VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

Deep Learning Based Scheduler for Cloud and Fog Computing Environments

A project submitted in partial fulfillment of the requirements for the degree of Bachelor of Technology inComputer Science and Engineering

By

Aman Sharma	20BCT0267
Samruddhi Dhavale	20BCT0074
Ananya Thakre	20BCT0269

Course Instructor

Dr.Baiju B V

Assistant Professor

April 2023

UNDERTAKING

This is to declare that the project entitled **Deep Learning Based Scheduler for Cloud and Fog Computing Environments** is an original work done by undersigned, in partial fulfillment of the requirements for the degree "Bachelor of Technology in Computer Science and Engineering" at School of Computer Science and Engineering, Vellore Institute of Technology (VIT), Vellore.

All the analysis, design and system development have been accomplished by the undersigned. Moreover, this project has not been submitted to any other college orUniversity.

Team Member Name

Aman Sharma

[illegible]

Samruddhi Dhavale

[illegible]

Ananya Thakre

ABSTRACT

A potential paradigm for providing low-latency services at the network edge to resource-hungry Internet of Things (IoT) applications is EC, or edge computing. Scheduling application operations is quite challenging, though, because the edge server's computational resources have a limited capacity. In our project, we investigate the issue of work scheduling in the EC scenario and schedule various jobs to virtual machines (VMs) created at the edge server by optimising the level of long-term job satisfaction (LTSD). While taking into account problem variety and resource heterogeneity, we apply deep reinforcement learning (DRL) to tackle the problems of time scheduling (i.e., the order in which tasks are executed) and resource allocation (i.e., which VM the task is given to). A fully-connected neural network is used to extract the features for the task scheduling problem using a policy-based REINFORCE technique (FCN). According to simulation results, the suggested DRL-based task scheduling algorithm outperforms the standard methods in terms of average task satisfaction level and success ratio.

TABLE OF CONTENTS

1. INTRODUCTION

Two specific issues, time scheduling and resource allocation, must be taken care of in order to plan jobs in edge computing. Time scheduling determines the order in which tasks are completed, while resource allocation is in charge of assigning tasks to suitable virtual resources execution on virtual machines (VMs). For the purpose of achieving a long-term objective, the Markov decision process (MDP) is a valuable tool for modelling the sequential decision-making problem.

Reinforcement learning (RL), in which the agent continuously interacts with the environment and makes sequential decisions, has been established as a potential means of resolving MDP problems. Instead of focusing on the local optimal solution in real time, the agent's ultimate goal is to create an ideal policy that maximises the cumulative reward. It is inconvenient, especially for large state spaces and continuous action spaces, for the mapping between state and action in real-world systems to be maintained in a tabular format.

When paired with the deep neural network (DNN), model-free deep reinforcement learning (DRL) can produce intelligent sequential decisions in challenging circumstances, and the RL table is thus replaced by the DNN's function approximation. In recent years, DRL has been successfully used for resource allocation and time scheduling in edge computing. The deep Q-network (DQN) technique uses numerous replay memories to improve overall latency and resource consumption. The compute resource allocation problem in edge computing is defined as an MDP.

We outline a model for intelligent job scheduling in edge computing in our study. To maximize the long-term value of QoE while accounting for the anticipated delay necessary, we focus on heterogeneous VM resources for task scheduling. The task satisfaction level is defined as the reward in order to accomplish this goal using the DRL approach. The task execution order is determined by the mechanism, and the work is then assigned to the appropriate VM in the second portion of its operation. An MDP is created from the work scheduling process in edge computing, and a policy-based DRL algorithm

2. LITERATURE SURVEY

Ref No	**Authors and Year (Reference)**	**Title (Study)**	**Algorithm Used**	**Merits**	**Demerits**
1	Shuran Sheng, Peng Chen, Zhimin Che , Lenan Wu and Yuxuan Ya	Deep Reinforcement Learning-Based Task Scheduling in IoT Edge Computing	The optimization issues are expressed as an MDP model. The scheduling time step is isolated from the real time step to minimise the dimension of the actionspace. To solve the MDP, a policy-based deep reinforcement learning system is used.	When compared with Greedy-FFCS and Greedy-SJF algorithms, the DRL algorithm performs better in task satisfaction and task completion ratio.	Communication delay has not been taken into account. For that, they need to collaborate on cloud computing and edge computing
2	Zahra Movahedi, Bruno Defude and Amir mohammad Hosseininia	An efficient population based multi-objective task scheduling approach in fog computing systems	The OppoCWOA technique, which combines WOA, Opposition-based learning, and Chaos theory, is applied. Opposition-based learning is utilised to introduce variety during the WOA initialization phase. To prevent the influence of unpredictability on the progress towards the best solution, the chaos theory is included into WOA.	They have used a partial opposition strategy to enhance task scheduling performance while attaining faster convergence. Their simulation was shown to be better than the WOA, ABC, PSO and GA approach.	To deal with the stochastic nature of the fog environment, AI-based technologies like deep reinforcement learning need to be used to rapidly learn and change policy weights based on fog node and task burden characteristics
3.	Shreshth Tuli, Shashikant Ilager, Kotagiri Ramamohanarao and Rajkumar Buyya	Dynamic Scheduling for Stochastic Edge-Cloud Computing Environments using A3C learning and Residual Recurrent Neural Networks	To enable effective scheduling decisions, the R2N2 architecture captures a large number of host and task characteristics as well as temporal trends. A3C is used to quickly adapt to dynamic scenarios with less data. The input and output parameters of the Neural Networks are presented and then it is modelled.	This technique can cut energy usage by 14.4%, response time by 7.74%, SLA violations by 31.9%, and cost by 4.64%. Also, when compared to the existing baseline, the model has a low scheduling cost of 0.002 percent.	This model is not implemented in real cloud-edge environments. Implementing new activities in real-world situations would necessitate continual assessment of CPU, RAM, and disc requirements.

4.	Malik Louail, Moez Esseghir and Leila Merghem-Boulahia	Dynamic task scheduling for fog nodes based on deadline constraints and task frequency for smart factories	A real-time dynamic task scheduler is made that treats deadline and frequency limitations as an urgency degree at the fog level in order to appropriately handle incoming tasks. A fog network based on auctions is taken into account, in which jobs rejected by one fog might be accepted by another.	It is seen that when compared to other works, this scheduler demonstrated an optimization in terms of the number of rejected tasks by prioritising essential jobs with a low value of emergency degree.	They have not developed the distributed queuing network to maintain workload balance across fog nodes. They can also develop the process of resource allocation when heterogeneou s multi fog nodes virtualize create a distributed system.
5.	Simar Preet Singh, Anand Nayyar, Harpreet Kaur and Ashu Singla	Dynamic Task Scheduling using Balanced VM Allocation Policy for Fog Computing Platforms	Multiple task scheduling models are worked on based upon the Local Regression (LR), InterQuartile Range (IQR), Local Regression Robust (LRR), Non-Power Aware (NPA), Median Absolute Deviation (MAD), DynamicVoltage and Frequency Scheduling (DVFS) and The Static Threshold (THR) methods using the ifogsim simulation.	The proposed model was trained to minimise queue size by effectively allocating resources, resulting in faster completion of user activities. The LRR model performed better than other models in terms of VM migrations.	This proposed algorithm was confined to task scheduling only. It can also be extended towards load balancing and scheduling algorithms.
6.	Tao Zheng, Jian Wan, Jilin Zhang & Congfeng Jiang	Deep Reinforcement Learning-Based Workload Scheduling for Edge Computing	With the intention of balancing the workload, cutting down on service time, and decreasing the rate of failed tasks, they suggested a deep reinforcement learning (DRL)-based workload scheduling approach. They use Deep-Q-Network(DQN) methods in the meantime to resolve the difficult and multidimensional workload scheduling problem.	Demonstrate that the suggested approach performs best in terms of service time, VM consumption, and failure task rate.	One of it's main limitation as showed in their "Results and Evaluation" section is presence of tasks failure which happen in the simulation due to a multitude of reasons.

7.	Sundas Iftikhar, Mirza Mohammad Mufleh Ahmad, Shreshth Tuli, Deepraj Chowdhury Minxian Xu, Sukhpal Singh Gill, Steve Uhlig	HunterPlus: AI based energy-efficient task scheduling for cloud–fog computing environments	This study examines the impact of changing the GGCN's Gated Recurrent Unit to a Bidirectional Gated Recurrent Unit using a novel methodology dubbed HunterPlus. The use of Convolutional Neural Networks (CNNs) for cloud-fog job scheduling optimization is also explored in this paper. According to experimental findings, the CNN scheduler beats the GGCN-based models by at least 17 and 10.4%, respectively, in terms of energy consumption per task and work completion rate measures.	By consuming less energy, cloud providers might potentially increase cost reduction. The quantity of energy used can be decreased by using clever task-scheduling algorithms to distribute user-deployed jobs to servers. Heuristic and metaheuristic techniques are both used in traditional work scheduling algorithms.	It's main limitation is due to the use of gated recurrent units. (GRUs) in this paper it cannot capture long range memory dependencies
8.	S. Tuli, Nipam Basumatary, R. Buyya	EdgeLens: Deep Learning based Object Detection in Integrated IoT, Fog and Cloud Computing Environments	EdgeLens, the suggested framework, can deliver high accuracy or low latency service modes by adapting to the application or user needs. They demonstrated how EdgeLens adapts to various service requirements as well as tested the software's performance in terms of accuracy, response time, jitter, network bandwidth, and battery consumption.	Object recognition and surveillance have become of particular relevance, the cloud is unable to reduce network latencies to fulfil the response time standards. Fog computing, which uses faraway cloud resources as well as network edge resources as needed, solves this problem.	One of the main limitation of this Deep learning based object detection in IOT systems is the presence of unstable training problems while training the deep learning model to reduce latency.

9.	Shashank Swarup, Elhadi M. Shakshuki, Ansar Yasar	Energy Efficient Task Scheduling in Fog Environment using Deep Reinforcement Learning Approach	In order to reduce energy use, costs, and service delays, the scheduling of IoT tasks in a fog-based environment is the main topic of this study. In order to do this, the Clipped Double Deep Q-learning algorithm, which is based on deep reinforcement learning and uses target networks and experience replay techniques, is suggested. A parallel queueing system is used to guarantee that resources are used as quickly as possible. The long wait times for tasks in the virtual machine queue are a challenge that needs to be addressed in cloud (and fog) computing research. In this paper, a dual queue approach is suggested..	Contrary to standard cloud computing, edge and fog nodes can be positioned close to IoT devices to significantly reduce latency. A number of time-sensitive services and applications benefit from the ultra-low latency response of the new fog computing technology. Fog nodes are placed in less centralised locations.	It's main limitation is the training of reinforcement learning agents/ environment is very time consuming due to high latency period in the sensors used in this experiment.
10.	Reza Ebrahim Pourian, Mehdi Fartash, Javad Akbari Torkestani	A Deep Learning Model for Energy-Aware Task Scheduling Algorithm Based on Learning Automata for Fog Computing	An energy-conscious jobscheduling technique based on learning automata (LA) in the Fog Computing (FC) Applications is presented in this research as an AI deep learning model. The task scheduling problem is also solved using an algorithm based on LA that measures the cost and makespan (MK) parameters. The results of the suggested model demonstrate that all desired parameters can be predicted with great accuracy	The task scheduling problem is also solved using an algorithm based on LA that measures the cost and makespan (MK) parameters. Then, on the basis of the above LA task scheduling fog computing technique, a novel deep artificial neural network model is suggested. For the first time, the suggested neural model can forecast the relationship between MK, energy, and cost parameters versus VM length.	One of it's major limitations is the algorithm is very computation intensive due to presence of millions of parameters in the deep learning model.

3. PROPOSED METHODOLOGY

INTRODUCTION

- In this model we develop a deep reinforcement learning based model that can identify the different resources available in a cloud centre and allocate the same efficiently. Our model uses the functionality of DQN- Deep Q-Learning, which is implemented using the Pytorch Library.
- The three main parts of the model are:
  - The Agent
  - The Environment
  - The Action
- The Agent is essentially the function of the allocation and scheduling model which uses the linear DQN neural network architecture to do its job in the environment.
- By environment, here we refer to the resources that require allocation and scheduling And finally, the action in itself is the allocation and scheduling of said resources in the environment.

INPUT DATASET

input.txt

Input.txt: the input data of user workload model from Google cluster-usage traces after extracting.

LIBRARIES USED

- Numpy
- Pytorch
- Matplotlib

DEEP LEARNING CONCEPTS EMPLOYED

- Linear Neural Network
- Deep Q Network
- Reinforcement Learning
- Google cluster-usage traces- for training purposes

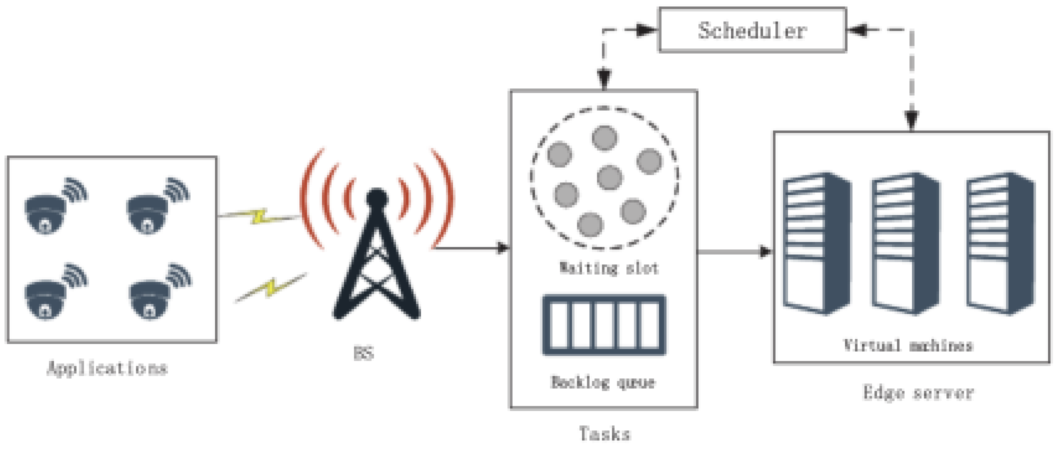
PYTHON BASED PARAMETERS- REWARD AND REINFORCEMENT PURPOSES

- Mean Squared Error- Loss function
- Learning rate was 0.0001, epsilon = 1 to 0.01 to explore the action space

These parameters decide the performance of the model based on which the following three models decide the reward points and the levels of reinforcement learning that must take place to make the model more efficient. Models defining the environment:

- Energy model,
- reward functions,
- rejection mode

4. ARCHITECTURE/DESIGN



1. Using the PyTorch library, create the DQN.

2. Space of state: The current state of accessible resources, such as the server's CPU and memory

3. Space for action: Total number of options, i.e. total number of server farms (stage 1), total number of servers (stage 2), total number of servers (stage 3), total number of servers (stage 4) (stage2)

4. Reward Function: Based on the DQN agent's decision, we provide it with rewards by estimating the amount of power it used during the procedure.

5. Experience points: To gain from unusual experiences, we store the actions and rewards and then choose from the memory buffer and learns from it over a period of time. As a result, our model is not overly biassed even when the same sort of actions occur.

6. Environment: To imitate the cloud environment, a hierarchy comprising server farms, servers, and virtual machines was constructed and CPU and memory were allotted. Here, an energy model, reward functions, and a rejection model are created.

5. DESCRIPTION ON VARIOUS MODULES

- **NumPy**

  NumPy is a Python library for scientific computing that provides support for arrays, matrices, and mathematical operations on them. It is widely used for numerical calculations in fields such as engineering, science, and data science. NumPy is built on top of the C programming language and is designed to be efficient and fast.

- **PyTorch**

  PyTorch is an open-source machine learning framework that is widely used for building and training neural networks. It is developed by Facebook's AI research team and is based on the Torch library, which is a scientific computing framework. PyTorch is written in Python and allows for dynamic computation graphs, which enables users to change the network architecture on the fly during training.

- **Matplotlib**

  Matplotlib is a popular open-source data visualization library for Python that provides a wide range of tools for creating static, animated, and interactive visualizations in Python. It is widely used in scientific computing, data analysis, and machine learning for visualizing data and results.

6. IMPLEMENTATION

TRAINING MODEL OF DEEP NEURAL NETWORK

```
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch as T
import random
#from util import plot_learning_curve
class LinearDeepQNetwork(nn.Module):
    def __init__(self, lr, n_actions, input_dims):
        super(LinearDeepQNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dims, 128)
        self.fc2 = nn.Linear(128, n_actions)
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)
#takes the current and returns list of actions
    def forward(self, state):
        layer1 = F.relu(self.fc1(state))
        actions = self.fc2(layer1)
        return actions
class Agent():
    def __init__(self, input_dims, n_actions, lr, gamma=0.99,
                 epsilon=1.0, eps_dec=1e-5, eps_min=0.01):
        self.lr = lr
        self.input_dims = input_dims
        self.n_actions = n_actions
        self.gamma = gamma
        self.epsilon = epsilon
        self.eps_dec = eps_dec
        self.eps_min = eps_min
        self.action_space = [i for i in range(self.n_actions)]
        self.Q = LinearDeepQNetwork(self.lr, self.n_actions, self.input_dims)
    def choose_action(self, state):
        if np.random.random() > self.epsilon:
            state1 = T.tensor(state, dtype=T.float).to(self.Q.device)
            # state =
            actions = self.Q.forward(state1)
            action = T.argmax(actions).item()
        else:
            action = np.random.choice(self.action_space)
        return action
    def decrement_epsilon(self):
        self.epsilon = self.epsilon - self.eps_dec \
                        if self.epsilon > self.eps_min else self.eps_min
    def learn(self, state, action, reward, state_):
        self.Q.optimizer.zero_grad()
        states = T.tensor(state, dtype=T.float).to(self.Q.device)
        actions = T.tensor(action).to(self.Q.device)
        rewards = T.tensor(reward).to(self.Q.device)
        states_ = T.tensor(state_, dtype=T.float).to(self.Q.device)
        q_pred = self.Q.forward(states)[actions]
        q_next = self.Q.forward(states_).max()
        q_target = reward + self.gamma*q_next
        loss = self.Q.loss(q_target, q_pred).to(self.Q.device)
        loss.backward()
        self.Q.optimizer.step()
        self.decrement_epsilon()
    def processDQN_stage1(self, initial_state):
        action = self.choose_action(initial_state)
        return action
    def processDQN_stage2(self,initial_state ):
        action = self.choose_action(initial_state)
            return action
```

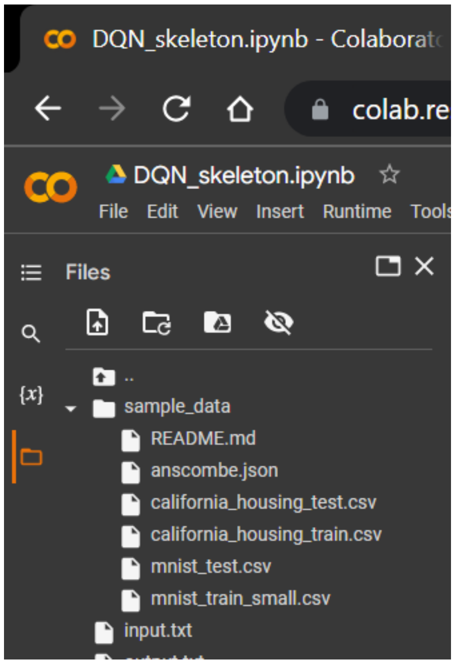
TESTING MODEL FOR ACTION AND ENVIRONMENT

```
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch as T
import random
import time
class Task(object):
    """
    information of each task
    parent, child base on dependency
    jobID, index, CPU, RAM, disk extracted from user data
    status indicates the current status of the task
    """
    def __init__(self, jobID, index, CPU, RAM, disk, status):
        import random
        import time
        self.parent = []
        self.child = []
        self.jobID = jobID
        self.index = index
        self.CPU = CPU
        self.RAM = RAM
        self.disk = disk
        self.status = status  #-1: rejected, 0: finished, 1: ready, 2: running
        self.runtime = random.randint(1, 10)/1000.0
        self.ddl = time.time() + self.runtime + random.randint(1, 1000) * 100
        self.endtime = 0

class DAG(object):
    """
    Transform job queue to task ready queue
    """
    def __init__(self, fname, num_task):
        self.fname = fname
        self.num_task = num_task
        self.job = []
        self.task = []
    def readfile(self):
        """
        Read the input job file
        All task are initialized to ready status
        """
        num_task = 0
        with open(self.fname, 'r') as f:
            task = []
            for line in f:
                if line[0] == 'J':
                    if len(task) != 0:
                        self.job.append(task)
                        task = []
                else:
                    info = list(line.strip(' ').split())
                    task.append(Task(info[1], info[2], float(info[4]), float(info[5]), info[6], 1))
                    num_task += 1
                if num_task == self.num_task:
                    break
            if len(task) != 0:
                self.job.append(task)

    def checkRing(self, parent, child):
        """
        Check whether there is a loop between parent and child
        Return True if has loop
        """
        if parent.index == child.index:
            return True
        if len(child.child) == 0:
            return False
        for c in child.child:

            if self.checkRing(parent, c):
                return True
        return False
    def buildDAG(self):
        """
        Randomly build dependencies between tasks within each job
        """
        import random
        for job in self.job:
            for task in job:
                i = random.randint(-len(job), len(job) - 1)
                if i < 0:
                    continue
                parent = job[i]
                if self.checkRing(parent, task) == False:
                    task.parent.append(parent)
                    parent.child.append(task)

    def rejTask(self, task):
        """
        If one task is rejected
        Then all tasks that depended on this task will be rejected
        """
        task.status = -1
        for c in task.child:
            self.rejTask(c)

    def hasParent(self, task):
        """
        When a task are finished
        Remove it from the parent for all child tasks
        """

        for c in task.parent:
            if c.status == 1:  #still has parent
                return True
        return False

    def updateStatus(self, task):
        """
        Given jobid and taskid, change status of all tasks that depend on it
        If the task with "-1" status, reject this tasks' all child tasks
```

```
        If the task with "0" status, remove it from all child tasks
        """
#        job_i, task_i = self.findTask(task.jobID, task.index)
#        if job_i == -1 or task_i == -1:
#            print("WRONG: The task with jobID: ", task.jobID, " and taskID: ", task.index, " not exist.")
#            return
#        job = self.job[job_i]
#        task = job[task_i]
        if task.status == -1:
            self.rejTask(task)
#        elif task.status == 0:
#            self.rmParent(task, task_i, job)

    def initTask(self):
        """
        run readfile and buildDAG functions
        """
        self.readfile()
        self.buildDAG()

    def taskQueue(self):
        """
        Build the task ready queue

        Just put the one whose status is 1
        and whose parent are all finished
        """
        for job in self.job:
#            num_task = len(job)
#            while num_task > 0:
            for task in job:
                if task.status == 1 and self.hasParent(task) == False:
                    self.task.append(task)
#                    task.status = 0
#                    self.updateStatus(task)
#                    num_task -= 1
#        for t in self.task:
#            t.status = 1
#        print(len(self.task), "requests")
#        self.printTask()

    def printTask(self):
        """
        Print tasks which are in task queue info
        """
        for j in self.task:
            print(j.jobID, ",", j.index, ",", j.status, ",", len(j.parent))
class environment(object):
    """docstring for environment
    the environment of RP/TS processor
    read the task from txt file
    calculate the Reward Function
    interface with DQN and baseline
    """
    def __init__(self, scale, fname, num_task, num_server):
        """
        initial the variable
        We assume each server has 10 VM
        For small-scale problems:
            200 servers
            10 server farms
        For large-scale problems:
            4000 servers
            70 server farms
        All servers have unit CPU, RAM, and Disk space
        """
        self.scale = scale
        self.fname = fname
        self.task = []
        self.dag = DAG(self.fname, num_task)
        self.VMNum = 5
        self.rej = 0
        self.num_task = num_task
        self.severNum = num_server
        if self.scale == 'small':
#            self.severNum = 200
            self.farmNum = 10
        elif self.scale == 'large':
#            self.severNum = 4000
            self.farmNum = int(self.severNum / 50)
        # self.init_severs()
        self.remainFarm = []
        self.FarmResources = []
        self.severs = [[1,1]for _ in range(self.severNum)]
        self.VMtask = []
        self.totalcost = 0

#        print("Total Number of tasks: {0}".format(num_task))

    def init_severs(self, severNum):
        """
        Set the initial values for each VMs
        Each VM has 1/n unit CPU and RAM
        Each VM has a task list
        """
        VM = [[[1.0/self.VMNum, 1.0/self.VMNum]for _ in range(self.VMNum)]for _ in range(severNum)]
#        VM = [[[1.0 , 1.0 ] for _ in range(self.VMNum)] for _ in range(severNum)]
        self.VMtask.append([[[]for _ in range(self.VMNum)]for _ in range(severNum)])
        return VM

    def generateQueue(self):
        self.dag.taskQueue()
        self.task = self.dag.task

    def setFarm(self):
        """
        Randomly set the servers to each farm
        Each farm has at least 1 server and at most 2*m/n-1 servers
        Initial power usage for each servers and each farm
        """
        import random
        self.farmOri = []
        m = self.severNum
        n = self.farmNum
        f = int(self.severNum / self.farmNum)
        for _ in range(self.farmNum):
#            f = random.randint(0,int(2*m/n))
#            f = random.randint(1, int(2 * m / n))
            self.remainFarm.append(self.init_severs(f))

            self.FarmResources.append([f, f])
            self.farmOri.append(f)
            m -= f
            n -= 1

        self.farmOri.append(m)
        self.pwrPre = [0]*self.severNum #power usage pre sever
        self.pwrPFarm = [0]*self.farmNum #power usage per farm

    def elecPrice(self, t, pwr):
        """
        The energy cost on time t
        threshold get from "Impact of dynamic energy pricing schemes on a novel multi-user home energy management system"
        price get from "Optimal residential load
        control with price prediction in real-time electricity pricing environments"
        """
        threshold = 1.5
        if pwr < threshold:
            p = 5.91 #dynamic price
        else:
            p = 8.27
        return pwr * p
    def getPwr(self, r, c):
        """
        Implement the energy consumption model
        r: the remain CPU
        c: the total(unit) CPU
        The parameters' value get from "An energy and deadline aware resource provisioning, scheduling and optimization
       framework for cloud systems"
        """
        # eq.2
        if r < c:
            pwrS = 1
        else:
            pwrS = 0
        alpha = 0.5 #alpha
        beta = 10 #beta
        Ur = (c-r)/c # eq.1
        if Ur < 0.7:
            pwrDy = alpha * Ur
        else:
            pwrDy = 0.7 * alpha + (Ur - 0.7)**2 * beta
        return pwrDy+pwrS

    def rewardFcn1(self):
        """
        Implement the reward function for each farm
        For stage 1: choose the farm
        """
        # eq.5
        pwrCFarm = []
        for i in range(self.farmNum):
            pwrc = self.getPwr(self.FarmResources[i][0], self.farmOri[i])
            pwrCFarm.append(pwrc)
        pwr = sum(pwrCFarm) - sum(self.pwrPFarm)
        self.pwrPFarm = pwrCFarm
        return self.elecPrice(1, pwr)

#     def EnergyFun(self):
#         """
#         Implement the reward function for each server

#         For stage 2: choose the server
#         """
#         # eq.6
#         self.totalCost += self.rewardFcn2()
#         print ("energy cost: ", self.totalCost)

    def rewardFcn2(self):
        """
        Implement the reward function for each server
        For stage 2: choose the server
        """
        # eq.6
        pwrCur = []
        for f in self.remainFarm:
            for s in f:
                sremain = 0
                for v in s:
                    sremain += v[0]
#                 print(sremain)
                pwrc = self.getPwr(sremain, 1.0)
                if pwrc < 0:
                    print("here", sremain)
                pwrCur.append(pwrc)
#                 print("pwrc", pwrc)
        pwr = sum(pwrCur) - sum(self.pwrPre)
        self.totalcost += sum(pwrCur)
#         print("sum(pwrCur)", sum(pwrCur), "sum(self.pwrPre)", sum(self.pwrPre))
        self.pwrPre = pwrCur
#         print("pwr",pwr)
#         print("r2", self.elecPrice(1,pwr))
        return self.elecPrice(1, pwr)
    def release(self):
        """
        Randomly release resources from each VM
        And set the corresponding task as finished
        """
        ranFarm = random.randint(0, self.farmNum-1)
        ranSer = random.randint(0, self.farmOri[ranFarm]-1)
        ranVM = random.randint(0, self.VMNum-1)
        if self.VMtask[ranFarm][ranSer][ranVM]:
            random.shuffle(self.VMtask[ranFarm][ranSer][ranVM])
            t = self.VMtask[ranFarm][ranSer][ranVM].pop()
            t.status = 0
            self.remainFarm[ranFarm][ranSer][ranVM][0] += float(t.CPU)
            self.remainFarm[ranFarm][ranSer][ranVM][1] += float(t.RAM)

    def releaseByTime(self, farm_i, server_i, vm_j):
        curtime = time.time()
        for t in self.VMtask[farm_i][server_i][vm_j]:
            if t.endtime < curtime:
                t.status = 0
                self.remainFarm[farm_i][server_i][vm_j][0] += float(t.CPU)
                self.remainFarm[farm_i][server_i][vm_j][1] += float(t.RAM)
                self.FarmResources[farm_i][0] += float(t.CPU)
                self.FarmResources[farm_i][1] += float(t.RAM)
                self.VMtask[farm_i][server_i][vm_j].remove(t)

    def training(self):
        """
        Run the DQN/baseline in the RP/TS processor environment
        Read the task file by readfile()
        Set the variables
        Pass tasks to agents in real time
        Get the corresponding reward value
        Reject task when R_cpu ≥ C_cpu or R_ram < C_ram
        """
        #send one tesk to dqn and calculate reward
        self.dag.initTask()
        self.generateQueue()
        time_start=time.time()
        print(self.farmNum, end=' ')
        print(self.severNum, end=' ')
        print(self.num_task, end=' ')
        self.trainDQN_v1()
        time_end=time.time()
        timecost = round(time_end-time_start, 3)
        print(timecost, end=' ')
        print(round(self.totalcost, 3), end=' ')
        print()

    def checkRej(self, farm_i, server_i, vm_j, task):
        """
        Check whether this task should be rejected in ith sever, jth VM
        Reject task when current time + task's runtime > task's ddl
        """
        import time
        if task.CPU > 1/self.VMNum or task.RAM > 1/self.VMNum:
            self.rej += 1
            return -1
        remain_cpu = self.remainFarm[farm_i][server_i][vm_j][0] - float(task.CPU)
        remain_ram = self.remainFarm[farm_i][server_i][vm_j][1] - float(task.RAM)
        curtime = time.time()
        if curtime + task.runtime <= task.ddl:
            if remain_cpu >= 0 and remain_ram >=0:

                return 0  # do not reject
            else:
                return 1  # reject temporarily because cpu or ram
        else:
            self.rej += 1
            return -1  #reject because ddl
    def UpdateServerState(self, tempServerFarm, tempSever, vm_numb, task):
        self.remainFarm[tempServerFarm][tempSever][vm_numb][0] -= float(task.CPU)
        self.remainFarm[tempServerFarm][tempSever][vm_numb][1] -= float(task.RAM)
        self.FarmResources[tempServerFarm][0]  -= float(task.CPU)
        self.FarmResources[tempServerFarm][1] -= float(task.RAM)
        return self.custom_reshape(self.remainFarm)
    def custom_reshape(self, a):
        result = []
        for farNum in a:
            for serNum in farNum:
                result.append(serNum)
        c = np.array(result)
        d = c.reshape(2 * self.VMNum * self.severNum)
        return d
    """
    # initialize the DQN stage 2 and environment for it, DQN(intial_state, Num_actions)
        # state = state of all VMs in a server farm
        # numb_actions = num of VMs in that sever
    # initiate DQN stage 1 and env for it, state will be only server remainFarm
            # reconstruct self.remainFarm with 3 Dimensional array with heiracrchy as
            # farmNumb->serverNumb->vmNumb[cpu, memory, local] eg. [[[0.002, 0.005, 0.003], [],],[],]
            # convert the remainFarm into 1D array
     # fun to upadate server state after the task has been assigned to server farm
     """
    def trainDQN_v1(self):
        rej = 0
        self.setFarm()
        energy = 0
        input_stage2 = input_stage1 = self.custom_reshape(self.remainFarm)
        Agent_stage1 = Agent(lr=0.0001, input_dims=len(input_stage1),
                          n_actions=self.farmNum)
        stage1_current_state = input_stage1
        # input_stage2 = np.array(self.remainFarm[0]).reshape(2*self.VMNum*int(self.severNum/self.farmNum))
        Agent_stage2 = Agent(lr=0.0001, input_dims=len(input_stage2),
                          n_actions=int(self.severNum/self.farmNum))
        stage2_current_state = input_stage2
        acc = 0
        while len(self.task) != 0:
#            print(len(self.task))
            while len(self.task) != 0:
                for t in self.task:
                    if t.status == -1: #rejected
                        self.dag.updateStatus(t)
                        self.task.remove(t)
                    elif t.status == 1:   #ready
                        f = stage1_action = Agent_stage1.processDQN_stage1(stage1_current_state)
                        s = stage2_action = Agent_stage2.processDQN_stage2(stage2_current_state)
                        vm = random.randint(0,self.VMNum-1)
                        self.releaseByTime(f, s, vm)  # release by time
                        rej = self.checkRej(f, s, vm, t)
                        if rej == -1:  #rejected due to ddl
                            t.status = -1
                        # if not reject:
                        elif rej == 0:
                            t.endtime = time.time() + t.runtime
                            stage1_next_state = stage2_next_state = self.UpdateServerState(f, s, vm, t)

#                            print(self.remainFarm)
                            reward_stage2 = self.rewardFcn2()
                            energy += reward_stage2
                            Agent_stage2.learn(stage2_current_state, stage2_action, reward_stage2, stage2_next_state)
                            stage2_current_state = stage2_next_state
                            reward_stage1 = self.rewardFcn1()
                            Agent_stage1.learn(stage1_current_state, stage1_action, reward_stage1, stage1_next_state)
                            stage1_current_state = stage1_next_state
                            self.VMtask[f][s][vm].append(t)
                            t.status = 2
#                            self.dag.updateStatus(t)
                            self.task.remove(t)
                            acc += 1
#                        else:
#                            t.status = -1
#                            rej += 1
            self.generateQueue()
        # print("total number of tasks: {0}, rejected tasks: {1}".format(len(self.task), rej))
        print(round(1 - acc/self.num_task, 3), end= ' ')
p1 = environment('small', 'input.txt')
    p1.training()
```

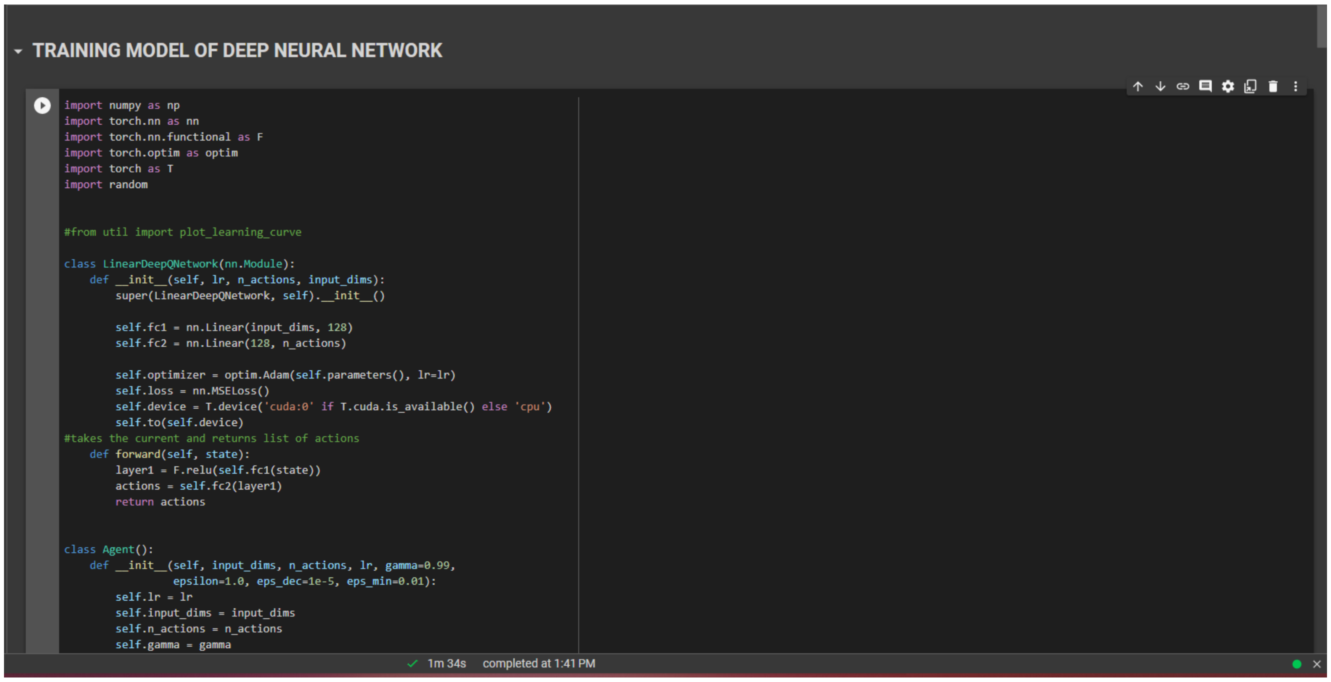
7. TESTING

INSERT INPUT FILE IN DATABASE:



```
..
sample_data
    README.md
    anscombe.json
    california_housing_test.csv
    california_housing_train.csv
    mnist_test.csv
    mnist_train_small.csv
input.txt
```

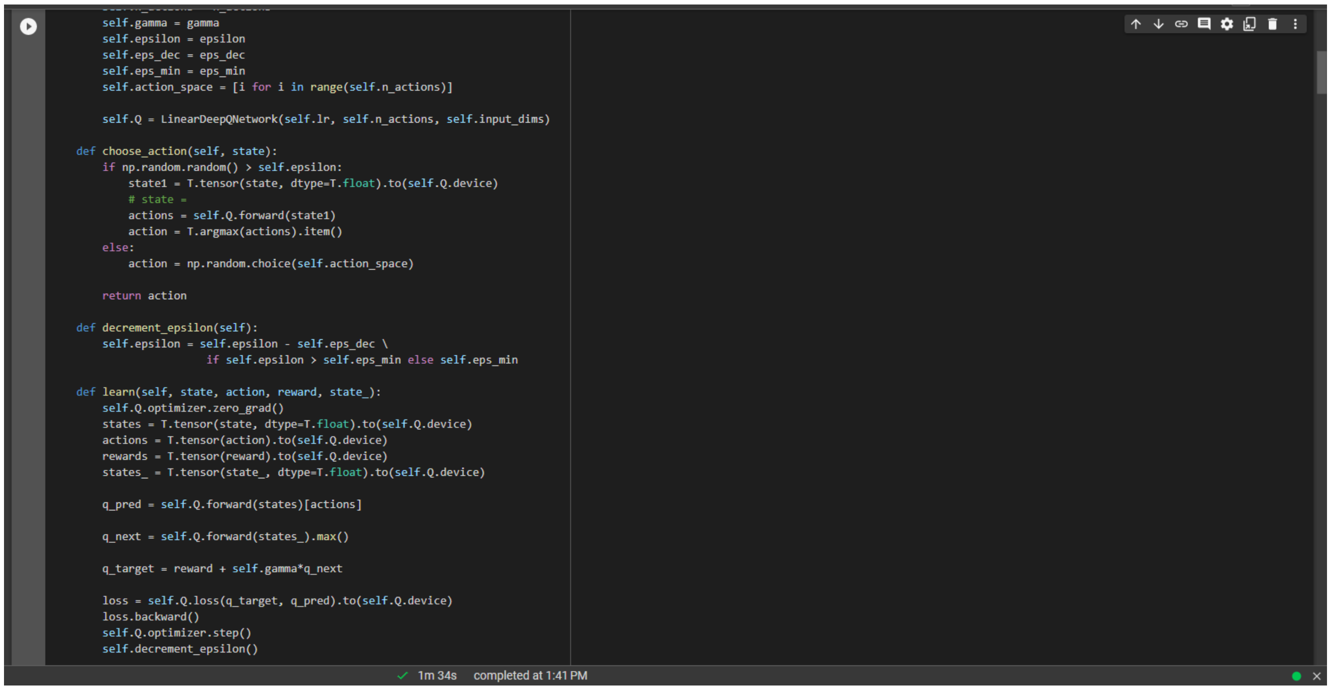
SNAPSHOTS:



TRAINING MODEL OF DEEP NEURAL NETWORK

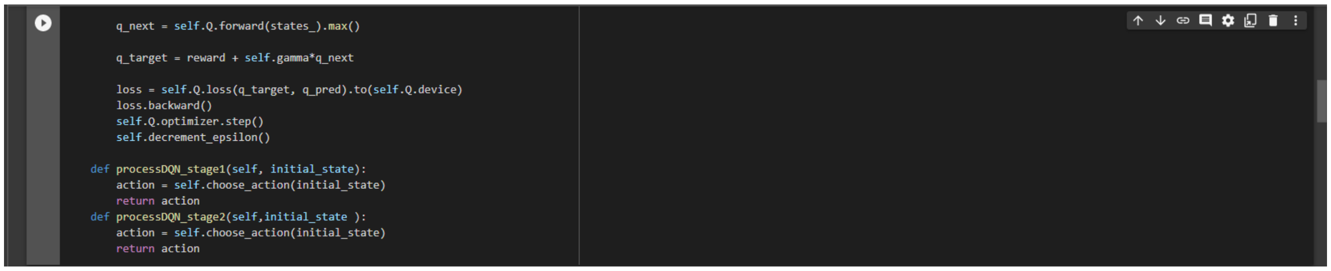
```python
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch as T
import random


#from util import plot_learning_curve

class LinearDeepQNetwork(nn.Module):
    def __init__(self, lr, n_actions, input_dims):
        super(LinearDeepQNetwork, self).__init__()

        self.fc1 = nn.Linear(input_dims, 128)
        self.fc2 = nn.Linear(128, n_actions)

        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)
#takes the current and returns list of actions
    def forward(self, state):
        layer1 = F.relu(self.fc1(state))
        actions = self.fc2(layer1)
        return actions


class Agent():
    def __init__(self, input_dims, n_actions, lr, gamma=0.99,
                 epsilon=1.0, eps_dec=1e-5, eps_min=0.01):
        self.lr = lr
        self.input_dims = input_dims
        self.n_actions = n_actions
        self.gamma = gamma
```

✓ 1m 34s completed at 1:41 PM

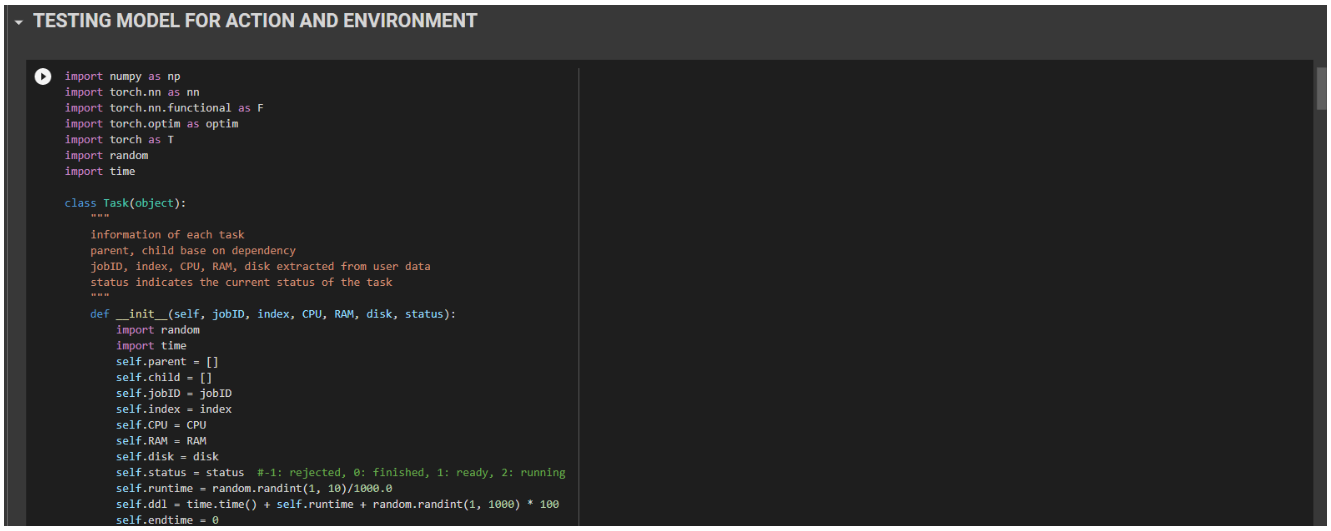


```
        self.gamma = gamma
        self.epsilon = epsilon
        self.eps_dec = eps_dec
        self.eps_min = eps_min
        self.action_space = [i for i in range(self.n_actions)]

        self.Q = LinearDeepQNetwork(self.lr, self.n_actions, self.input_dims)

    def choose_action(self, state):
        if np.random.random() > self.epsilon:
            state1 = T.tensor(state, dtype=T.float).to(self.Q.device)
            # state =
            actions = self.Q.forward(state1)
            action = T.argmax(actions).item()
        else:
            action = np.random.choice(self.action_space)

        return action

    def decrement_epsilon(self):
        self.epsilon = self.epsilon - self.eps_dec \
                        if self.epsilon > self.eps_min else self.eps_min

    def learn(self, state, action, reward, state_):
        self.Q.optimizer.zero_grad()
        states = T.tensor(state, dtype=T.float).to(self.Q.device)
        actions = T.tensor(action).to(self.Q.device)
        rewards = T.tensor(reward).to(self.Q.device)
        states_ = T.tensor(state_, dtype=T.float).to(self.Q.device)

        q_pred = self.Q.forward(states)[actions]

        q_next = self.Q.forward(states_).max()

        q_target = reward + self.gamma*q_next

        loss = self.Q.loss(q_target, q_pred).to(self.Q.device)
        loss.backward()
        self.Q.optimizer.step()
        self.decrement_epsilon()
```

✓ 1m 34s completed at 1:41 PM

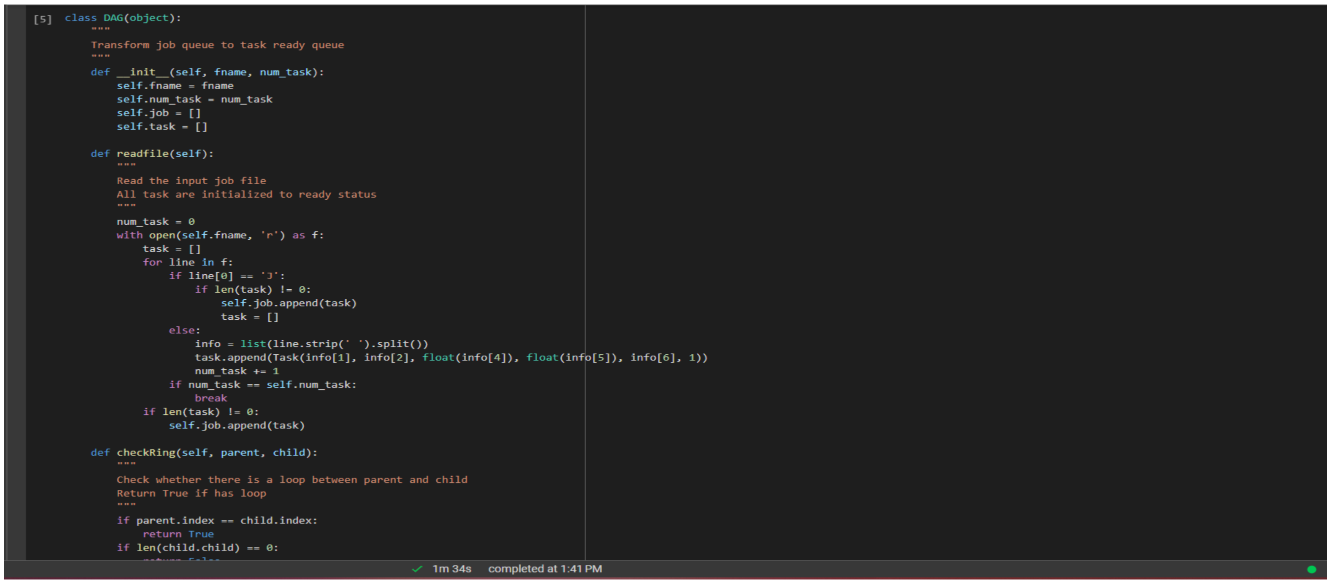


```
        q_next = self.Q.forward(states_).max()

        q_target = reward + self.gamma*q_next

        loss = self.Q.loss(q_target, q_pred).to(self.Q.device)
        loss.backward()
        self.Q.optimizer.step()
        self.decrement_epsilon()

    def processDQN_stage1(self, initial_state):
        action = self.choose_action(initial_state)
        return action
    def processDQN_stage2(self,initial_state ):
        action = self.choose_action(initial_state)
        return action
```



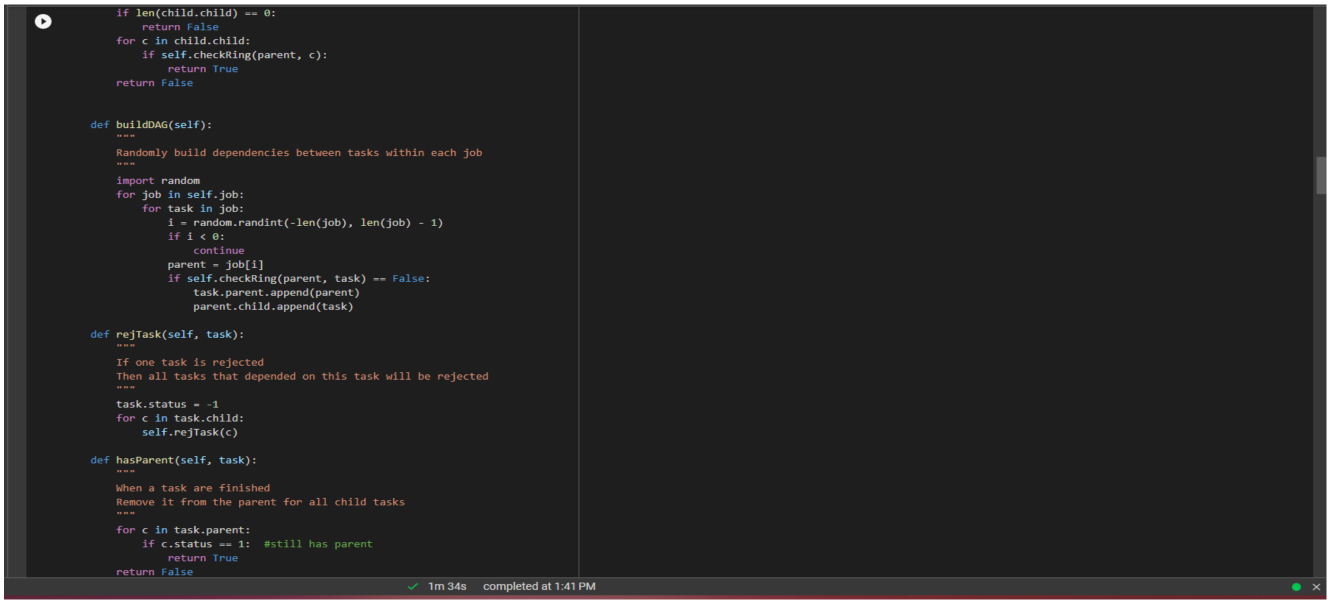
TESTING MODEL FOR ACTION AND ENVIRONMENT

```
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch as T
import random
import time

class Task(object):
    """
    information of each task
    parent, child base on dependency
    jobID, index, CPU, RAM, disk extracted from user data
    status indicates the current status of the task
    """
    def __init__(self, jobID, index, CPU, RAM, disk, status):
        import random
        import time
        self.parent = []
        self.child = []
        self.jobID = jobID
        self.index = index
        self.CPU = CPU
        self.RAM = RAM
        self.disk = disk
        self.status = status  #-1: rejected, 0: finished, 1: ready, 2: running
        self.runtime = random.randint(1, 10)/1000.0
        self.ddl = time.time() + self.runtime + random.randint(1, 1000) * 100
        self.endtime = 0
```



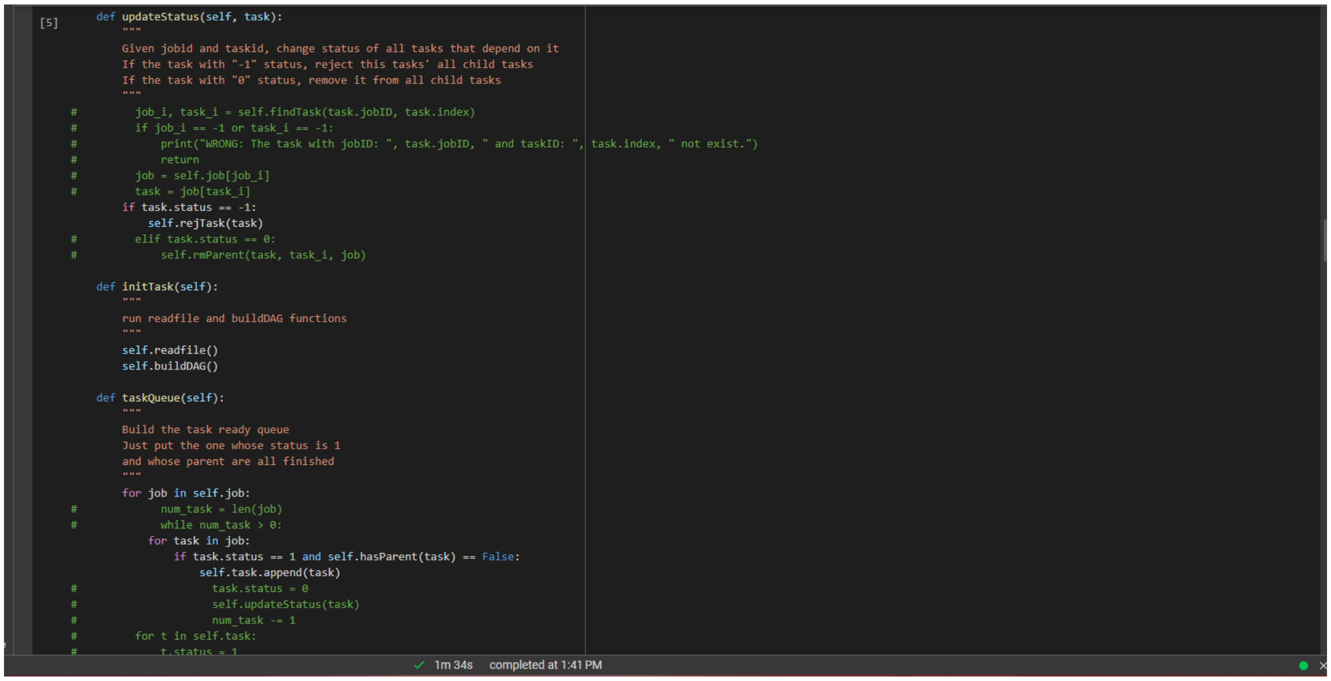
```
[5] class DAG(object):
        """
        Transform job queue to task ready queue
        """
        def __init__(self, fname, num_task):
            self.fname = fname
            self.num_task = num_task
            self.job = []
            self.task = []

        def readfile(self):
            """
            Read the input job file
            All task are initialized to ready status
            """
            num_task = 0
            with open(self.fname, 'r') as f:
                task = []
                for line in f:
                    if line[0] == 'J':
                        if len(task) != 0:
                            self.job.append(task)
                            task = []
                    else:
                        info = list(line.strip(' ').split())
                        task.append(Task(info[1], info[2], float(info[4]), float(info[5]), info[6], 1))
                        num_task += 1
                    if num_task == self.num_task:
                        break
                if len(task) != 0:
                    self.job.append(task)

        def checkRing(self, parent, child):
            """
            Check whether there is a loop between parent and child
            Return True if has loop
            """
            if parent.index == child.index:
                return True
            if len(child.child) == 0:
```

✓ 1m 34s completed at 1:41 PM



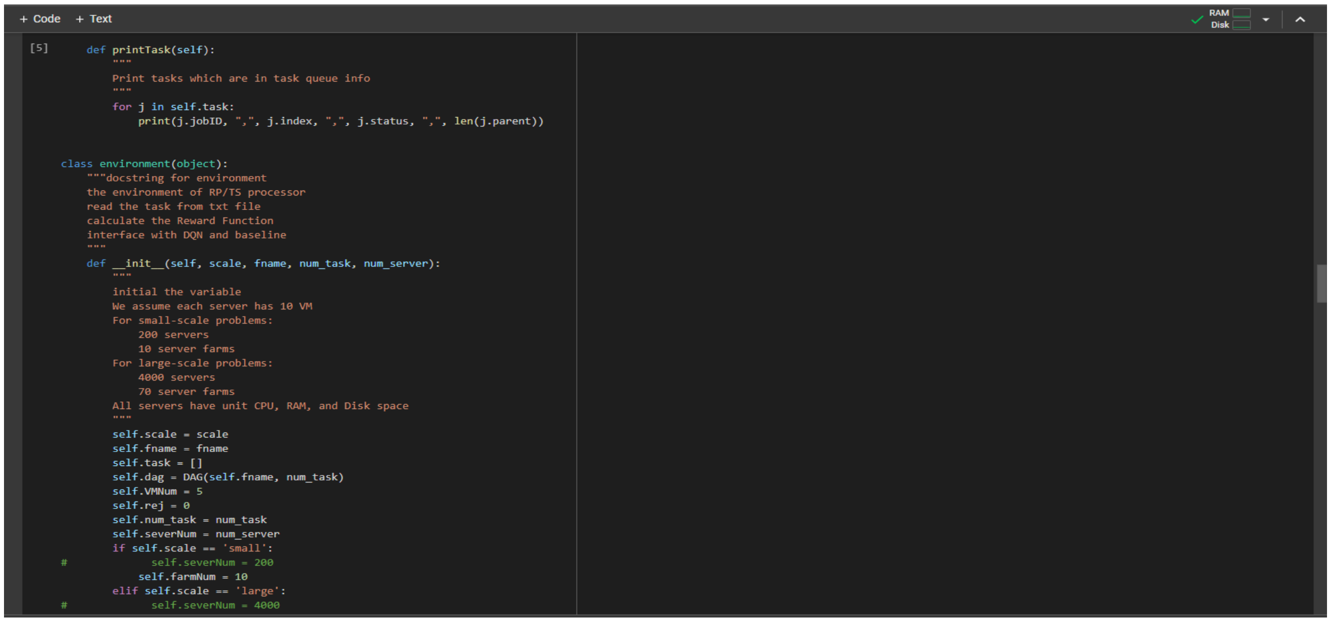
```
            if len(child.child) == 0:
                return False
            for c in child.child:
                if self.checkRing(parent, c):
                    return True
            return False


        def buildDAG(self):
            """
            Randomly build dependencies between tasks within each job
            """
            import random
            for job in self.job:
                for task in job:
                    i = random.randint(-len(job), len(job) - 1)
                    if i < 0:
                        continue
                    parent = job[i]
                    if self.checkRing(parent, task) == False:
                        task.parent.append(parent)
                        parent.child.append(task)

        def rejTask(self, task):
            """
            If one task is rejected
            Then all tasks that depended on this task will be rejected
            """
            task.status = -1
            for c in task.child:
                self.rejTask(c)

        def hasParent(self, task):
            """
            When a task are finished
            Remove it from the parent for all child tasks
            """
            for c in task.parent:
                if c.status == 1:  #still has parent
                    return True
            return False
```

✓ 1m 34s completed at 1:41 PM

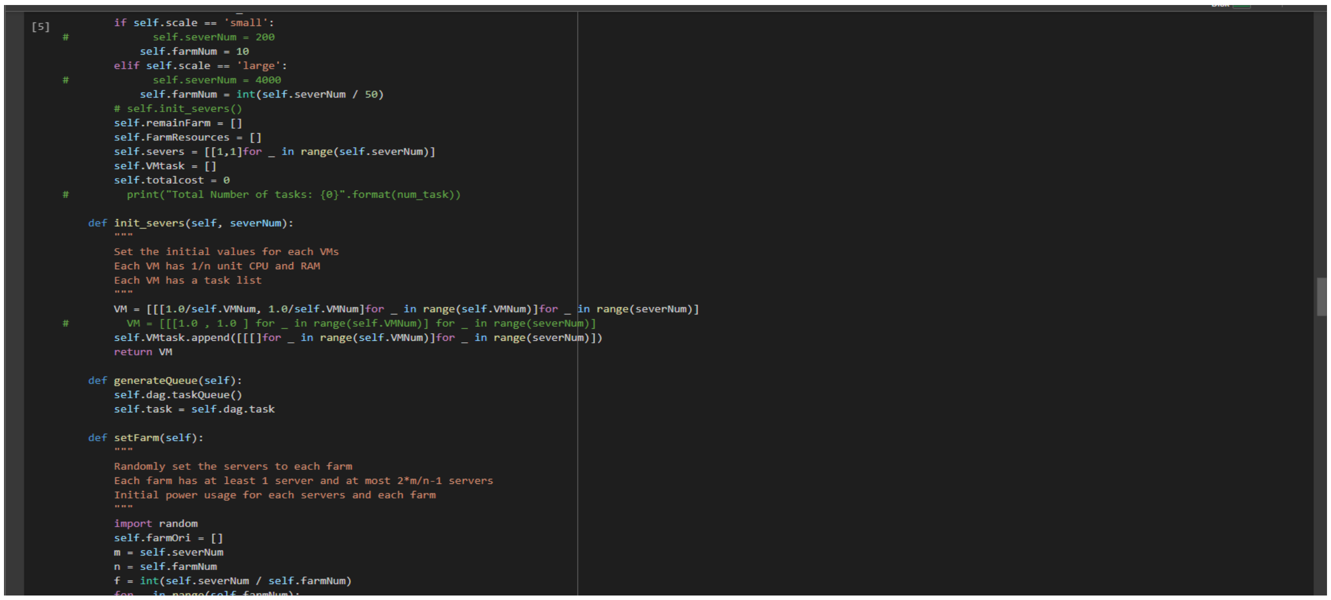


```
[5]     def updateStatus(self, task):
            """
            Given jobid and taskid, change status of all tasks that depend on it
            If the task with "-1" status, reject this tasks' all child tasks
            If the task with "0" status, remove it from all child tasks
            """
#             job_i, task_i = self.findTask(task.jobID, task.index)
#             if job_i == -1 or task_i == -1:
#                 print("WRONG: The task with jobID: ", task.jobID, " and taskID: ", task.index, " not exist.")
#                 return
#             job = self.job[job_i]
#             task = job[task_i]
            if task.status == -1:
                self.rejTask(task)
#             elif task.status == 0:
#                 self.rmParent(task, task_i, job)

        def initTask(self):
            """
            run readfile and buildDAG functions
            """
            self.readfile()
            self.buildDAG()

        def taskQueue(self):
            """
            Build the task ready queue
            Just put the one whose status is 1
            and whose parent are all finished
            """
            for job in self.job:
#                 num_task = len(job)
#                 while num_task > 0:
                for task in job:
                    if task.status == 1 and self.hasParent(task) == False:
                        self.task.append(task)
#                         task.status = 0
#                         self.updateStatus(task)
#                         num_task -= 1
#             for t in self.task:
#                 t.status = 1
```

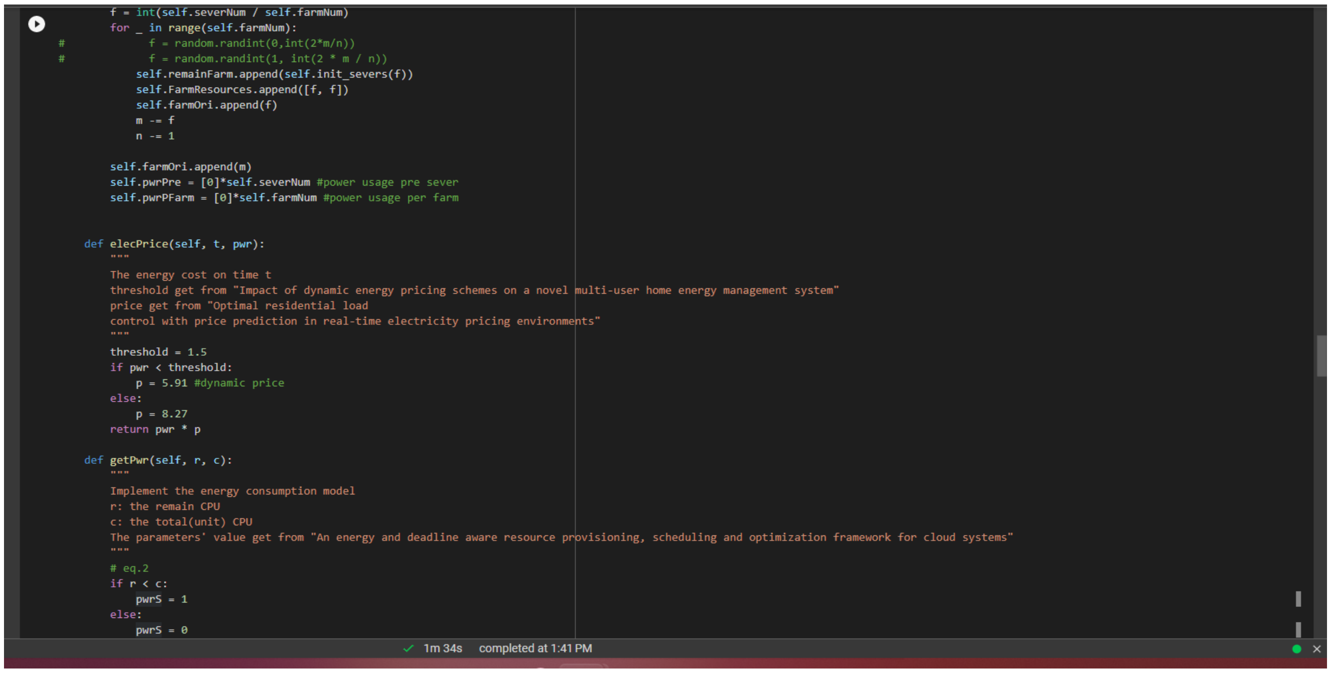
✓ 1m 34s completed at 1:41 PM



```
[5]     def printTask(self):
            """
            Print tasks which are in task queue info
            """
            for j in self.task:
                print(j.jobID, ",", j.index, ",", j.status, ",", len(j.parent))


    class environment(object):
        """docstring for environment
        the environment of RP/TS processor
        read the task from txt file
        calculate the Reward Function
        interface with DQN and baseline
        """
        def __init__(self, scale, fname, num_task, num_server):
            """
            initial the variable
            We assume each server has 10 VM
            For small-scale problems:
                200 servers
                10 server farms
            For large-scale problems:
                4000 servers
                70 server farms
            All servers have unit CPU, RAM, and Disk space
            """
            self.scale = scale
            self.fname = fname
            self.task = []
            self.dag = DAG(self.fname, num_task)
            self.VMNum = 5
            self.rej = 0
            self.num_task = num_task
            self.severNum = num_server
            if self.scale == 'small':
#                 self.severNum = 200
                self.farmNum = 10
            elif self.scale == 'large':
#                 self.severNum = 4000
```

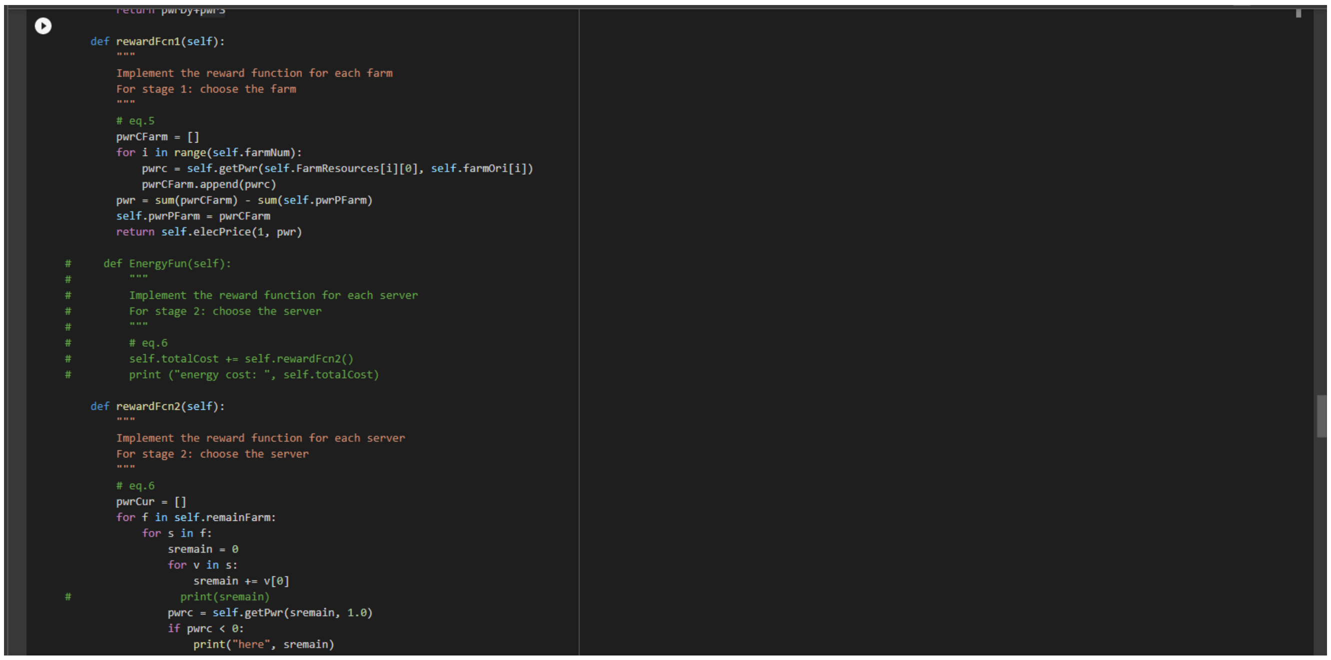


```
[5]         if self.scale == 'small':
#               self.severNum = 200
                self.farmNum = 10
            elif self.scale == 'large':
#               self.severNum = 4000
                self.farmNum = int(self.severNum / 50)
            # self.init_severs()
            self.remainFarm = []
            self.FarmResources = []
            self.severs = [[1,1]for _ in range(self.severNum)]
            self.VMtask = []
            self.totalcost = 0
#             print("Total Number of tasks: {0}".format(num_task))

        def init_severs(self, severNum):
            """
            Set the initial values for each VMs
            Each VM has 1/n unit CPU and RAM
            Each VM has a task list
            """
            VM = [[[1.0/self.VMNum, 1.0/self.VMNum]for _ in range(self.VMNum)]for _ in range(severNum)]
#             VM = [[[1.0 , 1.0 ] for _ in range(self.VMNum)] for _ in range(severNum)]
            self.VMtask.append([[[]for _ in range(self.VMNum)]for _ in range(severNum)])
            return VM

        def generateQueue(self):
            self.dag.taskQueue()
            self.task = self.dag.task

        def setFarm(self):
            """
            Randomly set the servers to each farm
            Each farm has at least 1 server and at most 2*m/n-1 servers
            Initial power usage for each servers and each farm
            """
            import random
            self.farmOri = []
            m = self.severNum
            n = self.farmNum
            f = int(self.severNum / self.farmNum)
            for _ in range(self.farmNum):
```

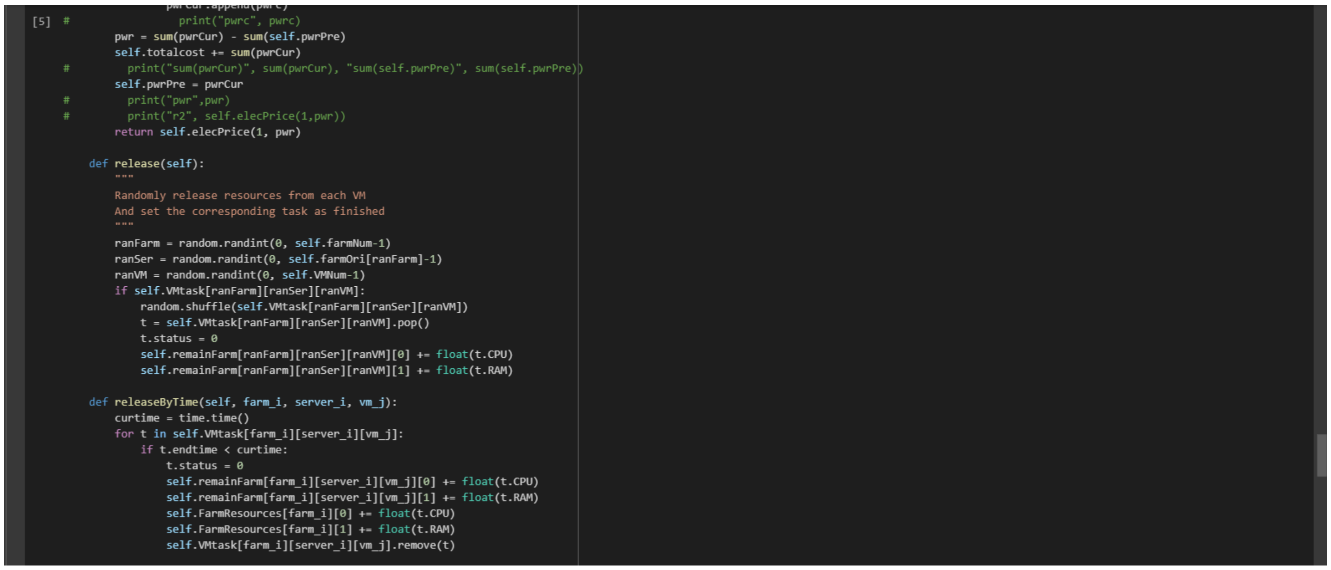


```
            f = int(self.severNum / self.farmNum)
            for _ in range(self.farmNum):
#               f = random.randint(0,int(2*m/n))
#               f = random.randint(1, int(2 * m / n))
                self.remainFarm.append(self.init_severs(f))
                self.FarmResources.append([f, f])
                self.farmOri.append(f)
                m -= f
                n -= 1

            self.farmOri.append(m)
            self.pwrPre = [0]*self.severNum #power usage pre sever
            self.pwrPFarm = [0]*self.farmNum #power usage per farm


        def elecPrice(self, t, pwr):
            """
            The energy cost on time t
            threshold get from "Impact of dynamic energy pricing schemes on a novel multi-user home energy management system"
            price get from "Optimal residential load
            control with price prediction in real-time electricity pricing environments"
            """
            threshold = 1.5
            if pwr < threshold:
                p = 5.91 #dynamic price
            else:
                p = 8.27
            return pwr * p

        def getPwr(self, r, c):
            """
            Implement the energy consumption model
            r: the remain CPU
            c: the total(unit) CPU
            The parameters' value get from "An energy and deadline aware resource provisioning, scheduling and optimization framework for cloud systems"
            """
            # eq.2
            if r < c:
                pwrS = 1
            else:
                pwrS = 0
```

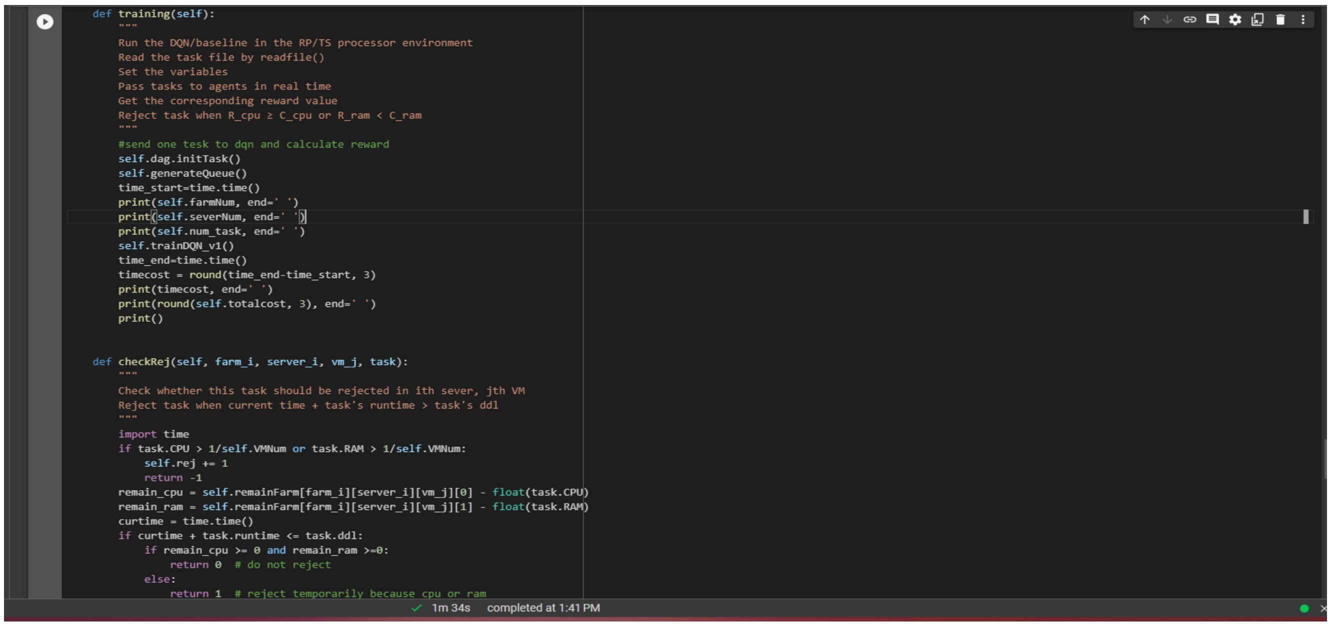
✓ 1m 34s completed at 1:41 PM



```python
        return pwrDy+pwrS

    def rewardFcn1(self):
        """
        Implement the reward function for each farm
        For stage 1: choose the farm
        """
        # eq.5
        pwrCFarm = []
        for i in range(self.farmNum):
            pwrc = self.getPwr(self.FarmResources[i][0], self.farmOri[i])
            pwrCFarm.append(pwrc)
        pwr = sum(pwrCFarm) - sum(self.pwrPFarm)
        self.pwrPFarm = pwrCFarm
        return self.elecPrice(1, pwr)

#     def EnergyFun(self):
#         """
#         Implement the reward function for each server
#         For stage 2: choose the server
#         """
#         # eq.6
#         self.totalCost += self.rewardFcn2()
#         print ("energy cost: ", self.totalCost)

    def rewardFcn2(self):
        """
        Implement the reward function for each server
        For stage 2: choose the server
        """
        # eq.6
        pwrCur = []
        for f in self.remainFarm:
            for s in f:
                sremain = 0
                for v in s:
                    sremain += v[0]
#                 print(sremain)
                pwrc = self.getPwr(sremain, 1.0)
                if pwrc < 0:
                    print("here", sremain)
```

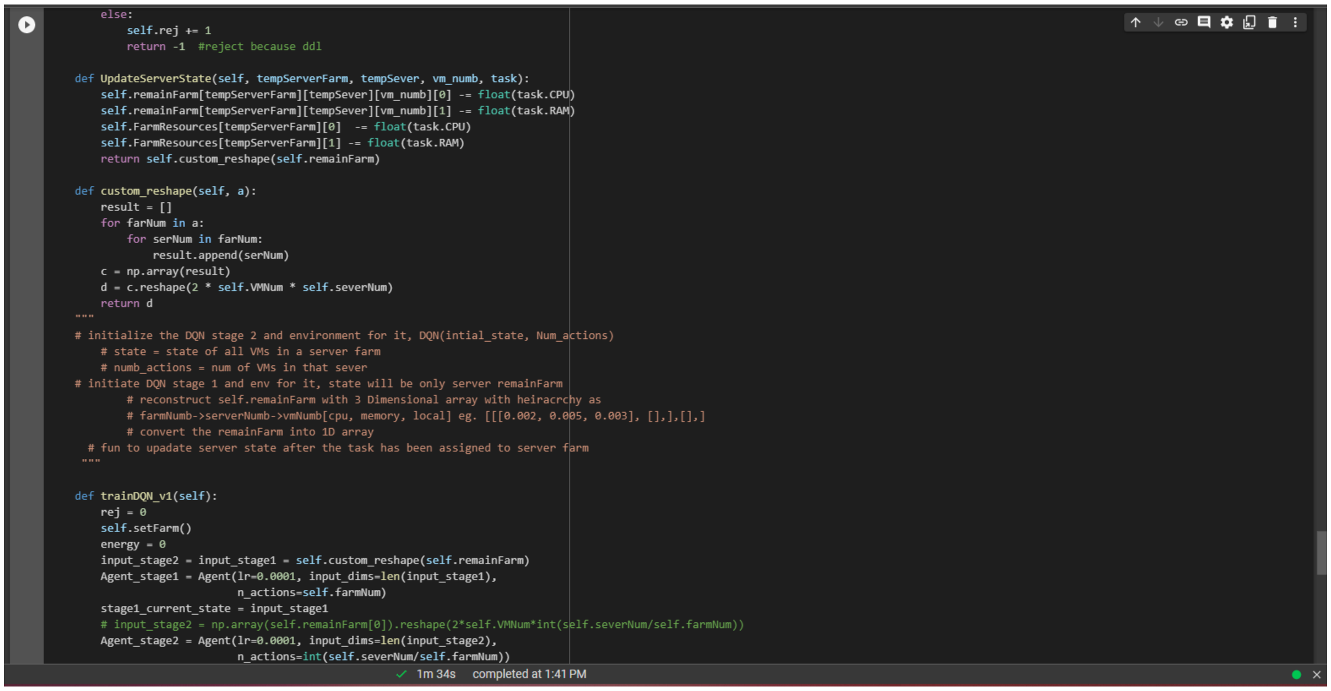


```python
                pwrCur.append(pwrc)
[5] #                 print("pwrc", pwrc)
        pwr = sum(pwrCur) - sum(self.pwrPre)
        self.totalcost += sum(pwrCur)
#         print("sum(pwrCur)", sum(pwrCur), "sum(self.pwrPre)", sum(self.pwrPre))
        self.pwrPre = pwrCur
#         print("pwr",pwr)
#         print("r2", self.elecPrice(1,pwr))
        return self.elecPrice(1, pwr)

    def release(self):
        """
        Randomly release resources from each VM
        And set the corresponding task as finished
        """
        ranFarm = random.randint(0, self.farmNum-1)
        ranSer = random.randint(0, self.farmOri[ranFarm]-1)
        ranVM = random.randint(0, self.VMNum-1)
        if self.VMtask[ranFarm][ranSer][ranVM]:
            random.shuffle(self.VMtask[ranFarm][ranSer][ranVM])
            t = self.VMtask[ranFarm][ranSer][ranVM].pop()
            t.status = 0
            self.remainFarm[ranFarm][ranSer][ranVM][0] += float(t.CPU)
            self.remainFarm[ranFarm][ranSer][ranVM][1] += float(t.RAM)

    def releaseByTime(self, farm_i, server_i, vm_j):
        curtime = time.time()
        for t in self.VMtask[farm_i][server_i][vm_j]:
            if t.endtime < curtime:
                t.status = 0
                self.remainFarm[farm_i][server_i][vm_j][0] += float(t.CPU)
                self.remainFarm[farm_i][server_i][vm_j][1] += float(t.RAM)
                self.FarmResources[farm_i][0] += float(t.CPU)
                self.FarmResources[farm_i][1] += float(t.RAM)
                self.VMtask[farm_i][server_i][vm_j].remove(t)
```



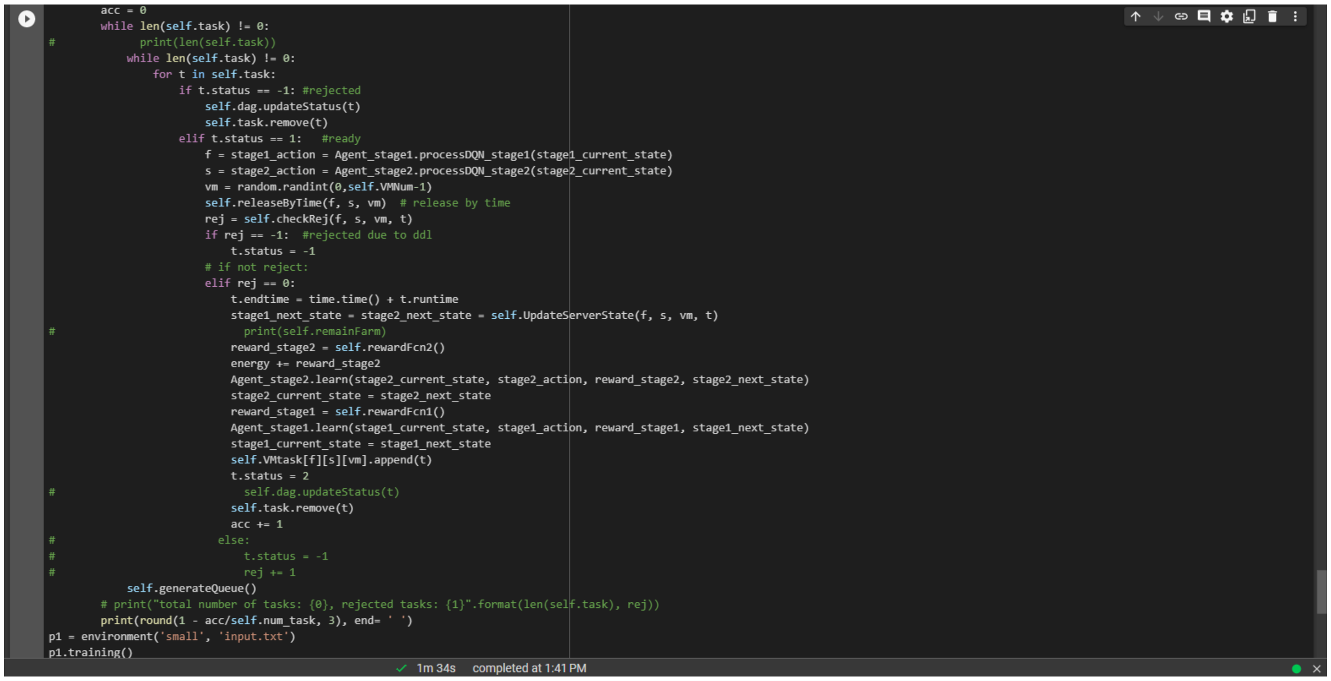
```python
    def training(self):
        """
        Run the DQN/baseline in the RP/TS processor environment
        Read the task file by readfile()
        Set the variables
        Pass tasks to agents in real time
        Get the corresponding reward value
        Reject task when R_cpu ≥ C_cpu or R_ram < C_ram
        """
        #send one tesk to dqn and calculate reward
        self.dag.initTask()
        self.generateQueue()
        time_start=time.time()
        print(self.farmNum, end=' ')
        print(self.severNum, end=' ')
        print(self.num_task, end=' ')
        self.trainDQN_v1()
        time_end=time.time()
        timecost = round(time_end-time_start, 3)
        print(timecost, end=' ')
        print(round(self.totalcost, 3), end=' ')
        print()


    def checkRej(self, farm_i, server_i, vm_j, task):
        """
        Check whether this task should be rejected in ith sever, jth VM
        Reject task when current time + task's runtime > task's ddl
        """
        import time
        if task.CPU > 1/self.VMNum or task.RAM > 1/self.VMNum:
            self.rej += 1
            return -1
        remain_cpu = self.remainFarm[farm_i][server_i][vm_j][0] - float(task.CPU)
        remain_ram = self.remainFarm[farm_i][server_i][vm_j][1] - float(task.RAM)
        curtime = time.time()
        if curtime + task.runtime <= task.ddl:
            if remain_cpu >= 0 and remain_ram >=0:
                return 0  # do not reject
            else:
                return 1  # reject temporarily because cpu or ram
```

✓ 1m 34s completed at 1:41 PM



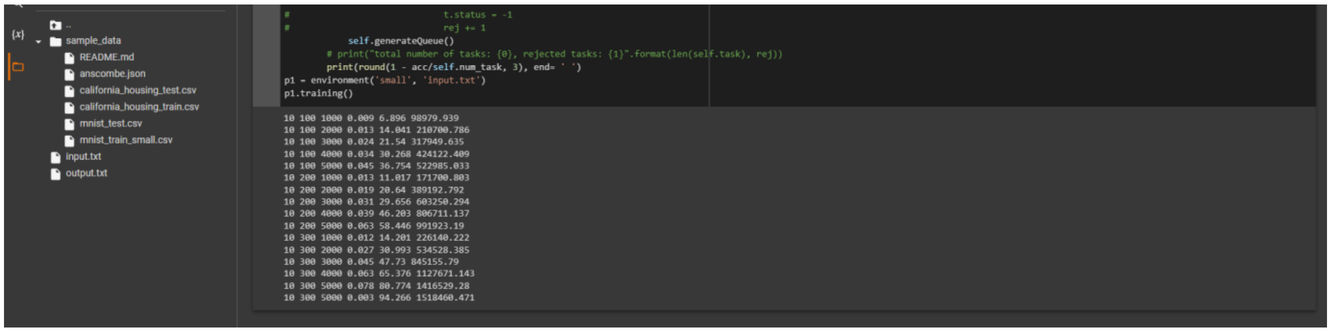
```
        else:
            self.rej += 1
            return -1  #reject because ddl

    def UpdateServerState(self, tempServerFarm, tempSever, vm_numb, task):
        self.remainFarm[tempServerFarm][tempSever][vm_numb][0] -= float(task.CPU)
        self.remainFarm[tempServerFarm][tempSever][vm_numb][1] -= float(task.RAM)
        self.FarmResources[tempServerFarm][0]  -= float(task.CPU)
        self.FarmResources[tempServerFarm][1] -= float(task.RAM)
        return self.custom_reshape(self.remainFarm)

    def custom_reshape(self, a):
        result = []
        for farNum in a:
            for serNum in farNum:
                result.append(serNum)
        c = np.array(result)
        d = c.reshape(2 * self.VMNum * self.severNum)
        return d
    """
    # initialize the DQN stage 2 and environment for it, DQN(intial_state, Num_actions)
        # state = state of all VMs in a server farm
        # numb_actions = num of VMs in that sever
    # initiate DQN stage 1 and env for it, state will be only server remainFarm
            # reconstruct self.remainFarm with 3 Dimensional array with heiracrchy as
            # farmNumb->serverNumb->vmNumb[cpu, memory, local] eg. [[[0.002, 0.005, 0.003], [],],[],]
            # convert the remainFarm into 1D array
      # fun to upadate server state after the task has been assigned to server farm
     """

    def trainDQN_v1(self):
        rej = 0
        self.setFarm()
        energy = 0
        input_stage2 = input_stage1 = self.custom_reshape(self.remainFarm)
        Agent_stage1 = Agent(lr=0.0001, input_dims=len(input_stage1),
                             n_actions=self.farmNum)
        stage1_current_state = input_stage1
        # input_stage2 = np.array(self.remainFarm[0]).reshape(2*self.VMNum*int(self.severNum/self.farmNum))
        Agent_stage2 = Agent(lr=0.0001, input_dims=len(input_stage2),
                             n_actions=int(self.severNum/self.farmNum))
```

✓ 1m 34s completed at 1:41 PM



```
        acc = 0
        while len(self.task) != 0:
#             print(len(self.task))
            while len(self.task) != 0:
                for t in self.task:
                    if t.status == -1: #rejected
                        self.dag.updateStatus(t)
                        self.task.remove(t)
                    elif t.status == 1:   #ready
                        f = stage1_action = Agent_stage1.processDQN_stage1(stage1_current_state)
                        s = stage2_action = Agent_stage2.processDQN_stage2(stage2_current_state)
                        vm = random.randint(0,self.VMNum-1)
                        self.releaseByTime(f, s, vm)  # release by time
                        rej = self.checkRej(f, s, vm, t)
                        if rej == -1:  #rejected due to ddl
                            t.status = -1
                        # if not reject:
                        elif rej == 0:
                            t.endtime = time.time() + t.runtime
                            stage1_next_state = stage2_next_state = self.UpdateServerState(f, s, vm, t)
#                             print(self.remainFarm)
                            reward_stage2 = self.rewardFcn2()
                            energy += reward_stage2
                            Agent_stage2.learn(stage2_current_state, stage2_action, reward_stage2, stage2_next_state)
                            stage2_current_state = stage2_next_state
                            reward_stage1 = self.rewardFcn1()
                            Agent_stage1.learn(stage1_current_state, stage1_action, reward_stage1, stage1_next_state)
                            stage1_current_state = stage1_next_state
                            self.VMtask[f][s][vm].append(t)
                            t.status = 2
#                             self.dag.updateStatus(t)
                            self.task.remove(t)
                            acc += 1
#                         else:
#                             t.status = -1
#                             rej += 1
            self.generateQueue()
        # print("total number of tasks: {0}, rejected tasks: {1}".format(len(self.task), rej))
        print(round(1 - acc/self.num_task, 3), end= ' ')
p1 = environment('small', 'input.txt')
p1.training()
```

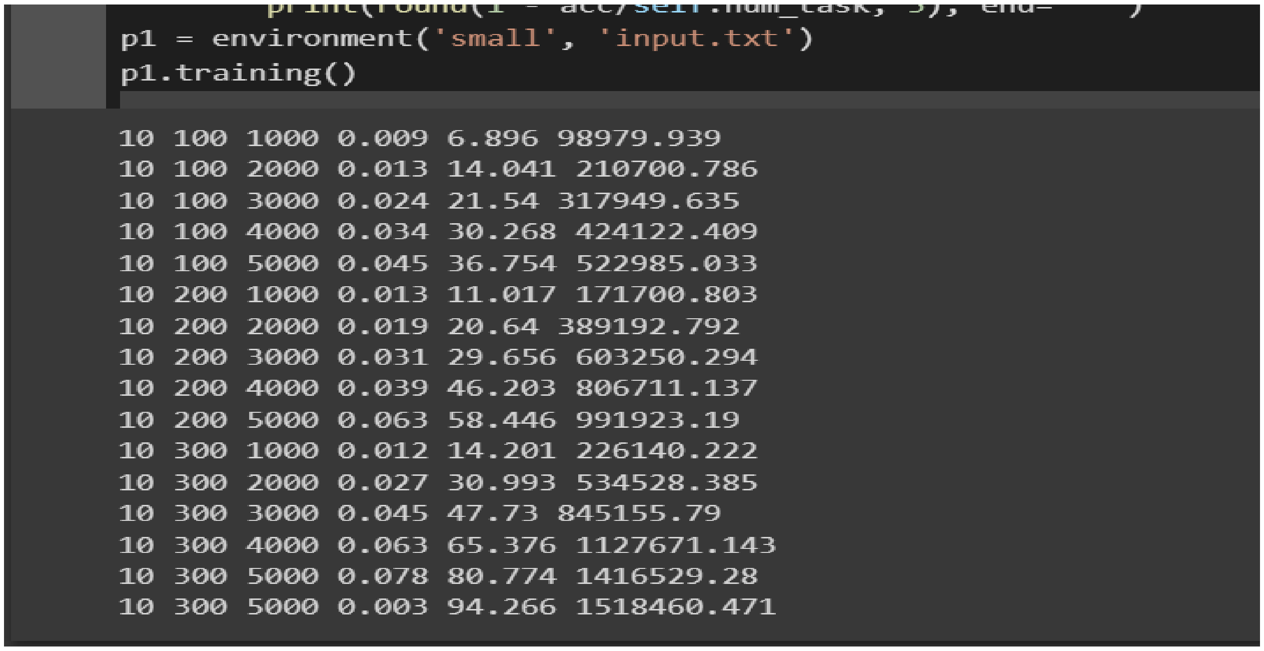
✓ 1m 34s completed at 1:41 PM



```
#                             t.status = -1
#                             rej += 1
            self.generateQueue()
        # print("total number of tasks: {0}, rejected tasks: {1}".format(len(self.task), rej))
        print(round(1 - acc/self.num_task, 3), end= ' ')
p1 = environment('small', 'input.txt')
p1.training()
```

```
10 100 1000 0.009 6.896 98979.939
10 100 2000 0.013 14.041 210700.786
10 100 3000 0.024 21.54 317949.635
10 100 4000 0.034 30.268 424122.409
10 100 5000 0.045 36.754 522985.033
10 200 1000 0.013 11.017 171700.803
10 200 2000 0.019 20.64 389192.792
10 200 3000 0.031 29.656 603250.294
10 200 4000 0.039 46.203 806711.137
10 200 5000 0.063 58.446 991923.19
10 300 1000 0.012 14.201 226140.222
10 300 2000 0.027 30.993 534528.385
10 300 3000 0.045 47.73 845155.79
10 300 4000 0.063 65.376 1127671.143
10 300 5000 0.078 80.774 1416529.28
10 300 5000 0.003 94.266 1518460.471
```

OUTPUT:



```
        print(round(1 - acc/self.num_task, 3), end=    )
p1 = environment('small', 'input.txt')
p1.training()
```

```
10 100 1000 0.009 6.896 98979.939
10 100 2000 0.013 14.041 210700.786
10 100 3000 0.024 21.54 317949.635
10 100 4000 0.034 30.268 424122.409
10 100 5000 0.045 36.754 522985.033
10 200 1000 0.013 11.017 171700.803
10 200 2000 0.019 20.64 389192.792
10 200 3000 0.031 29.656 603250.294
10 200 4000 0.039 46.203 806711.137
10 200 5000 0.063 58.446 991923.19
10 300 1000 0.012 14.201 226140.222
10 300 2000 0.027 30.993 534528.385
10 300 3000 0.045 47.73 845155.79
10 300 4000 0.063 65.376 1127671.143
10 300 5000 0.078 80.774 1416529.28
10 300 5000 0.003 94.266 1518460.471
```

8. CONCLUSION AND FUTURE ENHANCEMENTS

Cloud service providers spend a lot of money on electricity each year, but these costs can be cut by carefully controlling resources through job scheduling and resource provisioning. In this project, we built a two-stage deep Q network based on deep learning, with an agent that can be trained to choose the best candidate for the task at hand and experience decreasing rejection rates. In the first stage, the agent chooses the server farm, and in the second stage, we choose the server. The DQN-based job scheduler for computing systems based on fog clouds was successful in reaching its goal of energy efficiency

There are several potential future enhancements for a deep learning-based scheduler for cloud and fog computing environments, including:

Incorporating Reinforcement Learning: The addition of reinforcement learning techniques to the scheduler can enable it to learn from its environment and make decisions based on the feedback it receives.

Multi-Objective Optimization: Multi-objective optimization can help the scheduler to optimize multiple objectives simultaneously, such as minimizing power consumption, maximizing resource utilization, and improving response time.

Hybrid Scheduling: Hybrid scheduling, which combines both centralized and distributed scheduling mechanisms, can be used to leverage the benefits of both approaches.

Resource Allocation: Resource allocation algorithms can be incorporated to allocate resources to tasks more efficiently, based on factors such as task requirements and resource availability.

Federated Learning: The integration of federated learning techniques can enable the scheduler to learn from data distributed across multiple devices and locations, without the need for data to be centrally stored.

Edge Intelligence: Edge intelligence techniques can be used to enable the scheduler to make decisions at the edge of the network, closer to the data source, which can improve performance and reduce latency.

Explainability and Interpretability: The addition of explainability and interpretability mechanisms can help users understand how the scheduler makes decisions and provide insights into how the system is functioning.

Overall, the future enhancements for a deep learning-based scheduler for cloud and fog computing environments are focused on improving its ability to optimize resource utilization, reduce latency, improve performance, and make decisions in a more efficient and effective manner

9. REFERENCES

[1] Sheng, S., Chen, P., Chen, Z., Wu, L. and Yao, Y., 2021. Deep reinforcement learning-based task scheduling in IoT edge computing. Sensors, 21(5), p.1666.

[2] Movahedi, Z. and Defude, B., 2021. An efficient population-based multi-objective task scheduling approach in fog computing systems. Journal of Cloud Computing, 10(1), pp.1-31.

[3] Tuli, S., Ilager, S., Ramamohanarao, K. and Buyya, R., 2020. Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks. IEEE transactions on mobile computing.

[4] Louail, M., Esseghir, M. and Merghem-Boulahia, L., 2020, October. Dynamic task scheduling for fog nodes based on deadline constraints and task frequency for smart factories. In 2020 11th International Conference on Network of the Future (NoF) (pp. 16-22). IEEE.

[5] Singh, S.P., Nayyar, A., Kaur, H. and Singla, A., 2019. Dynamic task scheduling using balanced VM allocation policy for fog computing platforms. Scalable Computing: Practice and Experience, 20(2), pp.433-456.

[6] Hossain, M.R., Whaiduzzaman, M., Barros, A., Tuly, S.R., Mahi, M.J.N., Roy, S., Fidge, C. and Buyya, R., 2021. A scheduling-based dynamic fog computing framework for augmenting resource utilization. Simulation Modelling Practice and Theory, 111, p.102336.

[7] Bitam, S., Zeadally, S. and Mellouk, A., 2018. Fog computing job scheduling optimization based on bees swarm. Enterprise Information Systems, 12(4), pp.373-397.

[8] Choudhari, T., Moh, M. and Moh, T.S., 2018, March. Prioritized task scheduling in fog computing. In Proceedings of the ACMSE 2018 Conference (pp. 1-8).

[9] Fellir, F., El Attar, A., Nafil, K. and Chung, L., 2020, February. A multi-Agent based model for task scheduling in cloud-fog computing platform. In 2020 IEEE international conference on informatics, IoT, and enabling technologies (ICIoT) (pp. 377-382). IEEE.

[10] Nikoui, T.S., Balador, A., Rahmani, A.M. and Bakhshi, Z., 2020, June. Cost-aware task scheduling in fog-cloud environment. In 2020 CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST) (pp. 1-8). IEEE

THE END