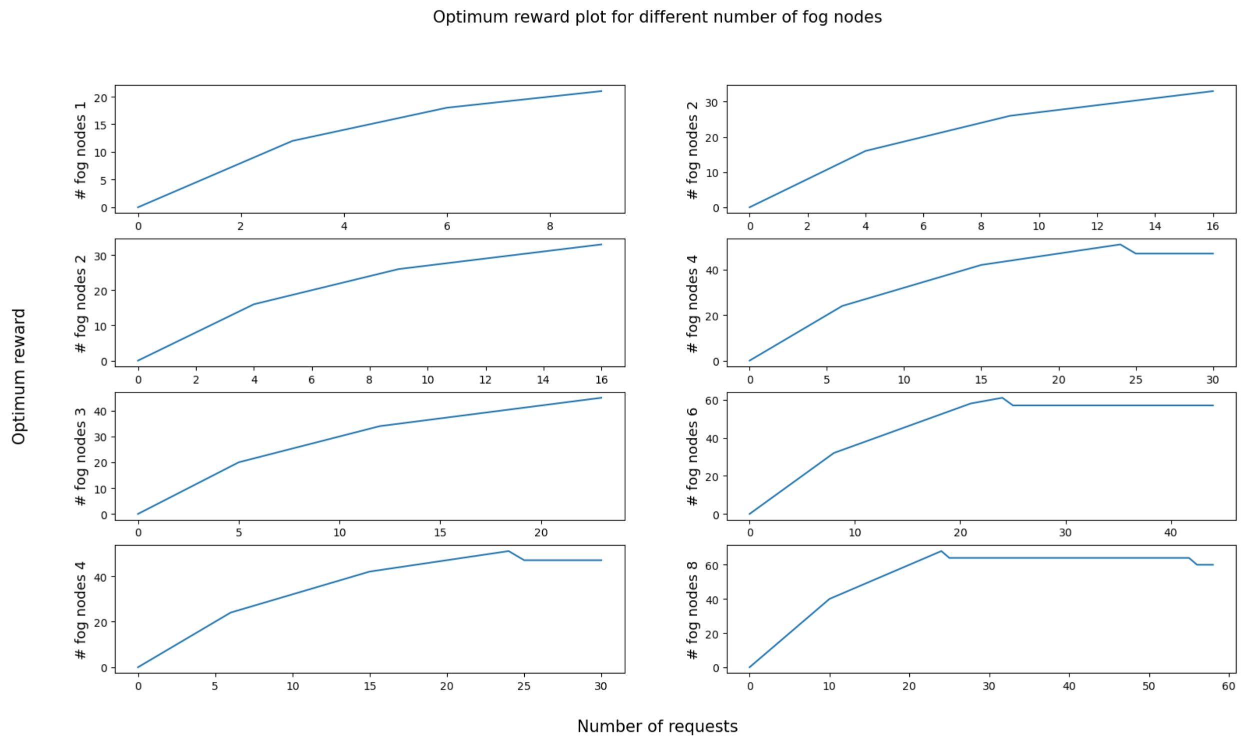
- *Inp_res + const * num_nodes * node_res*

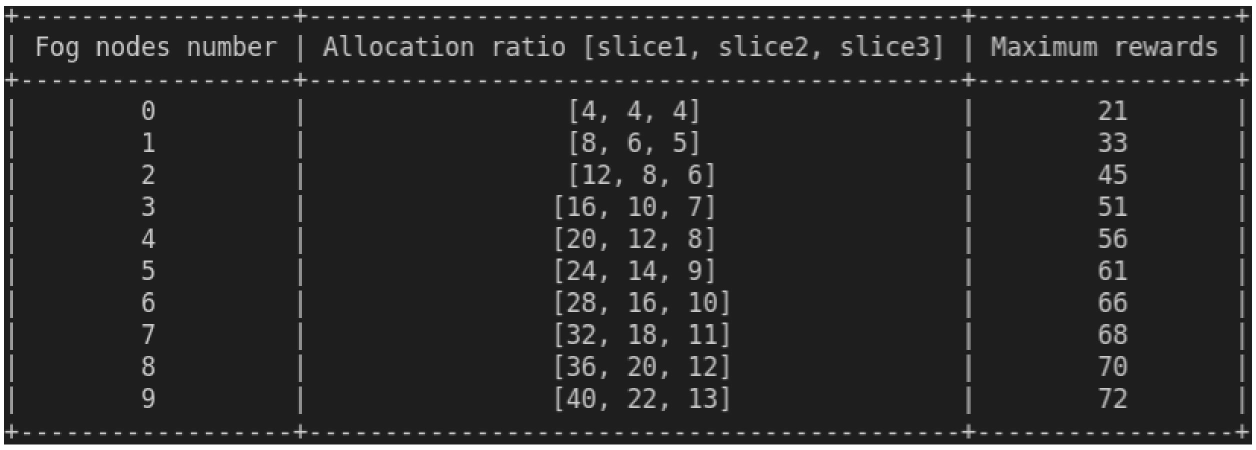
- Maximum number of request (Inp_res) is set to 4
- Number of nodes (num_nodes) varies from 1 to 8t
- Resources per fog node ($node\_res$) is set to 1
- Immediate rewards ($const$) varies between 4, 2 and 1 for slice-1 (eMMB), slice-2 (mMTC) and slice-3 (URLLC), respectively.

The system gets really slow when allocating too many fog resources. Therefore we limit the maximum number of fog nodes to 8.
In **prob2.py** we store the maximum reward and the corresponding allocation for each different num_nodes and plot the result. The plot can be seen in the following figure:

Number of Fog Nodes vs Maximum Reward

The respective allocation ratio for slice-1, slice-2 and slice-3 looks as follows:





```
+------------------+----------------------------------------+------------------+
| Fog nodes number | Allocation ratio [slice1, slice2, slice3] | Maximum rewards |
+------------------+----------------------------------------+------------------+
|        0         |               [4, 4, 4]                |        21        |
|        1         |               [8, 6, 5]                |        33        |
|        2         |               [12, 8, 6]               |        45        |
|        3         |              [16, 10, 7]               |        51        |
|        4         |              [20, 12, 8]               |        56        |
|        5         |              [24, 14, 9]               |        61        |
|        6         |              [28, 16, 10]              |        66        |
|        7         |              [32, 18, 11]              |        68        |
|        8         |              [36, 20, 12]              |        70        |
|        9         |              [40, 22, 13]              |        72        |
+------------------+----------------------------------------+------------------+
```

Problem 3

Implement the Q-learning algorithm described in [Section IV, ref2] for the enhanced network slicing model. You can use Matlab or Tensorflow and the simulation parameters described in [ref2]. Compare the performance with the optimum solution obtained in question 2.

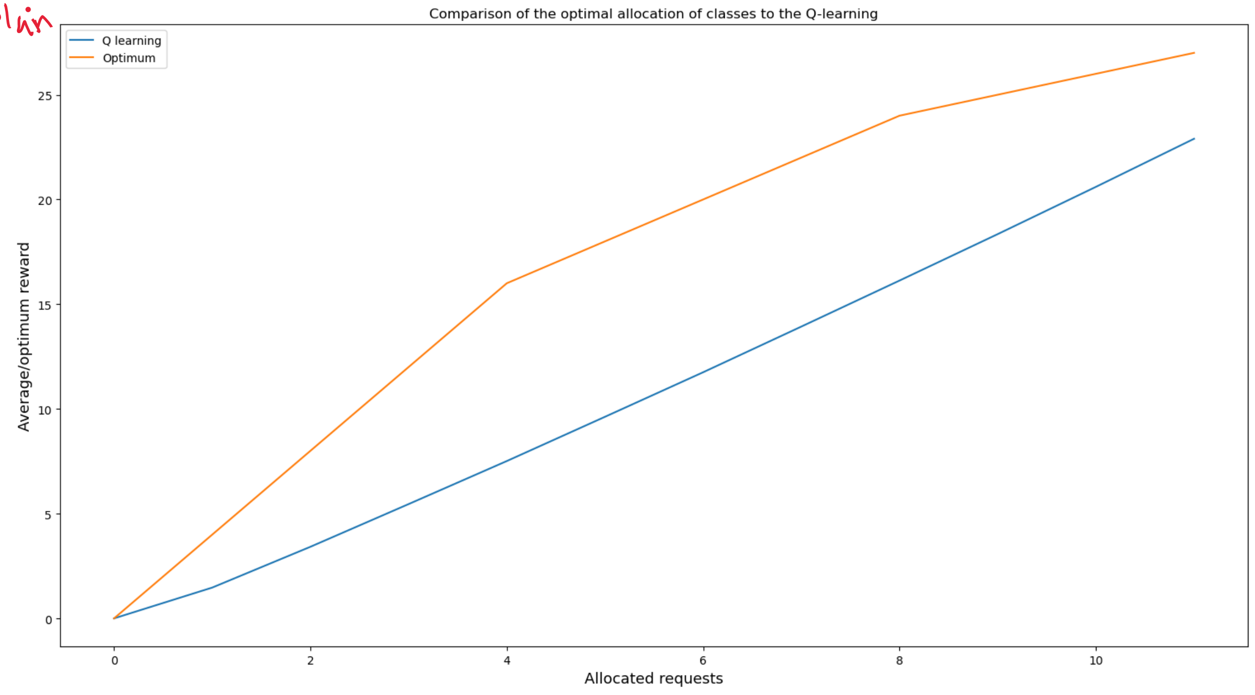
Answer:
In order to implement the Q-learning algorithm we first need to define the Q-table for the problem. For that we created the class *Qtable* in the file **Qtable.py**. The dimensions of the Q-table are *[num_classes, num_resources, num_actions]*. Each cell of the table stores a Q-value for the given state and the given action. Several functions in the class make it easier to access and change the Q-table. We experimented with having the state represent the amount of resources used in each class, however, we found this to be less effective. A convenient thing of creating the Q-table as a class is that it can be easily translated to use a neural network rather than a static table.

State transition model: To model when events would happen each class is given a uniform probability of happening in a given timestep. If the resource is allocated, a release time is formed using an exponential distribution. The allocated resources are stored in a buffer alongside the release time, when the release time comes the allocated resources are removed from the state and the values are popped from the buffer. We chose an exponential distribution because it was used in the paper and allows the network to have a diverse set of random resource requirement lengths. In our case the probability of slice 0, slice 1, and slice 2 having an event in a given time step was .3, .2, and .25 respectively. Each time step corresponds to 1.5 minutes or 1/40th of an hour. The fog resources were lumped together with InP for simplicity, however, this could be changed easily by including more resource types and having a third "assign to fog resource" action. This would increase computing complexity. In these simulations each request used 2 CPUs, 1GB of storage, and 500 Mbps of radio. The System has a total of 10 CPUs, 5 GBs, and 500 Mbps.

To compare the optimum allocations with the performance of the Q table, we adjusted the values for the optimized allocation in file **prob2_forcomp.py** and store the optimum allocation for each number of requests in *optimum.npy*. In **Qtable.py**, we store the Q-table as *qtable.npy* and use it in file **prob3_comp.py** to compare the performance of the Q-learning algorithm with the loaded optimum allocation. To be able to compare the Q-learning's performance to the Double Deep and the optimum allocation in the following problem we also store the average rewards for the Q-learning as *avg_reward_prob3.npy*. The resulting plot comparing Q-learning to the optimum allocation is seen in the next figure.

For the comparison of the convergence of the Q-learning algorithm to the convergence of the Double Deep Q-learning algorithm, we stored the intermediate rewards while training in file *qtable_convergence.npy*. This one is later used in problem 5.

-5 Explain



Problem 4

Implement the Double Deep Q-learning algorithm described in [Section V.A, ref2] for the enhanced network slicing model. You can use Matlab or Tensorflow and the simulation parameters described in [ref2]. Compare the performance with the optimum solution obtained in question 2 and Q-learning in question 3.

Answer:
As described in *[ref2]*, the Double Deep Q-Learning is used to address the slow-convergence problem of Q-learning. To stay as close as possible to *[ref2]*, we implemented the Deep Double Q-learning algorithm as it is described in *Fig.3*, *Fig. 4* and *Algorithm 1* of *[ref2]*. The code in which we implemented the adjusted Q-table, the Double Deep neural network and *Algorithm 1* with the networks seen in *Fig.3* is **DoubleDeepQ.py**.

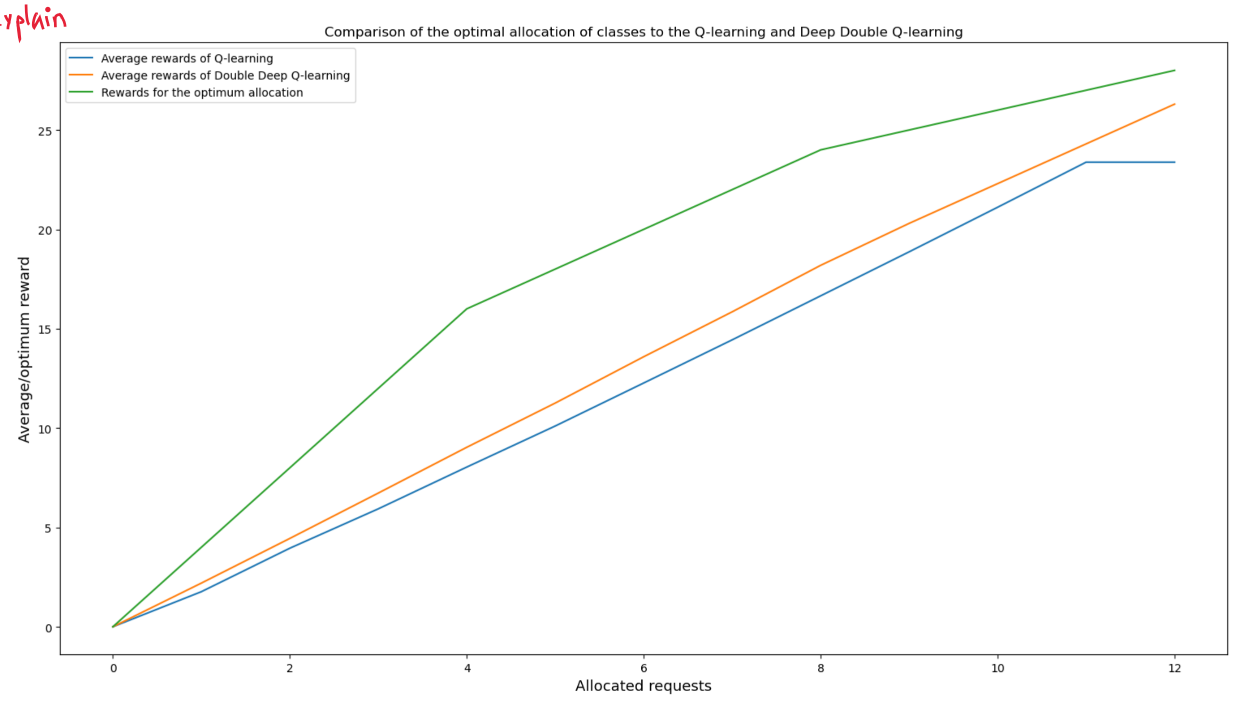
The deep learning network took 4 inputs: storage, compute, radio, and slice number. The Slice number was one hot encoded to allow the network to easily understand it and the resources were normalized to between 0 and 1. There are three dense layers, the two hidden layers have 5 outputs and a relu activation and the final layer has a softmax activation with an output for actions 0 and 1.

Note: the accuracy of the model seemed to significantly depend on the initialization of the weights. You will most likely reproduce similar but not identical results.

We stored the results for the comparison of the network's convergence as *rewards_list_DD.npy* and the results for the comparison of the network's performance as *avg_reward.npy*.

We compare the results of the performance of the Double Deep Q-learning to the Q-learning and the optimal allocation in file **prob4_comp.py** which creates the following figure.

−5 explain



Problem 5

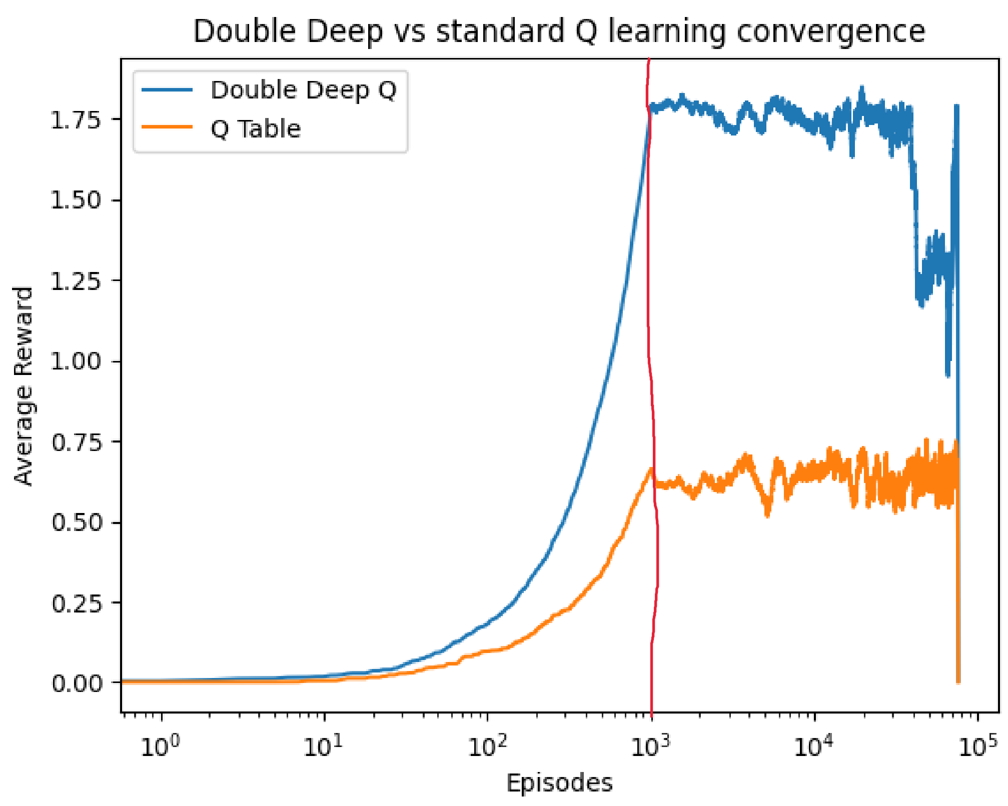
Illustrate the convergence of the Q-learning and Double Deep Q-learning algorithms developed in questions 3 and 4, respectively.

Answer:

−5

This problem is showing the faster convergence of Double Deep Q-learning compared to standard Q-learning.

As you can see the deep learning converges in fewer iterations, however, each iteration takes much longer for the double deep method.


Double Deep vs standard Q learning convergence
Double Deep Q
Q Table
Average Reward
Episodes
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
10^0
10^1
10^2
10^3
10^4
10^5

?