Solution on Data Center Job Scheduling Problem

Linsen Dong
Jan 21, 2018
Github project page: https://github.com/Lukeeeeee/DataCenterJobSchedulingSolution

I. INTRODUCTION

1. With improvement on modern data center and supercomputing center, the algorithm on allocation job of data center, also known as data center job scheduling problem (DCJSP) to optimize energy efficiency and throughout, also control the temperature within the center became a critical part. Our method, by incorporating state-of-art deep reinforcement learning (DRL), can solve this control problem without adding any fluid mechanical and physical knowledge as domain knowledge, only by using historical trace on server utilization/temperature and job execution detail trace. In the following section, we first give the formulation of this problem as well as transforming it into Markov Decision Process (MDP) so it can be solved by DRL. In the third section, we introduce our method, which is based on the Deep Q-Learning (DQN) method. In the fourth section, we give the experiments and results. At last section, we concluded our method, and give directions for further improvements.

II. PROBLEM FORMULATION

1. Definition on DCJSP

   a. Server Entity

Server is the computing resources in data center, it handles the job that required to be done. In data center, there are usually many servers which placed on some racks. However, for the simplicity of formulation, we define the entity of server as a square, and all of them will shape into a grid.

One server entity will be denoted by a tuple $Server = (R, x, y, C_t, U_t)$, the R denote the computing resources (like the number of CPU/GPU) this server have. For simplicity of our method, the computing resources of each server is converting into a scalar without changing the original problem's formulation, and all servers are homogeneous, so the R is same for all servers. (x, y) denotes the position of this server in the grid, since different position in data center would influence thermal condition.

Also, we define the energy each server costs at each time step with *eq 1* due to that there is a linear relation between it utilization and its energy consumption. The energy cost by server i at time step t denoted by C_t^i, the usage at time step t is denoted by U_t^i,, which means how many computing units was being used at this time.

$$C_t^i = a \cdot U_t^i, + b(b \geq 0) \quad \textit{Eq 1}$$

b means the idle consumption of server and the total energy it costs denoted by the sum over time step:

$$C^{i} = \sum_{k=1}^{T} C_{k}^{i}, \quad Eq\ 2$$

b. Job Entity

Job is the request given by users, and it is denoted by a tuple $Job = (T_{submitted}, R, T_{process}, T_{deadline})$

$T_{submitted}$: the time when this job was submitted to the data center.

$T_{deadline}$: the time the job should be done before.

R : the resources the job reqiured.

$T_{process}$: the total time needed to be processed.

c. Thermal Condition

For simplicity, thermal condition is only influenced by the servers which generated the heat. We do not consider the cooling system or any physical condition in the data center since we only focus on the scheduling system. And the heat generated by server i at time step t, $Heat_t^{server_i}$ will be decided by its energy cost C at time step t.

$$Heat_t^{server_i} = G(C_t^{server_i}) \quad Eq\ 3$$

The overrall thermal condition in data center is directly decided by the overrall heat of each server and the heat that was lost. To simplify problem, the heat that was loss at each time was a fixed ration to the current total heat in data center which means the heat at time step t, $Heat_t^{i}$ can be modeled as the following abstract function(since we don't have the true model of thermodynamics):

$$Heat_t^{all} = F(Heat_t^1 \ldots Heat_t^n, Heat_{t-1}^{all}) \qquad Eq\ 4$$

n is the total number of server

d. Objective

Our problem foucs on maximzing the data center energy efficiency, as well as maintaning the data center good thermal condition. So the objective function is consist of two parts.

Firstly, we will define the data center energy efficiency. For each server, we use the proportion of its usage U_t to its energy cost C_t at each time step as the energy efficiency $E_t = \frac{U_t}{Ct}$. For thermal condtion, we consider maximizing heat change $Heat_{all}^{t-1} - Heat_{all}^{t}$ as the objective.

Since we will use reinforcement learning, these objective will be transformed into reward function with a clear definition in next section.

2. Transformation into Markov Decision Process (MDP)

   a. Agent based formulation

      To solve a decision-making problem, we usually used agent-based model in *Figure 1*

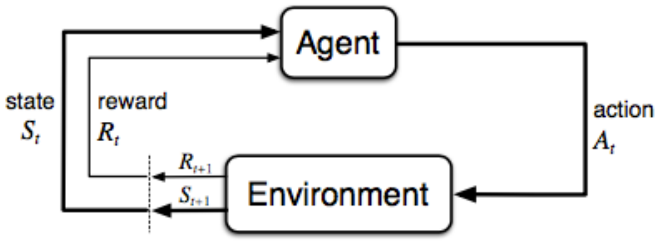

Figure 1 [Richard S. Sutton etc. 2017]

      In this problem, the agent is the scheduling system that assign each new coming job to a certain server. The environment we care about is related to server's state like usage, energy cost, also the thermal condition like heat or temperature in data center. For simplicity, we assume that each time step, there will be one new job submitted and the agent make a decision.

   b. State

      $State_t$ at each time is the input of the agent. It should consist the information of all servers' current state, new job's detail and overall thermal condition.

$$State_t = (Server_t^{1..n}, job_t, heat_t^{all})$$

   c. Policy

      Since DQN is a value-based control method, the policy will be $greedy$ or $\epsilon - greedy$

   d. Action

      $Action_t$ of the agent is assigning the new job to a certain server that will be processed instantly or in the waiting queue of this server according its policy

   e. State Transition

      Since we will use historical data instead of simulation platform, we will not cover this part.

   f. Reward Function

As stated before, we will give the definition of reward at time t, with $State_t$, $Action_t$ as follows:

$$Reward_t(S_t, A_t) = \alpha \sum_{server} E_t^{server} + \beta(Heat_{all}^t - Heat_{all}^{t+1}) \quad \textit{Eq 5}$$

α, β is the scaling factor

III. APPROACH

1. Incorporating Deep Q Learning (DQN) with modification

In this part, we will cover how to use DQN to solve this problem. Since the action is discrete, there is no need to use any policy gradient methods. Also, considering no real environments or simulation were involved and other real-world situation, we have to made some modification to original DQN method instead of using an end-to-end system.

a. Design of Q Network

As the common design in DQN, we will design our Q network as a mapping from State to all actions' Q value and using policy like $greedy, \epsilon - greedy$ to choose servers.

The Q network consists of two main parts, first part is a feature extractions network $\phi(Server_t^i)$ for all servers and this network is identical to every server. The reason for using identical network that the state defined in our MPD section consist many servers' information, and as our assumption, the servers are homogeneous so it is natural for all servers to share a same feature extraction network.

All features of servers will concatenate with the state of jobs and thermal condition to a fully-connected network, the output size is equal to the number of servers, and each output unit is the Q value of its corresponding action.

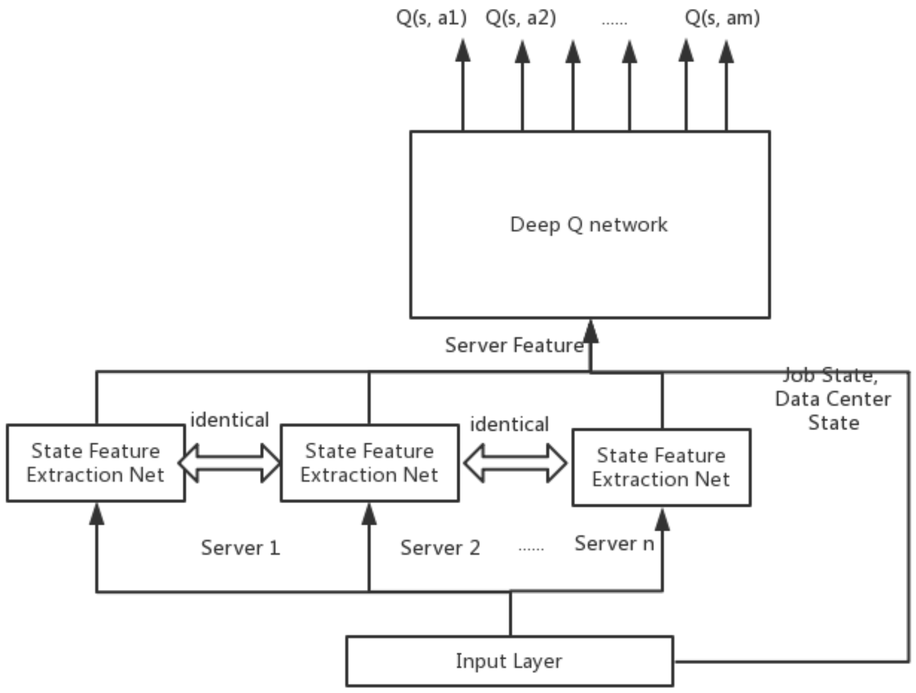

Figure 2. Structure of DQN

b. Modification to DQN

1. Handle Invalid Action

   Every time step, we will compute the Q value for all actions pair with current state, so we have to handle the invalid action. An invalid action means that server can't process this job by its deadline, and this can be calculated by its current state including this server's waiting queue precisely. So firstly, in our scheduling system, we add a process to evaluate every server at each step to verify whether it will satisfy the job's requirement. And this information will be added to the servers' state at each time step as a Boolean variable, true for it can, false for it can't. We want the network to learn this rule in an implicitly and natural way. Therefore, the state of server will be changed to:

$$Server = (R, x, y, C_t, U_t, isValid_t)$$

   Also, for the output of network, we will ignore the invalid actions' output and choose by $greedy, \epsilon - greedy$ among the valid actions.

2. Off-Policy

Actually, our method still partly accords with the off-policy method in original DQN. Instead of using $\epsilon - greedy$ as the behavior policy, we use an implicitly policy from the historical data. And then evaluated this policy by learning the Q function. But since the behavior policy and target policy did not share the same Q function as the Q-Learning did, behavior policy will not be updated during the training process.

This feature also brings some potential modification to the method, we can replace any behavior policy like policy from human experiences. Or making our training off-line which means we can deploy the policy to real environment for test after training. After testing, we can re-train policy, this would prevent any unpredictable situation happen if we use on-line learning.

3. Non-Terminal State
   During training, we will not set the terminal state since we do not use Monte-Carlo method.

4. Training Sample

   Since no real environment was involved, the interaction between agent and environment is eliminated. In experiments, we will handle the data into to minibatch of transitions like: $(State_t, Action_t, Reward_t, State_{t+1})$ for training.

2. Modified Deep Q-Learning Algorithm Pipeline

```
**Algorithm 1** Modified Q-learning
Initialize action-value function $Q$ with random weights $\theta$ and target $Q^{'}$ function
with weights $\theta^{'}$
  **for** $episode = 1, M$ **do**
    **for** $t = 1, T$ **do**
        Randomly select minibatch from dataset $(state_j, action_j, reward_j, state_{j+1})$
        Set $y_j = reward_j + \gamma max_{a^{'}} Q^{'}(state_{j+1}, a^{'}; \theta^{'})$
        Perform a gradient descent step on $(y_j - Q(state_j, action_j; \theta)^2$ to update $\theta$
        **if** t mod UpdateTargetQEveryStep = 0 **then** Update $\theta^{'}$
      **end if**
    **end for**
  **end for**
```

Figure 3. Pseudocode of method

IV. SIMULATION AND RESULTS

1. Configuration and Generating Historical Data

According to problem, we have the historical data with following formulation (one sample):

```
{
    "ACTION": 5,
    "NEXT_STATE": {
      "JOB_STATE": {
        "FINISH_TIME": 4,
        "PROCESS_TIME": 4,
        "DEADLINE": 1086,
        "RESOURCES": 9,
        "SUBMIT_TIME": 1
      },
      "DC": {
        "TOTAL_HEAT": 8.351714200886092e-17
      },
      "SERVER_STATE": {
        "0": {
          "RESOURCE": 40,
          "ENERGY": 0.0,
          "USAGE": 0.0,
          "HEAT": 0.0,
          "IS_VALID": 1,
          "Y": 0,
          "X": 0
        }
      }
    }
    "STATE": {
      "JOB_STATE": {
        "FINISH_TIME": 2,
        "PROCESS_TIME": 3,
        "DEADLINE": 1795,
        "RESOURCES": 10,
        "SUBMIT_TIME": 0
      },
      "DC": {
        "TOTAL_HEAT": 4.6919742701607256e-17
      },
      "SERVER_STATE": {
        "0": {
          "RESOURCE": 40,
          "ENERGY": 0.0,
          "USAGE": 0.0,
          "HEAT": 0.0,
          "IS_VALID": 1,
          "Y": 0,
          "X": 0
        }
      }
    },
    "REWARD": {
      "REWARD": 27.000000047999997
    },
    "TIME": 0
}
```

Figure 4. Structure of sample

Due to the limitation when this report was written, we have to generated this data by some simple simulation and it was given by a very simple simulation without losing the features and formulations of the original problems. The data generator process was shown as below:

a. Assume currently time is t.
b. Randomly generate a job request job_t
c. Evaluate each server to verify whether this server can satisfy the job's requirements like finishing before deadline, enough resources etc. The proposed server list was defined as $ValidServerList_t^{job_t}$. And we assume once this job was starting being computed by server, it can't pause or change hosting server.
d. Choose randomly (or following some basic policy) from $ValidServerList_t^{job_t}$.
e. Simulate next new state: $Server_{t+1}, Heat_{t+1}$.
f. Compute reward at $Reward_t$
g. $t++$, Go back to step b

Figure 5. Pseudocode of data generating process

Some basic configurations and computing methods of environment were defined as follows:

1. Total time steps: 2200
2. Total simulation time steps: 2000 (after 2000, we didn't add new jobs)
3. Total server counts: 6
4. Server resources: 40
5. Server heat constant: 200 (the heat one server generated each time when under full usage)
6. Servers position (in 2-dimensional grid): (0, 0), (1, 0), (-2, 0), (-2, 2), (3, 1), (3, 3)
7. Total job counts: 2000 (equal with total simulation time steps)
8. Compute $Heat_t^{all}$:

$$Heat_t^{all} = \sum_{i=1}^{ServerCount} (Heat_t^i * C_t * ServerHeatConstant * (1/exp(Distance^{server_i})) + decay * Heat_{t-1}^{all}$$

Eq 6

$Distance^{server_i}$ is the server's distance to the center of data center. By multiply factor $1/exp(Distance^{server_i})$, we assumed that the serve located nearing edge of data center will have a better thermal condition, like more cooling air etc. so it will decay its heat by multiplying this factor.

$Decay$ is the factor deciding how much heat from previous state will be loss to the outside of data center.

2. Implementation details of modified DQN:

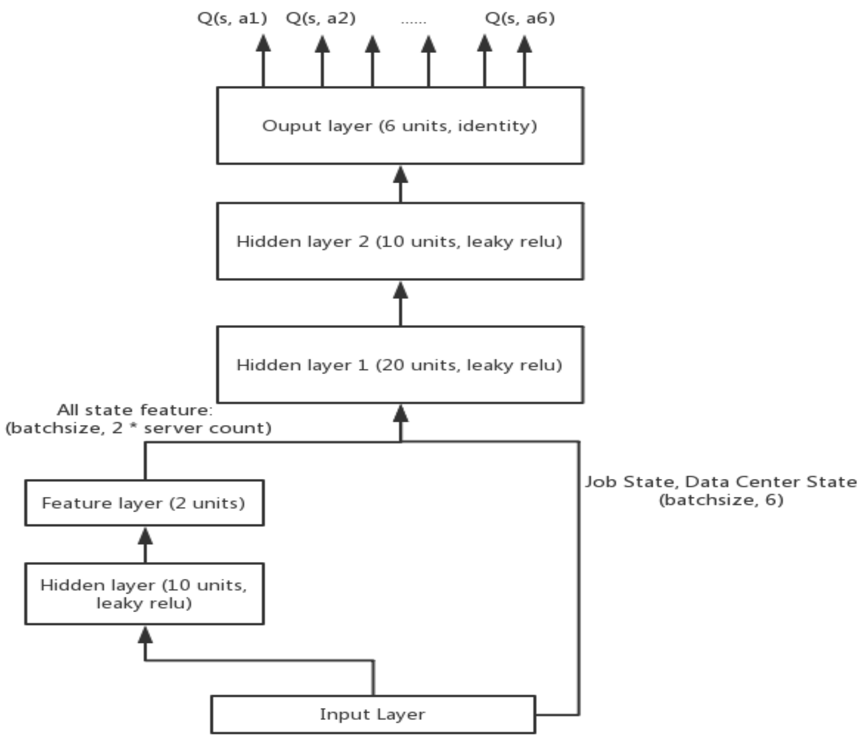
Q(s, a1)
Q(s, a2)
......
Q(s, a6)
Ouput layer (6 units, identity)
Hidden layer 2 (10 units, leaky relu)
Hidden layer 1 (20 units, leaky relu)
All state feature:
(batchsize, 2 * server count)
Job State, Data Center State
(batchsize, 6)
Feature layer (2 units)
Hidden layer (10 units, leaky relu)
Input Layer

Figure 6. Detailed structure of Q network

Since we avoid sample data from real environment, we didn't have to use experiences replay to break the time relation between each sample. We used the same target Q network trick to stable the training process. Target Q network was updated every fixed time steps.

The loss function, beside we defined before, was added L1 and L2 regularization loss w.r.t the parameters in Q network. The optimizer was RMSProp.

The activation for hidden layer is Leaky-Relu, for output layer is identity function.

All input state is standardized into the range of [0, 1], and the action was transformed into one-hot code.

3. Results:
   Due to the time limitation, we didn't build an interaction environment to test the model and compute the cumulative reward. But since we know the data generating process and the design of reward function, so we should have some expectation of our learned Q function and its action selection with greedy selection.

   a. Firstly, we define the efficiency which indices that we should put as much jobs together running on one server as possible instead of scattering them into many servers. By doing so (which is a common method in scheduling problem called task consolidation), we can decrease more servers' usage to 0 and turn it off to eliminating the idle consumption. So we believe a reasonable policy should tend to choose a server with higher usage or efficiency in the proposed valid server list. By averaging the chosen servers' efficiency and non-chosen servers' efficiency over each mini-batch we have the following graph, which shows a tendency that chosen server mostly had a higher efficiency tend.

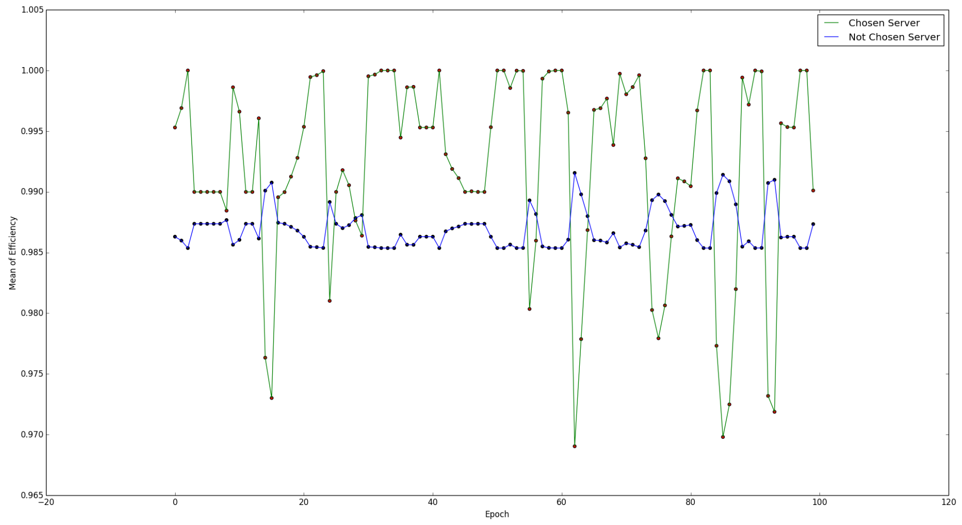

Figure 7. Average efficiency over chosen actions (green line), and over not chosen actions (blue line). The x-axis is different mini-batch

b. Secondly, as the way we define the heat computing method in *Section Configuration and Generating Historical Data,* we think a reasonable policy should tend to choose the server with larger distance to its center of data center. Because a server with larger distance will loss more heat and create a better thermal condition under our assumption. The following graph was computing by the similar way as *Figure 7.*

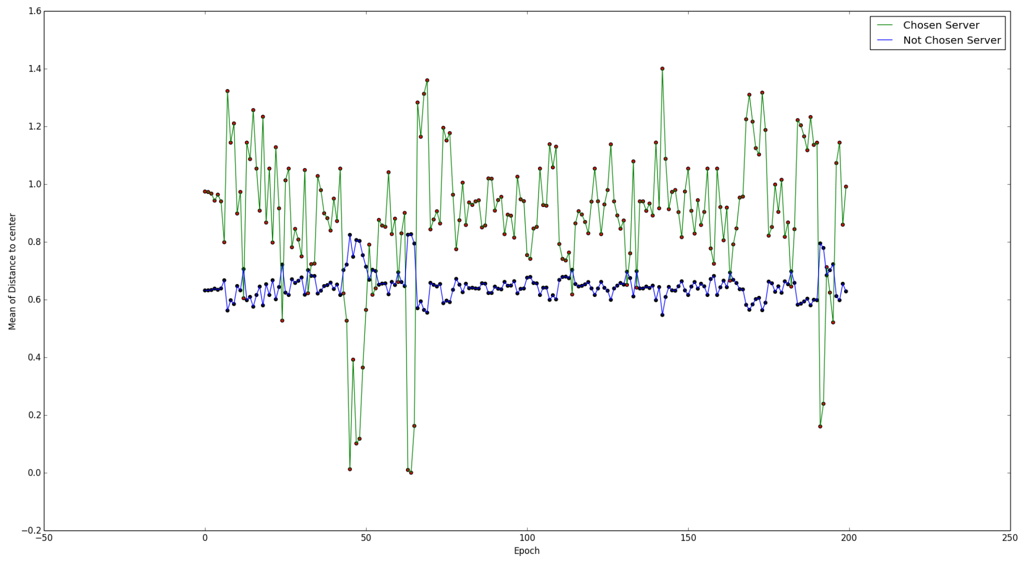

Figure 8. Average Distance over chosen actions (green line), and over not chosen actions (blue line). The x-axis is different mini-batch

V. CONCLUSION

1. In our report, we give the formal definition of Data Center Job Scheduling Problem (DCTSP), then in order to solving it by Deep Reinforcement Learning (DRL), we give the formulation of Markov Decision Process (MDP) according to DCTSP's formulation and objective. At last, we used Deep Q-learning to solve it and give some reasonable results.

2. The key challenges to solve this problem can be seen as follows:
   a. First is how to define MDP according to original problem, especially the design of the reward functions is the key part inside it. The reward function is the only signal to direct our controller to update. So it should be designed so that it can indicate what the state we think is good and what the features a good policy should have. We may don't need to define them precisely, but we should hold a basic expectation of them, like in our experiments, the policy should tend to choose server with higher efficiency and higher distance.

   b. Second is how to use history data to fit into the model instead of learning in a real environment. Usually, in Q learning off policy method, the behavior policy and target policy always shared a same Q value function, but in this problem, we only have the history trace of the behavior policy and we can't update it. Usually, with this situation, we should use importance sampling for off-policy method, further improvement could focus on this part.

   c. The last one is how to add human domain knowledge, in this problem, some decisions like how to propose the valid server to choose from can be done with precisely defined rule. By doing so, we will make the training process more easily, and make the model focusing on the crucial decision part instead of learning some redundant knowledge. In our method, we add some extra information to the state of servers (a boolean to indicates if this server is valid) to achieve it.