Minimizing Costs in Energy Consumption of a Data Center

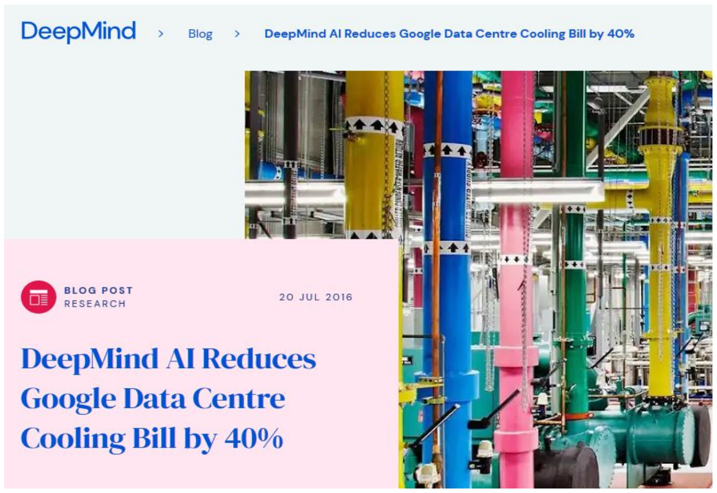
(Using Deep Q-Learning)

Abstract :

Tech companies have tremendous cost coming from their data center. A large portion of that is the cooling required to keep their servers operating with optimal performance. So, the problem to solve is *minimizing* the energy consumption of cooling the servers to keep them in their optimal range which will as a result minimize the electricity cost and save tremendous amounts of money and therefore, millions of dollars can be saved.

Inspiration :

In 2016, DeepMind AI minimized a big part of Google's cost by reducing Google Data Centre Cooling Bill by 40% using their DQN AI model (Deep Q-Learning). In our project we will try and work on this very problem. We will set up our own server environment, and we will build an AI that will be controlling the cooling/heating of the server so that it stays in an optimal range of temperatures while saving the maximum energy, therefore minimizing the costs.

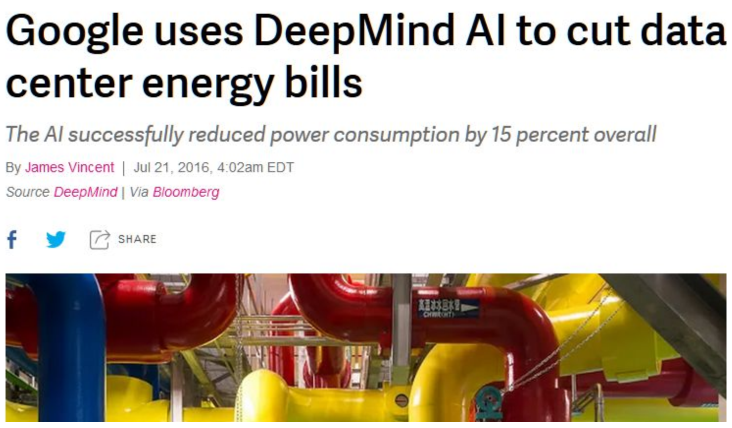

DeepMind > Blog > DeepMind AI Reduces Google Data Centre Cooling Bill by 40%

BLOG POST
RESEARCH

20 JUL 2016

DeepMind AI Reduces Google Data Centre Cooling Bill by 40%

After accounting for "electrical losses and other non-cooling inefficiencies," this 40 percent reduction translated into a 15 percent reduction in overall power saving, says Google. Considering that the company used some 4,402,836 MWh of electricity in 2014 (equivalent to the amount of energy consumed by 366,903 US households), this 15 percent will translate into savings of hundreds of millions of dollars over the years.


Google uses DeepMind AI to cut data center energy bills

The AI successfully reduced power consumption by 15 percent overall

By James Vincent | Jul 21, 2016, 4:02am EDT

Source DeepMind | Via Bloomberg

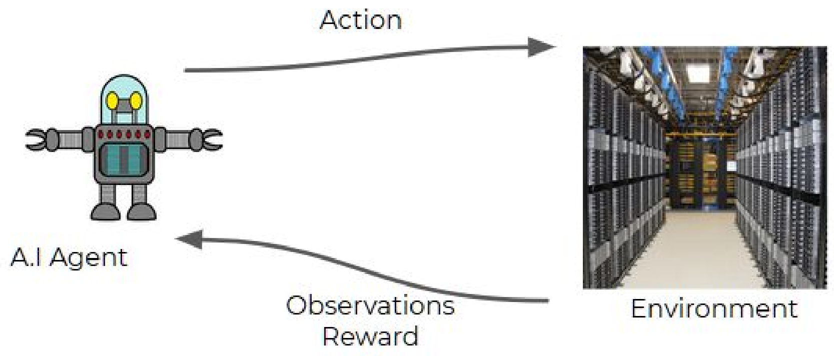
SHARE

1. Introduction :

We will set up our own server environment, and we will build an AI that will be controlling the cooling/heating of the server so that it stays in an optimal range of temperatures while saving the maximum energy, therefore minimizing the costs. We are going to optimise this using **Deep-Q Reinforcement Learning** for the optimization problem. We'll be comparing the performance of the server's internal cooling system with our RL agent's performance in order to conclude energy savings. The internal cooling system simply brings the current temperature of the server into the optimal range bounds whereas our agent in its neural network architecture will learn weights using which it will predict a suitable action that results into less energy use.

Environment :

Before we define the states, actions and rewards, we need to explain how the server operates. At a given timestep, the server model has as input its intrinsic temperature, the number of users online and the data rate going through it. Given, the 3-tuple, the agent has to predict an action as defined in the action space. The states and action spaces are described on the following page. This RL setup is illustrated below :



The **variables** of this environment (at any minute) are given as follows :

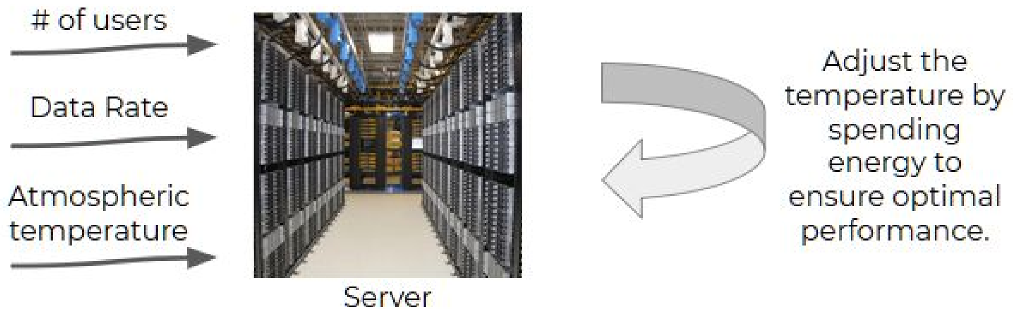
• The number of users online
• The temperature of the server
• The data transmission rate through the server

• The energy spent by the server's internal cooling system that automatically brings the server's temperature back to the optimal range whenever the server's temperature goes outside this optimal range.
• The energy spent by the agent onto the server in order to heat it or cool it.

The **state** of the server at a time t is given as the 3-tuple shown below :

$$\begin{pmatrix} \text{number of users online} \\ \text{atmospheric temperature} \\ \text{data transmission rate} \end{pmatrix}$$

Thus, the input vector will consist of three elements.The agent will take this vector as input, and will return the action to play at each timestep.


# of users
Data Rate
Atmospheric temperature
Server
Adjust the temperature by spending energy to ensure optimal performance.

The temperature of the server can be approximated as given and for simplicity purposes, we just suppose that these correlations are linear.

$$\textit{server temperature} = (\textit{atmospheric temperature}) + 1.25 \times (\textit{number of users}) + 1.25 \times (\textit{data transmission rate})$$

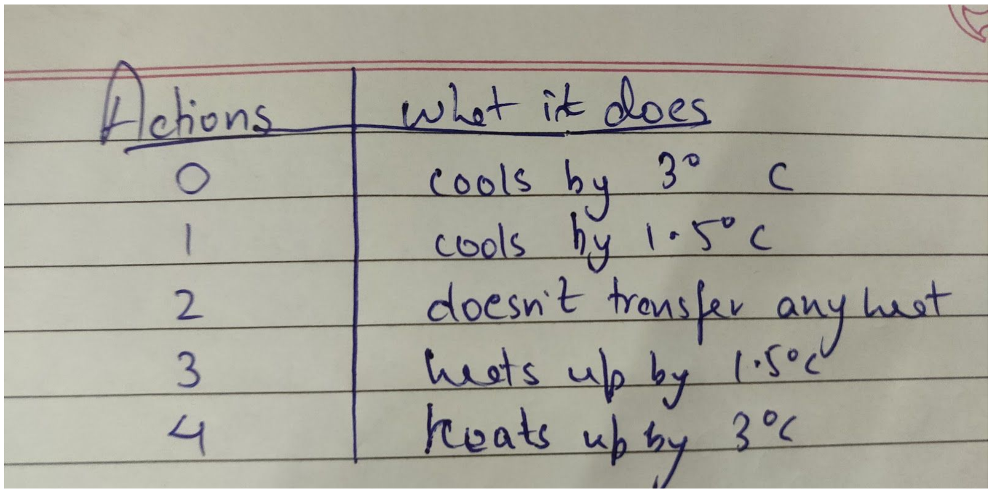
This is the case because the more the number of users online, the more processing power will be used by the server which will lead to heat dissipation and raise the temperature. Also, a high data network throughput will lead to heat dissipation which similarly raises the temperature. Different months will have varying atmospheric conditions and temperatures for example :

$$\begin{array}{r} \textit{Month} \\ \textit{Temperature}\ (\ C) \end{array} \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 5 & 7 & 11 & 18 & 20 & 25 & 28 & 22 & 20 & 18 & 10 & 7 \end{bmatrix}$$

The energy spent by a system (our agent or the server's internal cooling system) that changes the server's temperature within 1 unit of time can be approximated as proportional to change in the temperature caused i.e. $\triangle E = \alpha \triangle T$ for simplicity purposes; taking constant as 1. Thus,

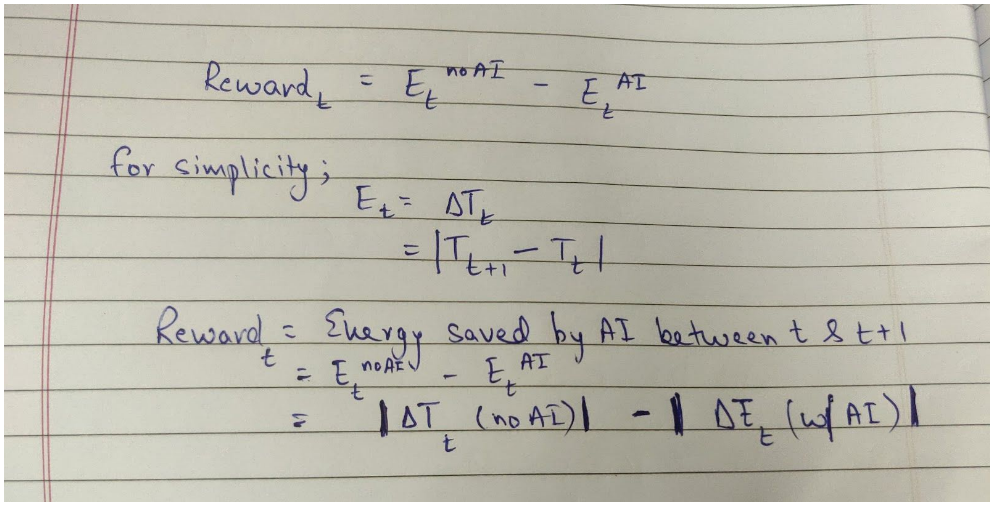
$$\triangle E = |\triangle T|$$

The actions are simply the temperature changes that the agent can cause inside the server, in order to heat it up or cool it down. In order to make our actions discrete, we will consider 5 possible temperature changes from −3 ◦ C to +3 ◦ C, so that we end up with the 5 following possible actions that the agent can play to regulate the temperature of the server :



Actions	what it does
0	cools by 3° C
1	cools by 1.5°C
2	doesn't transfer any heat
3	heats up by 1.5°C
4	heats up by 3°C

The **reward** in the RL framework which is used to train the agent is the energy difference with our agent turned on vs the internal cooling system.



$$\text{Reward}_t = E_t^{noAI} - E_t^{AI}$$

for simplicity;

$$E_t = \Delta T_t = |T_{t+1} - T_t|$$

$$\text{Reward}_t = \text{Energy saved by AI between } t \,\&\, t+1$$
$$= E_t^{noAI} - E_t^{AI}$$
$$= |\Delta T_t \text{ (no AI)}| - |\Delta E_t \text{ (w/ AI)}|$$

2. Mathematical Formulation :

The environment of the server and its interaction is modelled as a Markov Decision Process (MDP) which follows the Markov Property : The value **V(s)** is not dependent on past actions or states and just depends on the current state. The MDP is used to modelling process where the decision depends upon agent and environment can be stochastic or non-stochastic. **Bellman Equation** is used to model in the MDP which consists includes recursion of future rewards.

The Value function for non-stochastic environment is given below :

$$V(s) = MAX\ (\ R(s,a) + \gamma\ V(s')\ )$$

As an example, consider this grid world where the values of each of the states is given, and the agent upon learning this policy can navigate this MDP maze in order to get reward at **WIN** state.

Thus, the value function recursive equation is used to move towards the reward.

0.81 →	0.90 →	1.00 →	**WIN**
0.73 ↑	X	0.90 ↑	X
0.64 →	0.73 →	0.81 ↑	0.73

The value function for stochastic environment where agent decisions are probabilistic is given as follows:

$$V(s) = \max_{a}\ (\ R(s,a) + \gamma\ E[V(s')]\ )$$

$$= \max_{a}\left(\ R(s,a) + \gamma \sum P(s,a,s')\ V(s')\ \right)$$

In the MDP model, Q-learning is an extension used to determine the quality of each action for a given state. The Q-value (s,a) is dependent on both the current state and the action taken. Therefore, the maximum Q-value of a given state is its value.

$$V(s) = \max_{a} Q(s,a)$$

$$Q(s,a) = R(s,a) + \gamma \max_{a} Q(s',a')$$

Temporal Difference (TD) :

This is the difference between the predicted Q-value and the received Q-value from the environment itself. TD has to be minimised in order for our agent to learn the MDP.

$$TD(a,s) = R(s,a) + \gamma \max_{a} Q_t(s',a') - Q_{t-1}(s,a)$$

TD is like an intrinsic reward. The agent will learn the Q-values in such a way that: we use the temporal difference to reinforce the (action, state) from time t − 1 to time t, according to the following equation:

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha TD_t(a,s)$$

This is a form of Gradient Descent. If the action a(t) has a higher Q-value, the agent is more likely to choose that action and move to the next state and if an action a(t) has a lower Q-value, then the agent has a lower probability to select that action.

Deep Q-learning :

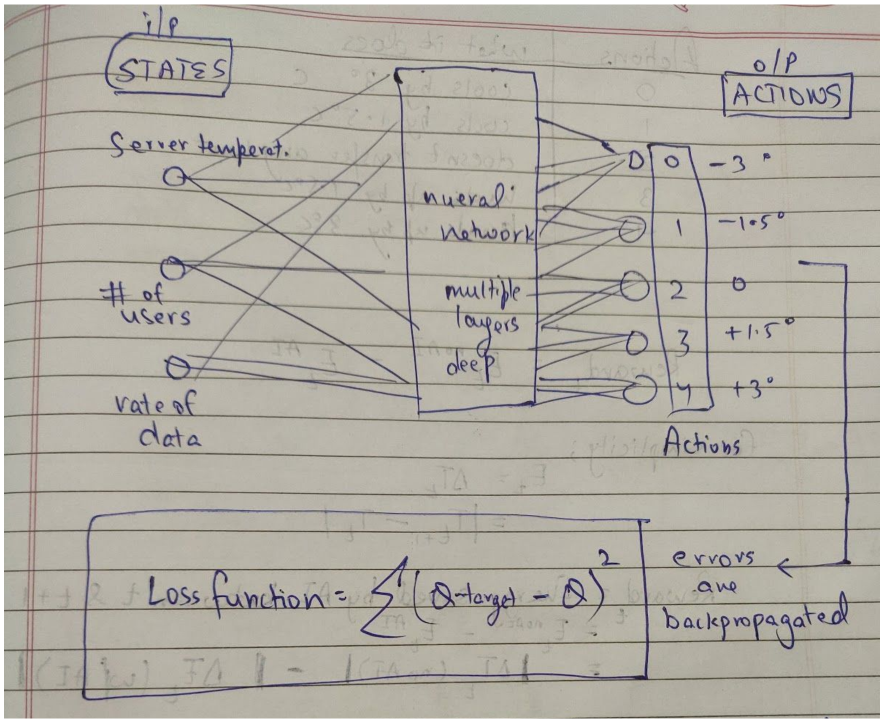
Deep Q-Learning consists of combining Q-Learning to an Artificial Neural Network that is performing regression. Inputs are encoded vectors, each one defining a state of the environment;

$$\begin{pmatrix} \text{number of users online} \\ \text{atmospheric temperature} \\ \text{data transmission rate} \end{pmatrix}$$

These inputs are propagated through the network, and agent chooses an action to play. Then the action played is the one associated with the output neuron that has the highest Q-value(argmax).

Experience Replay [5]

We only considered transitions from one state s(t) to the next state s(t+1) . The problem with this is that s(t) is mostly very correlated with s(t+1) . Thus, our agent is not learning much.



This could be improved if, instead of considering only this one previous transition, we considered the last **M** transitions where M is a large number. This pack of the last M transitions is what is called the Experience Replay. Then from this Experience Replay we take some random batches of transitions to make our updates. So that our agent learns better and does not overfit.

3. Solution Method:

Deep-Q Learning algorithm used :

Phase 1 :

In the start, in order to utilize the experience replay, we initialize a memory to store transitions for later use.The memory of the Experience Replay is initialized to an empty list M.We choose a max size of memory. For example an array of 100 transitions. We start in the first state, corresponding to a month within the year which was defined in the state space.

Repeat for each Epoch :

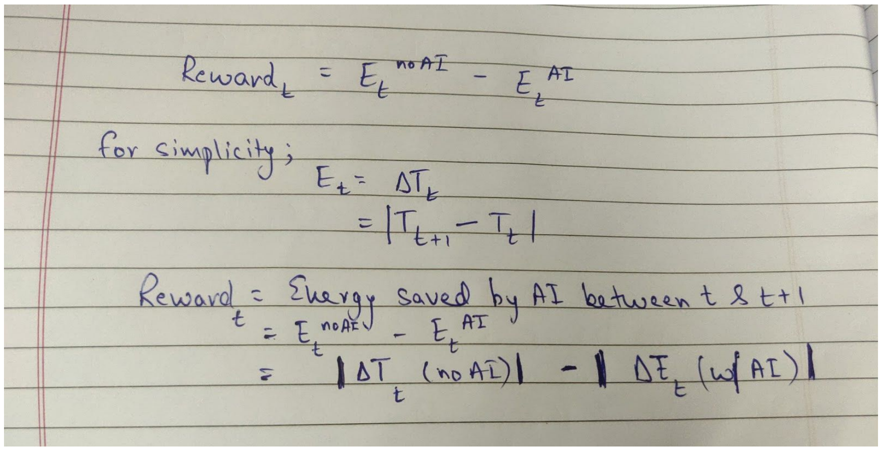
Repeat for each time instant i.e every minute until end of epoch :

- Predict the Q-value of current state.
- Perform the action that corresponds to the max of all predicted Q-value:

$$a_t = \underset{a}{argmax}\ Q\left(s_t, a_t\right)$$

- Agent receives a reward

$$r_t = E_t^{noAI} - E_t^{AI}$$



$$Reward_t = E_t^{noAI} - E_t^{AI}$$

for simplicity;

$$E_t = \Delta T_t$$
$$= |T_{t+1} - T_t|$$

$$Reward_t = \text{Energy saved by AI between } t \ \& \ t+1$$
$$= E_t^{noAI} - E_t^{AI}$$
$$= |\Delta T_t \text{ (no AI)}| - |\Delta T_t \text{ (w/ AI)}|$$

- Perform action and proceed to next state s_{t+1}
- Append the transition s_t, a_t, r_t, s_{t+1} in memory M

Phase 2 :

Training the agent on random batches to utilise experience replay.

We take a random batch B ∈ M of transitions. For all s_t, a_t, r_t, s_{t+1} transitions of the random batch B :

- Get the predictions: $Q\left(s_t, a_t\right)$
- Get the targets: $r_t + \gamma \max_a Q\left(s_{t+1}, a\right)$

- Compute the loss b/w the predictions and the targets over the whole batch B :

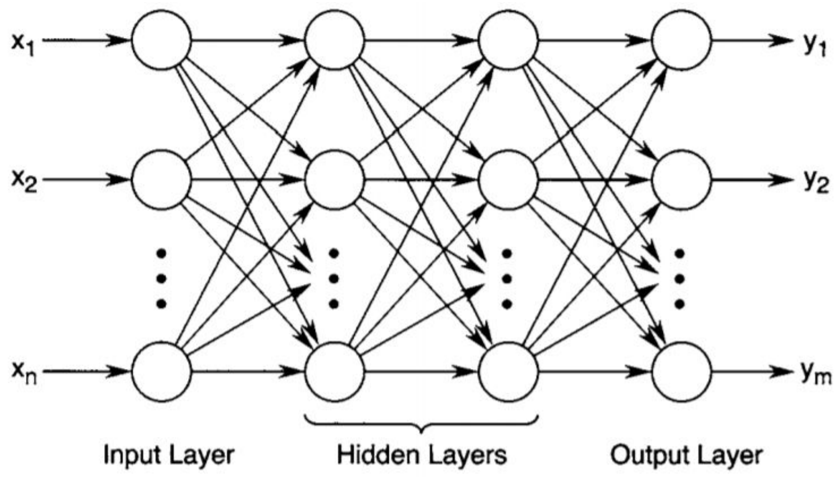
$$\text{Loss} = \frac{1}{2}\sum_{Batch}\left[\ \left\{r_t + \gamma \max_a Q\left(s_{t+1}, a\right)\right\} - \left\{ Q\left(s_t, a_t\right)\right\}\ \right]^2$$

$$= \frac{1}{2}\sum_{Batch} TD_t\left(s_t, a_t\right)^2$$

We **backpropagate** this loss error back into the neural network, through stochastic gradient descent, we update weights.

Backpropagation :

Here in, we implement the Reinforcement Learning using neural networks. Backpropagation involves optimizing the weights such that we minimise the loss function.



This is an example of a neural network architecture with 2 hidden layers.

Now in order to minimize the cost function, and optimise the weights we would do the following :

a) Randomly Initialize the weights

b) Implement forward propagation for all training examples
c) Implement the cost function
d) Implement Backpropagation on the Weights to compute partial derivatives
e) Using Adam(A type of Gradient Descent optimiser) to minimise the cost function with the weights in theta.

Now in order to perform Backpropagation on the neural network, we will take the example of a single neuron to show the Backpropagation algorithm.

$$y = \sum_{i=1}^{n} w_i x_i = \boldsymbol{x}^\top \boldsymbol{w}$$

Here, y is the vector of outputs and x is the vector of inputs. w is the vector of weights.

Now in order to minimise the cost function, we will perform Backpropagation, general MSE

$$\text{minimize} \quad \frac{1}{2} \sum_{i=1}^{p} \left(y_{d,i} - \boldsymbol{x}_{d,i}^\top \boldsymbol{w} \right)^2$$

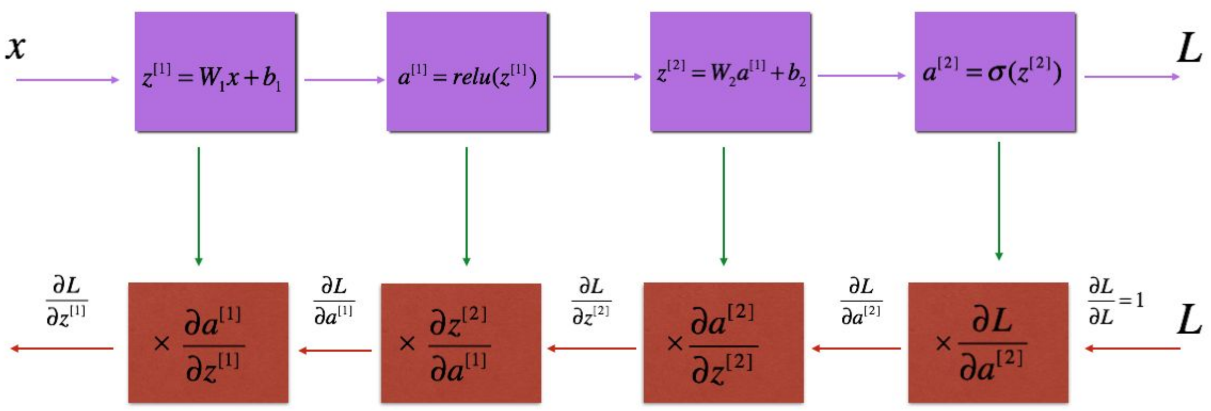
In our case, the loss function:

$$\frac{1}{2} \sum_{Batch} TD_t\left(s_t, a_t \right)^2$$

Further, we randomly initialize the weights to start with and forward propagate the neural architecture.

Now to minimise the cost function and obtain optimum weights we will perform Backpropagation. For such an architecture the Backpropagation will be done as follows. Back propagation is used to calculate the gradient of the loss function with respect to the parameters.

Let us denote the input to the jth neuron in the hidden layer by Vj and the output of the jth neuron in the hidden layer by Zj. Then, we have


x
$z^{[1]} = W_1x + b_1$
$a^{[1]} = relu(z^{[1]})$
$z^{[2]} = W_2a^{[1]} + b_2$
$a^{[2]} = \sigma(z^{[2]})$
L
$\frac{\partial L}{\partial z^{[1]}}$
$\times \frac{\partial a^{[1]}}{\partial z^{[1]}}$
$\frac{\partial L}{\partial a^{[1]}}$
$\times \frac{\partial z^{[2]}}{\partial a^{[1]}}$
$\frac{\partial L}{\partial z^{[2]}}$
$\times \frac{\partial a^{[2]}}{\partial z^{[2]}}$
$\frac{\partial L}{\partial a^{[2]}}$
$\times \frac{\partial L}{\partial a^{[2]}}$
$\frac{\partial L}{\partial L} = 1$
L

$$v_j = \sum_{i=1}^{n} w_{ji}^{h} x_i,$$
$$z_j = f_j^h \left(\sum_{i=1}^{n} w_{ji}^{h} x_i \right).$$

The output from the sth neuron of the output layer is

$$y_s = f_s^o \left(\sum_{j=1}^{l} w_{sj}^{o} z_j \right).$$

Therefore the relation between the inputs and outputs is given by

$$\begin{aligned} y_s &= f_s^o \left(\sum_{j=1}^{l} w_{sj}^{o} f_j^h(v_j) \right) \\ &= f_s^o \left(\sum_{j=1}^{l} w_{sj}^{o} f_j^h \left(\sum_{i=1}^{n} w_{ji}^{h} x_i \right) \right) \\ &= F_s(x_1, \ldots, x_n). \end{aligned}$$

Using the chain rule, we obtain

$$\frac{\partial E}{\partial w_{sj}^{o}}(\boldsymbol{w}) = -\left(y_{ds} - y_s\right) f_s^{o'}\left(\sum_{q=1}^{l} w_{sq}^{o} z_q\right) z_j,$$

where $f_s^{o'} : \mathbb{R} \to \mathbb{R}$ is the derivative of f_s^o. For simplicity of notation, we write

$$\delta_s = \left(y_{ds} - y_s\right) f_s^{o'}\left(\sum_{q=1}^{l} w_{sq}^{o} z_q\right).$$

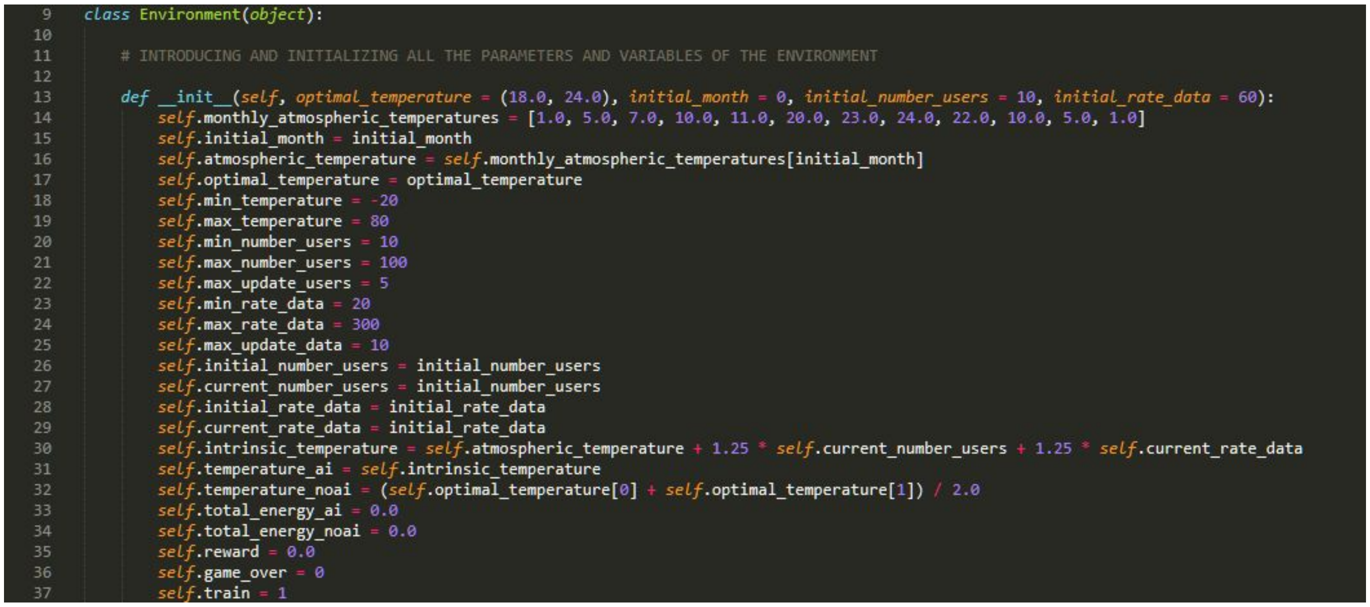
Now again using the chain rule we get

$$\begin{aligned}
v_j^{(k)} &= \sum_{i=1}^{n} w_{ji}^{h(k)} x_{di}, \\
z_j^{(k)} &= f_j^h\left(v_j^{(k)}\right), \\
y_s^{(k)} &= f_s^o\left(\sum_{q=1}^{l} w_{sq}^{o(k)} z_q^{(k)}\right), \\
\delta_s^{(k)} &= \left(y_{ds} - y_s^{(k)}\right) f_s^{o'}\left(\sum_{q=1}^{l} w_{sq}^{o(k)} z_q^{(k)}\right).
\end{aligned}$$

Now using these equations we will Backpropagate on the neural network architecture and obtain optimum weights which will minimize the cost function.

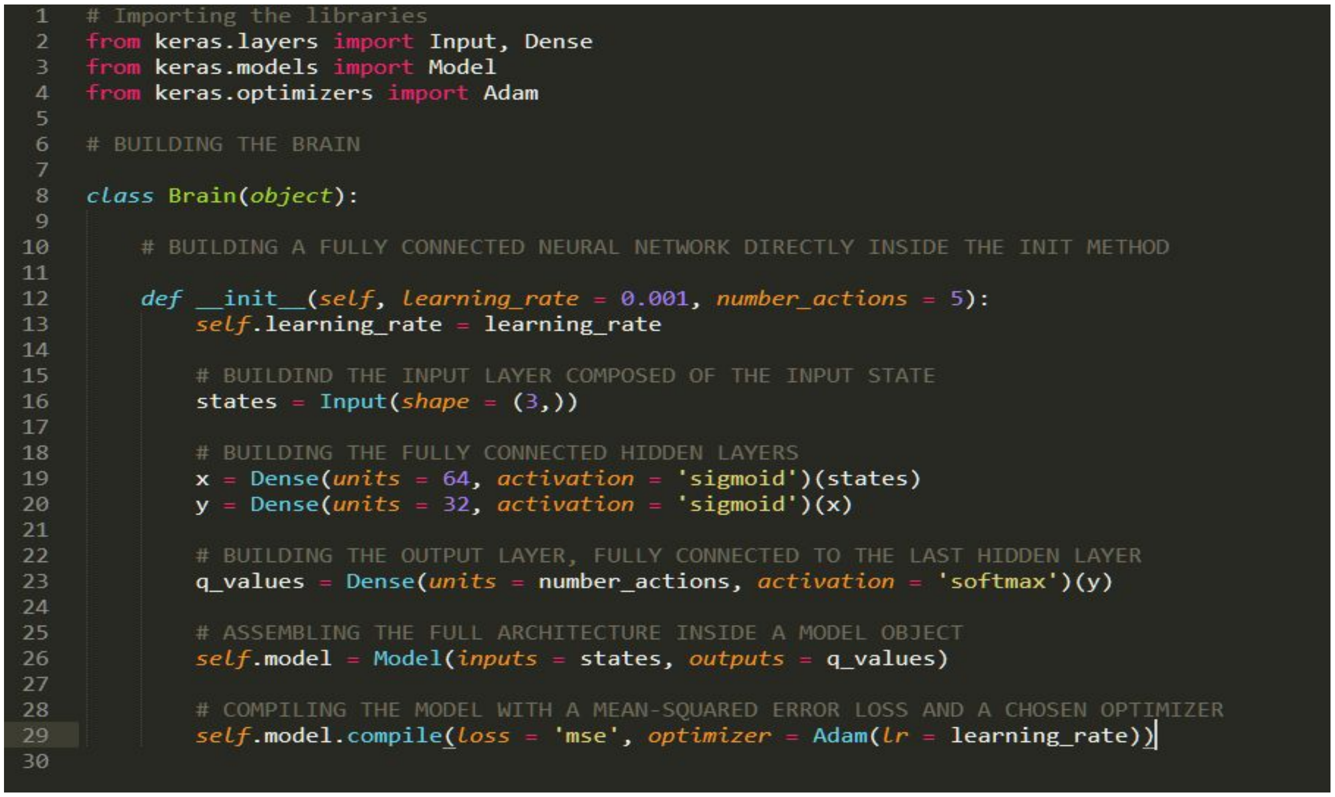
Simulation:

Building the Environment:



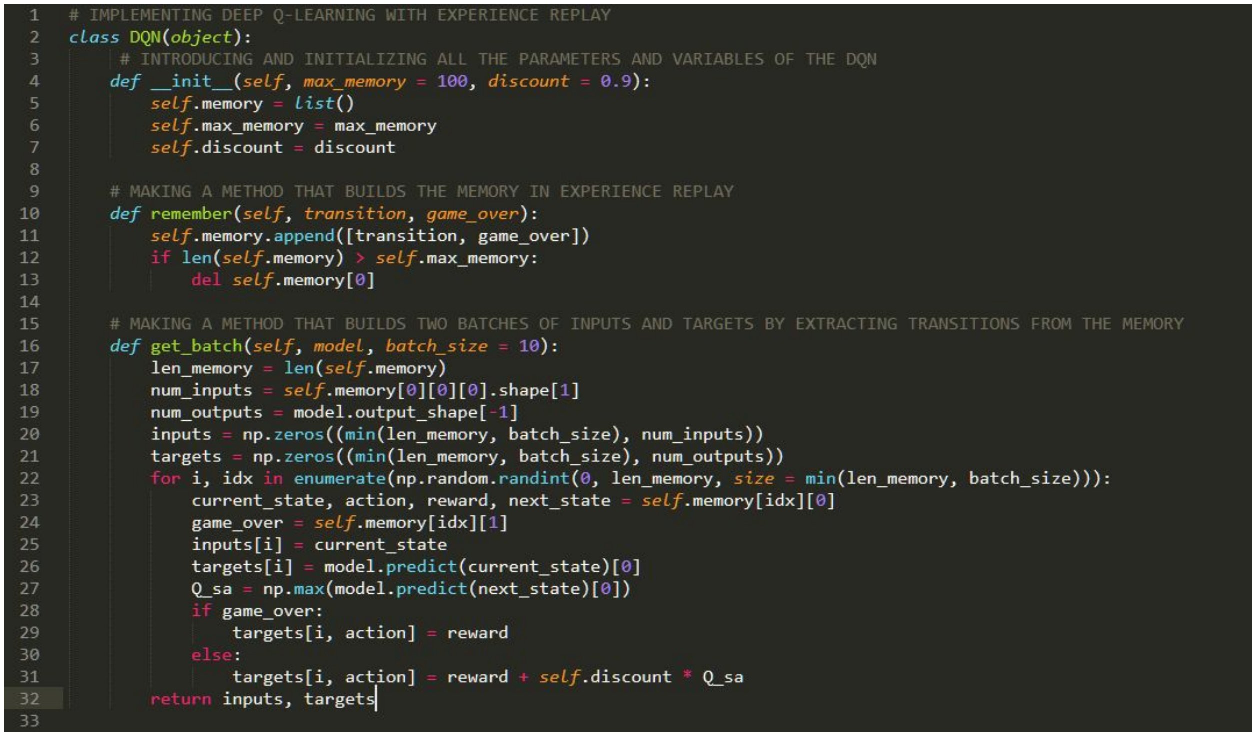
```python
class Environment(object):

    # INTRODUCING AND INITIALIZING ALL THE PARAMETERS AND VARIABLES OF THE ENVIRONMENT

    def __init__(self, optimal_temperature = (18.0, 24.0), initial_month = 0, initial_number_users = 10, initial_rate_data = 60):
        self.monthly_atmospheric_temperatures = [1.0, 5.0, 7.0, 10.0, 11.0, 20.0, 23.0, 24.0, 22.0, 10.0, 5.0, 1.0]
        self.initial_month = initial_month
        self.atmospheric_temperature = self.monthly_atmospheric_temperatures[initial_month]
        self.optimal_temperature = optimal_temperature
        self.min_temperature = -20
        self.max_temperature = 80
        self.min_number_users = 10
        self.max_number_users = 100
        self.max_update_users = 5
        self.min_rate_data = 20
        self.max_rate_data = 300
        self.max_update_data = 10
        self.initial_number_users = initial_number_users
        self.current_number_users = initial_number_users
        self.initial_rate_data = initial_rate_data
        self.current_rate_data = initial_rate_data
        self.intrinsic_temperature = self.atmospheric_temperature + 1.25 * self.current_number_users + 1.25 * self.current_rate_data
        self.temperature_ai = self.intrinsic_temperature
        self.temperature_noai = (self.optimal_temperature[0] + self.optimal_temperature[1]) / 2.0
        self.total_energy_ai = 0.0
        self.total_energy_noai = 0.0
        self.reward = 0.0
        self.game_over = 0
        self.train = 1
```

Neural Network Architecture :



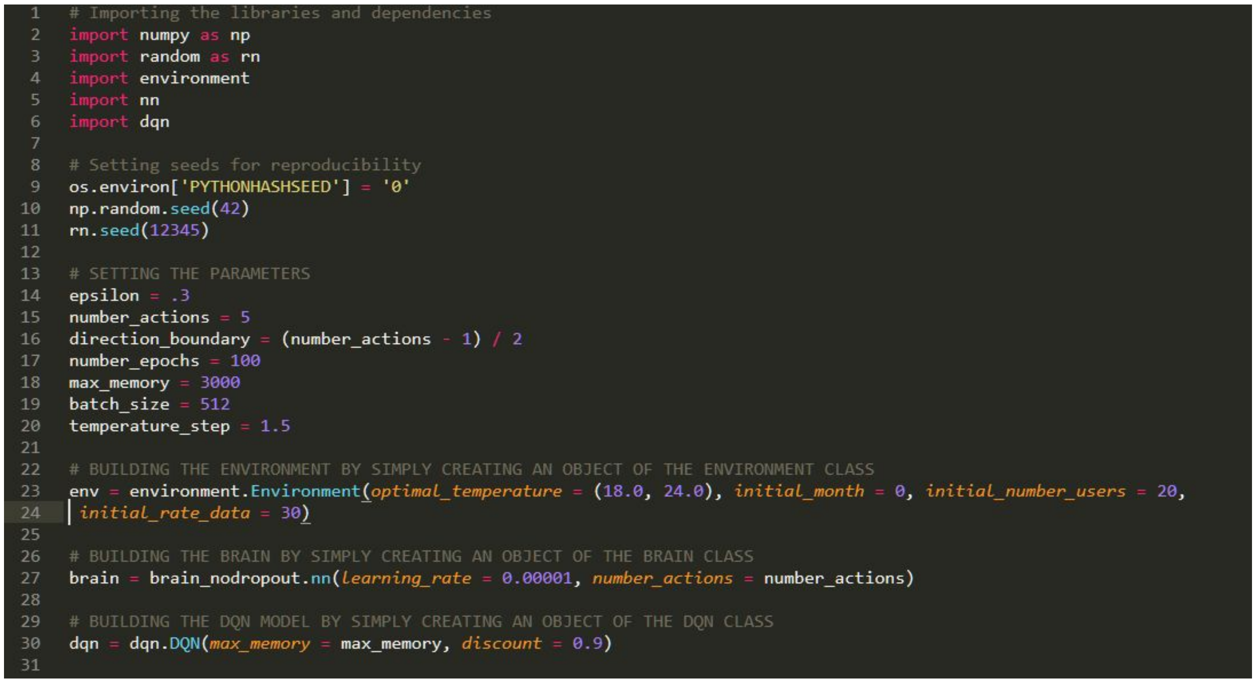
```python
# Importing the libraries
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import Adam

# BUILDING THE BRAIN

class Brain(object):

    # BUILDING A FULLY CONNECTED NEURAL NETWORK DIRECTLY INSIDE THE INIT METHOD

    def __init__(self, learning_rate = 0.001, number_actions = 5):
        self.learning_rate = learning_rate

        # BUILDIND THE INPUT LAYER COMPOSED OF THE INPUT STATE
        states = Input(shape = (3,))

        # BUILDING THE FULLY CONNECTED HIDDEN LAYERS
        x = Dense(units = 64, activation = 'sigmoid')(states)
        y = Dense(units = 32, activation = 'sigmoid')(x)

        # BUILDING THE OUTPUT LAYER, FULLY CONNECTED TO THE LAST HIDDEN LAYER
        q_values = Dense(units = number_actions, activation = 'softmax')(y)

        # ASSEMBLING THE FULL ARCHITECTURE INSIDE A MODEL OBJECT
        self.model = Model(inputs = states, outputs = q_values)

        # COMPILING THE MODEL WITH A MEAN-SQUARED ERROR LOSS AND A CHOSEN OPTIMIZER
        self.model.compile(loss = 'mse', optimizer = Adam(lr = learning_rate))

```

Implementing the DQN :



```python
# IMPLEMENTING DEEP Q-LEARNING WITH EXPERIENCE REPLAY
class DQN(object):
    # INTRODUCING AND INITIALIZING ALL THE PARAMETERS AND VARIABLES OF THE DQN
    def __init__(self, max_memory = 100, discount = 0.9):
        self.memory = list()
        self.max_memory = max_memory
        self.discount = discount

    # MAKING A METHOD THAT BUILDS THE MEMORY IN EXPERIENCE REPLAY
    def remember(self, transition, game_over):
        self.memory.append([transition, game_over])
        if len(self.memory) > self.max_memory:
            del self.memory[0]

    # MAKING A METHOD THAT BUILDS TWO BATCHES OF INPUTS AND TARGETS BY EXTRACTING TRANSITIONS FROM THE MEMORY
    def get_batch(self, model, batch_size = 10):
        len_memory = len(self.memory)
        num_inputs = self.memory[0][0][0].shape[1]
        num_outputs = model.output_shape[-1]
        inputs = np.zeros((min(len_memory, batch_size), num_inputs))
        targets = np.zeros((min(len_memory, batch_size), num_outputs))
        for i, idx in enumerate(np.random.randint(0, len_memory, size = min(len_memory, batch_size))):
            current_state, action, reward, next_state = self.memory[idx][0]
            game_over = self.memory[idx][1]
            inputs[i] = current_state
            targets[i] = model.predict(current_state)[0]
            Q_sa = np.max(model.predict(next_state)[0])
            if game_over:
                targets[i, action] = reward
            else:
                targets[i, action] = reward + self.discount * Q_sa
        return inputs, targets

```

Simulation Setup :

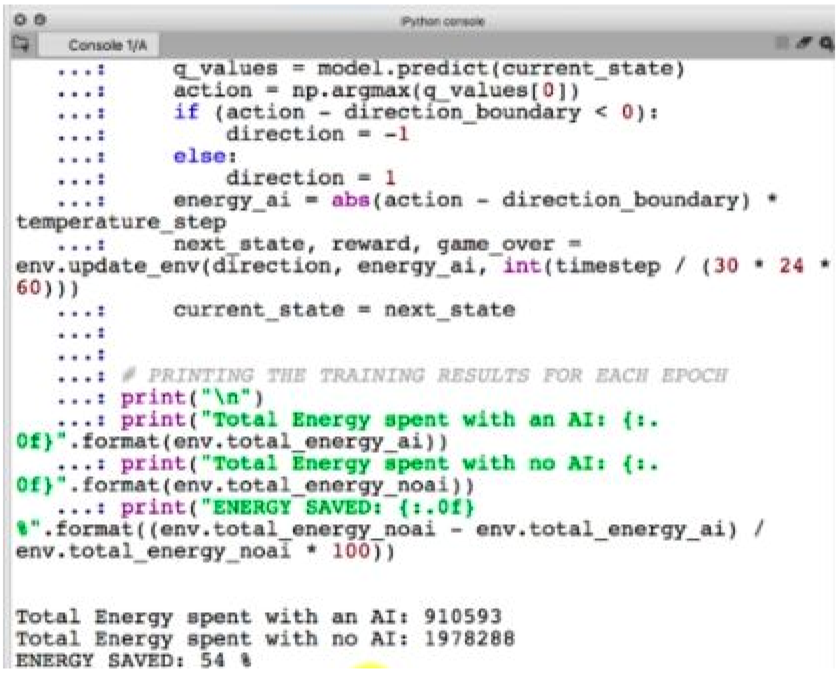


```python
# Importing the libraries and dependencies
import numpy as np
import random as rn
import environment
import nn
import dqn

# Setting seeds for reproducibility
os.environ['PYTHONHASHSEED'] = '0'
np.random.seed(42)
rn.seed(12345)

# SETTING THE PARAMETERS
epsilon = .3
number_actions = 5
direction_boundary = (number_actions - 1) / 2
number_epochs = 100
max_memory = 3000
batch_size = 512
temperature_step = 1.5

# BUILDING THE ENVIRONMENT BY SIMPLY CREATING AN OBJECT OF THE ENVIRONMENT CLASS
env = environment.Environment(optimal_temperature = (18.0, 24.0), initial_month = 0, initial_number_users = 20,
 initial_rate_data = 30)

# BUILDING THE BRAIN BY SIMPLY CREATING AN OBJECT OF THE BRAIN CLASS
brain = brain_nodropout.nn(learning_rate = 0.00001, number_actions = number_actions)

# BUILDING THE DQN MODEL BY SIMPLY CREATING AN OBJECT OF THE DQN CLASS
dqn = dqn.DQN(max_memory = max_memory, discount = 0.9)

```

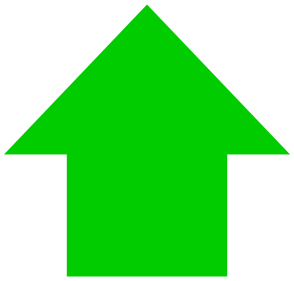
Simulation - Training and Testing Code.

3. Results and Analysis :

After training the model for 100 epochs each being 5 months, the model was tested for 1 year and the results were a savings on around 40-50% depending on different trials.



```
    ...:        q_values = model.predict(current_state)
    ...:        action = np.argmax(q_values[0])
    ...:        if (action - direction_boundary < 0):
    ...:            direction = -1
    ...:        else:
    ...:            direction = 1
    ...:        energy_ai = abs(action - direction_boundary) *
temperature_step
    ...:        next_state, reward, game_over =
env.update_env(direction, energy_ai, int(timestep / (30 * 24 *
60)))
    ...:        current_state = next_state
    ...:
    ...:
    ...:    # PRINTING THE TRAINING RESULTS FOR EACH EPOCH
    ...: print("\n")
    ...: print("Total Energy spent with an AI: {:.
0f}".format(env.total_energy_ai))
    ...: print("Total Energy spent with no AI: {:.
0f}".format(env.total_energy_noai))
    ...: print("ENERGY SAVED: {:.0f}
%".format((env.total_energy_noai - env.total_energy_ai) /
env.total_energy_noai * 100))


Total Energy spent with an AI: 910593
Total Energy spent with no AI: 1978288
ENERGY SAVED: 54 %
```



Important References :

1. *DeepMind, 2016;* ***DeepMind AI Reduces Google Data Centre Cooling Bill by 40%***
2. *Richard Sutton et al., 1998,* ***Reinforcement Learning I: Introduction***
3. *Arthur Juliani, 2016,* ***Simple Reinforcement Learning with Tensorflow (Part 4)***
4. *D. J. White, 1993,* ***A Survey of Applications of Markov Decision Processes***
5. *Richard Sutton, 1988,* ***Learning to Predict by the Methods of Temporal Differences***
6. *Michel Tokic, 2010,* ***Adaptive ε-greedy Exploration in Reinforcement Learning Based on Value Differences***
7. *Tom Schaul et al., Google DeepMind, 2016,* ***Prioritized Experience Replay***