Reinforcement Learning for Optimizing the Demand Response

Rithik Seth IIT2018032, Mrigyen Sawant IIT2018033, Harsh Kochar IIT2018049,
Adarsh Abhijat IIT2018056, Priyatam Reddy Somagattu IIT2018093

Guided By: Professor O.P. Vyas
VI Semester B.Tech, Information Technology,
Indian Institute of Information Technology, Allahabad, India

Abstract

In this paper, we will focus on learning and implementing a Reinforcement Learning model for handling and optimizing demand response with the aim of maximizing overall benefits for both demand and supply sides and minimizing energy losses and utilization by adjusting prices (tariff) which will be based on the consumer's usage pattern at different intervals of time. We have implemented a LSTM custom environment using OpenAI gym which provides our RL agent with Supply, Demand and prices at each time step. We have also implemented an A3C model interacts with our simulated environment and predicts the future prices based on the reward that it gets.

Index Terms

Reinforcement Learning, Demand Response, Smart Grid, Markov Decision Process, Monte Carlo, REINFORCE, Q-Learning, Deep Q-Learning, Actor Critic Algorithm, Asynchronous Advantageous Actor Critic Algorithm, Proximal Policy Optimization, Phasic Policy Gradient

CONTENTS

I. Introduction

With the increase in the use of smart grid-enabled technologies which enable communication between the producers and consumers we need to implement technology architecture to take advantage of this ability, and one of the best architectures for this is the Demand Response architecture, as evidenced by the myriad of literature out there endorsing its effectiveness in improving the efficiency of use of electricity and reducing the variability of the demand and supply.

Here our primary objective is to maximize profits for both demand and supply sides and reduce the variation between demand and supply of electricity by implementing demand response management through reinforcement learning by altering prices of electricity - what we will call "tariff" - based on demand response [1]. This is done in order to reduce energy losses.

A. *Understanding the environment*

Our environment mainly consists of three major components:

1) **Main Supplier** - This entity mainly provides resources/utility to another entity that demands it.
2) **Household/Consumer** - This entity consumes resources based on its need which it gets from the supplier. This entity could also act as a producer if it's capable of producing resources.
3) **Broker** (RL agent) - These brokers basically determine the prices (tariff) based on current market conditions which ensure the least fluctuation in supply and demand ratio and hence maximizing the profits on both sides without utilization heavy energy usage.

II. Literature Review

Environmental concerns for renewable resources exist for every nation and handling such issues concerning ecosystem balance is a must. It's always considered ideal if the supply is exactly equal to the demand that needs to be fulfilled but it's not always feasible in real-world scenarios especially during peak season where demand is high. If this demand becomes too high then deployment of additional power plants to cover up those demands seems to be not a good approach to handle such situations. In this paper, a notion of demand response (DR) is discussed thoroughly. The latest definition and how its classifications are done are part of this investigation. Moreover, the benefits and costs for deploying DR are also discussed in the later sections of the paper.

Demand Response is the change in demand of resources by consumers from their normal rates of consumption in response to the change in prices of those resources. These prices are computed in such a way that leads to lower consumption of such resources. Demand Side Management (DSM) programs [4] tend to maximize benefits in terms of both economic and operational levels. The primary focus of such programs is to minimize energy losses and optimize the power levels in the networks.

Demand Response architectures [13] require a lot of modern technology to implement - for example, smart meters, home energy controllers, wired and wireless communications. Such technologies enable benefits such as load reductions that occur during peak power demand, and rapidly decreasing the response times of the broker to shifts in demand. Considering the purpose of Demand Side Management [12], it's classified into three categories.

1) **Economic Driven:** The main purpose of such programs is to reduce general costs of energy supply and mitigate issues like price volatility in response to current market conditions.
2) **Environmental Driven:** These programs are more concerned about its effects on the environment and are more committed to reducing environmental damage by reducing greenhouse emission levels or by adopting more energy-efficient techniques.
3) **Network Driven:** The aim of such programs is to maintain the consistency and focus more on the reliability level of the network associated with energy supply management.

Demand Response (DR) has significant benefits [6] some of them are discussed below: Economic: These benefits are the most important ones of DR and help in the overall usage of such renewable resources. Here the savings are collected from consumers who reduce their consumption patterns at he peak hours when prices become high. Pricing and lower cost in energy generation: There are other advantages like reducing price volatility and energy costs from energy generation units from lesser demand and therefore reducing the overall costs of expensive units carrying out the energy generation process during peak hours. Environmental: The emissions from renewable resources are reduced since the demand for resources are discouraged by increasing prices and hence energy production is reduced during peak seasons [18].

DR costs [4] are classified based on both supply and demand side. Demand side have to install additional equipment such as smart thermostats, communication infrastructures, energy management systems in electrical appliances which use electricity. Supply side will have additional initial capital costs, program design architecture costs which will be followed by ongoing costs such as operating and maintenance costs for those energy generation units. Deployment costs for communication infrastructures in control side which are responsible for measuring, gathering, analyzing and transmitting energy consumption/ energy supply information.

Some research papers establish DR objectives as Peak Clipping and Load balancing. Peak clipping ensures grid stability so that this happens rarely. We want to ensure efficient grid utilization through incentivized tariff cost reduction on the consumer side, according to a paper thus can be done by load balancing.

Load reduction algorithms operate on the DR architectures , with the primary goal being reducing the consumer load in accordance with current supply and tariffs. The papers also talk about using Non-Intrusive Load Monitoring (NILM) [9] devices for collecting statistical data of users' appliances to reduce load.

Load Shifting Algorithms [7] are used when consumption cannot be reduced by reduction algorithms, cost savings and grid stability can be achieved by shifting the load to a cheaper billing period. Load shifting assumes the possibility of remotely scheduling an appliance, as opposed to locally scheduling an appliance.

A. Markov Decision Process

Markov Decision Process [14] (MDP) is a framework used to frame problems of learning from interaction to achieve a goal .

MDP is a formal approach for describing an environment that is fully observable for reinforcement learning. MDP can be used to for formalizing RL problems.

All states in the Markov Decision Process follow "Markov" property, where the future state is dependent on the present state, not the past states:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1,\ldots,S_t]$$

In other words, the present state is statistically sufficient to decide the future state and the past and future states are conditionally independent.

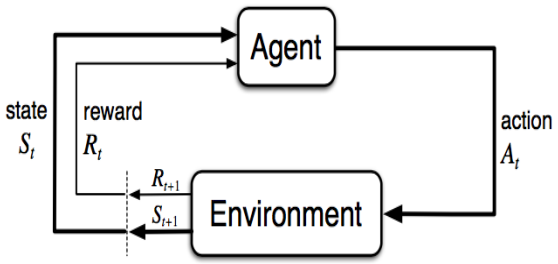

Figure 1. Interaction between agent and environment in Markov Decision Process [14]

A Markov decision process consists of five elements $M = \langle S,A,P,R,\gamma\rangle$, where the symbols mean:

- S - set of states A - set of actions
- P - transition probability function
- R - reward function
- γ - Discounting factor for rewards

B. Monte Carlo

The Monte Carlo [14] method for reinforcement learning learns without the need for prior knowledge of Markov Decision Process transitions but from episodes of raw experience without modeling the environment dynamics and it computes mean observed as an approximation of expected return. The random component used is the reward or return.

For computation of the empirical return G_t, Monte Carlo methods need to learn from entire episodes $S_1,a_1,R_2,\ldots,S_t$ to compute

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

For state s, the empirical mean return is:

$$V(s) = \frac{\sum_{t=1}^{T} 1[S_t=s]G_t}{\sum_{t=1}^{T} 1[S_t=s]}$$

Here $1[S_t = s]$ is a binary indicator function. We may visit a state multiple times but for this to happen we need to keep track of the number of visits occurred for a state s or only consider the first encounter of the state s without need of encountering a state multiple times. This approximation can be easily extended to action-value functions by keeping track of (s, a) pairs.

$$Q(s,a) = \frac{\sum_{t=1}^{T} 1[S_t=s,a_t=a]G_t}{\sum_{t=1}^{T} 1[S_t=s,a_t=a]}$$

For Monte Carlo method to learn optimal policy we need to iterate it multiple times.

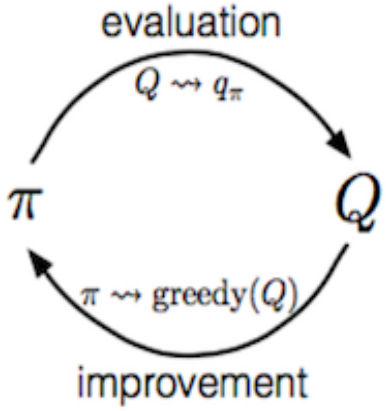

Figure 2. Iteration in Monte Carlo [14]

How we are going to Iterate and obtain optimal policy:

1) We improve the policy using the current value function in a greedy manner, $\pi(s) = \arg\max_{a\in A} Q(s,a)$
2) By the use of new policy π generate a new episode.
3) Estimate Q by using the new episode:

$$q_\pi(s,a) = \frac{\sum_{t=1}^{T}\left(1[S_t=s,a_t=a]\sum_{k=0}^{T-t-1}\gamma^k R_{t+k+1}\right)}{\sum_{t=1}^{T} 1[S_t=s,a_t=a]}$$

C. Q-Learning

Q-Learning [15] is a model-free reinforcement learning algorithm to learn quality of actions telling an agent what action to take under what circumstances. For any finite Markov decision process, it finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. (See appendix Algorithm 1)

1) Working: The main idea of Q-learning is that your algorithm predicts the value of a state-action pair, and then you compare this prediction to the observed accumulated rewards at some later time and update the parameters of your algorithm, so that next time it will make better predictions.[16] Most important part here is where the Q-value is updated, this is given by :

$$Q(S_t,a_t) \leftarrow Q(S_t,a_t) + \alpha(R_{t+1} + \gamma \max_{a\in\mathscr{A}} Q(S_{t+1},A) - Q(S_t,a_t))$$

- S_t - state at time t
- A - set of actions
- a_t - actions taken at time t
- α - Learning rate

- R - reward function
- γ - discounting factor for future rewards. In an unknown environment, we do not have perfect knowledge about P and R.

D. Deep Q-Learning

In Deep Q-Learning [11], we theoretically memorize $Q_*(.)$for all state-action pairs in Q-learning, like in a gigantic table. However, it quickly becomes computationally infeasible when the state and action space are large. Thus people use functions (i.e. a machine learning model) to approximate Q values and this is called function approximation. For example, if we use a function with parameter θ to calculate Q values, we can label Q value function as $Q(S,a;\theta)$. (See appendix Algorithm 2)

1) Working: Unfortunately Q-learning [8] may suffer from instability and divergence when combined with an nonlinear Q-value function approximation and bootstrapping.

It's Cost function:

$$\mathscr{L}(\theta) = E_{(S,a,r,S')\sim U(D)}\left[\left(r+\gamma\max_{a'} Q(S',a';\theta^-) - Q(S,a;\theta)\right)^2\right]$$

- $U(D)$ is a uniform distribution over the replay memory D
- θ is the parameters of the frozen target Q-network.

Deep Q-Network aims to greatly improve and stabilize the training procedure of Q-learning by two innovative mechanisms:

2) Experience Replay: All the episode steps $e_t = (S_t, a_t, R_t, S_{t+1})$ are stored in one replay memory $D_t = \{e_1, \ldots, e_t\}$. D_t has experience tuples over many episodes. During Q-learning updates, samples are drawn at random from the replay memory and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.

3) Periodically Updated Target: Q is optimized towards target values that are only periodically updated. The Q network is cloned and kept frozen as the optimization target every C steps (C is a hyper-parameter). This modification makes the training more stable as it overcomes the short-term oscillations.

III. Dataset

Since our aim is to create a Reinforcement Learning agent to maximize profits and reduce the variation between demand and supply, the first thing we need to look at is how we would train and test potential Reinforcement Learning candidate algorithms [5]. This involves creating an environment that emulates the environment that the Reinforcement Learning agent would find itself in, and find out any optimizations we can make to the Reinforcement Learning agent itself to improve its performance.

Initially we decided to use an LSTM model[2] [3] trained on the Ontario electric demand dataset. This LSTM model now takes an input a sequence of states that contain previous demand and price, and provides possible next state. This model will form part of the environment for the Reinforcement Learning agents to interact with.

So currently we are using the trimmed down version of the Ontario dataset as there were many unnecessary attributes as of now and that can later be added to extend the dataset.

IV. Methodology

We have proposed a methodology where we use Asynchronous Advantage Actor Critic model for implementing the agent and an environment that uses LSTM to simulate real world scenarios.

In our A3C model, we have a master agent which is responsible for the decision making based on the current state of the environment and we have worker agents whose sole responsibility is to explore and update both policy and value networks asynchronously which are common to all worker agents.

A. Working of Worker networks

Worker agents are created by the master agent that are responsible for exploration and updation of the policy and value networks. The work of worker agents can be divided into calculation of trajectory and updation of the networks.

1) Calculation of trajectory: A trajectory is the path that the agent takes through a state, action and reward space. The length of the trajectory can vary and be set. Consider T to be trajectory length that is set. It is assumed that every worker agent has a copy of the current state of environment upon which they explore.

- First the agent observes the current state of the environment at a given time t, S_t and this is given to the policy network to generate a probability distribution $\pi_{\theta A}(a_t|S_t)$.
- We create a categorical probability distribution that is respect to probability distribution function generated by policy network that helps in sampling random action.
- Upon taking an action a_t, the agent observes the next state S_{t+1} and reward r_t.
- The agent stores (S_t, a_t, r_t) tuple and repeats the above steps till trajectory length T.

This process generates a trajectory for each worker agent and the trajectory is different for every agent because of the random actions chosen to explore the environment.

2) Updating policy and value networks: Before updating the policy and value networks, the worker agents calculate advantage value of each tuple, target value of each states, and loss value of policy and value networks for the considering the entire trajectory.

The advantage value is calculated for each tuple present in the trajectory by using the n-step method for every tuple in the trajectory,

$$A(S_t,a_t) = \left(\sum_{i=0}^{T-1} \gamma^i * r_{t+i}\right) + \gamma^{T-1} * v_{predicted}(S_t), \forall t \in \{0,\ldots,T\}$$

where, S_t is state, a_t is action, r_t is reward at time t, γ is discount factor and $v_{predicted}(S_t)$ is value of state

calculated by the value network

The agent calculates v_{target} is value of a state S_t which is a better measure than $v_{predicted}$,

$$v_{target}(S_t) = (\sum_{i=0}^{T-1} \gamma^i * r_{t+i}) + \gamma^{T-1} * v_{predicted}(S_t), \forall t \in \{0, ..., T\}$$

Finally before updating the policy and value networks which are represented by θ_A and θ_C, we calculate loss of both policy and value networks.

Loss of policy network is calculated by the given equation,

$$L_{policy}(\theta_A) = \frac{\sum_{t=0}^{T}(v_{predicted}(S_t) - v_{target}(S_t))^2}{T}$$

Loss of value network is calculated by the given equation,

$$L_{value}(\theta_C) = \frac{\sum_{t=0}^{T}(-A(S_t, a_t)\log(\pi_{\theta_A}(a_t|S_t)))}{T}$$

where, $A(S_t, a_t$ is the advantange function and $\pi_{\theta A}(a_t|S_t)$ is the probability distribution of an action given a state at time t given by policy network.

The policy and value networks are updated in the following way, considering α_{θ_A} is the learning rate of policy network(actor), α_{θ_C} is the learning rate of value network(critic).

$$\begin{array}{rcl} \text{Policy network} & & \\ \theta_A & = & \theta_A + \alpha_{\theta_A} \nabla_{\theta_A} L_v alue(\theta_A) \\ \text{Value network} & \text{or} & \text{Critic network} \\ \theta_C & = & \theta_C + \alpha_{\theta_C} \nabla_{\theta_C} L_v alue(\theta_c) \end{array}$$

After the exploration is done by the worker agents and the networks are updated, the master agent based on the current environment state takes the best suitable action to maximize the overall reward.

V. Environment

The environment uses the OpenAI Gym library as a standard API for the agent's use. This enforces a limitation: all the inputs the environment gets from the agent, and sends to the agent, are either simple integers or floating points, or numpy arrays.

The API consists of these main functions:

- $reset()$: Send the initial state to the agent.
- $step(action)$: The most important function; it involves taking the *action* variable from the agent and compute the next state and the corresponding reward, which is then sent to the agent.

A. Actions

In reinforcement learning, there are two kinds of action spaces we can use: discrete and continuous. The bigger the action space is, the more complex our environment is for the agent to comprehend, which means that training the agent to give the right outputs and to process its inputs correctly takes even longer.

For our purposes, the simplest solution would consist of three discrete actions: increase the price by a certain amount, do nothing, and decrease the price by a certain amount. The downside of this solution is that at each step, you are limited to only increasing or decreasing the price by a fixed amount, making the agent's actions inflexible when faced with extreme changes in the environment. We could, of course, let the agent decide how much they want to increase or decrease the price, based on its uncertainty and understanding of the environment state. However, this requires a continuous action space, since we need to decide both the action - whether to increase or decrease the price or make no changes - and the scale of the action. We could, of course, just allow the agent to directly set the price to some arbitrary amount, but this would be the hardest environment for an agent to train in, especially since this allows for infinite permutations of trajectories of the environment. Therefore, all actions are formulated in terms of change to the previous state's price, instead of entirely new price points.

We have chosen to not use the more complex formulation to focus on correctness of the system and to ensure that training the system was not infeasible on a personal desktop computer. This means we use a fixed set of discrete actions, each of which increase or decrease the price by a fixed percentage, and an optional action that results in no change in price.

We have experimented with removing the "no change" action, and that has resulted in a faster training of the agent.

B. Step Workflow

Here is a detailed workflow of the step function:

- Verify action is legal.
- Get the next state by sending the history of environment states (including the current environment state) to the LSTM.
- Calculate the new price by taking into account the effect of the action.
- Set the price of the next state to the new price that has just been calculated.
- Calculate the non-normalized reward based on the new price and the demand and supply values of the *next state*
- Add the next state to the historical record of environment states.
- Return the next state and the non-normalized reward to the agent.

Focus on the fourth point: "Set the price of the next state to the new price that has just been calculated." What we are doing is updating the price of the *next state*, not the current state. Similarly, when we are calculating the reward, we are using the new price - the price of the *next state* - and the demand and supply values of the next state. The reward is calculated based on the output of the next state, not based on the current state.

This is because we have decided to make any state returned to the agent as immutable. If we assume the current state as mutable, then whatever action the agent sends changes the price of the *current state*, making the price sent as part of the current state in the previous step as irrelevant information likely to confuse the agent's calculations.

Next, our reward calculation function only takes one specific state into account when calculating the reward.

It would make no sense, for example, to use the price of the previous state and the demand and supply variables of the current state. Since we intend to use the action's effects as part of our reward calculation, we have to take into account the state that has been affected by the action, which is the next state - the state that will be returned to the agent once this function completes computation.

C. Reward function

The reward function is created keeping a few goals in mind:

- Ensure that the demand is greater than supply, so that the producer makes a profit, instead of having to pay consumers to consume electricity.
- Ensure that there is a buffer of demand, so that sudden changes in supply or demand do not reduce profitability of the producer by a huge amount.
- Ensure that the price of the electricity is not extremely high, which will cause long term reduction in demand.

To that end, we use the following formula to calculate the reward for some optimal value of x and y:

$$reward = |(demand - supply)^x| * |newPrice^y| * correction$$

Where correction is dependant on 3 criteria:

- Whether *newPrice* is within bounds.
- Whether $demandorsupply$.
- Whether reward is normalized.

and *newPrice* is the non-normalized value of the price that the action has .

D. Dataset normalization

We normalize the dataset to make training the neural networks easier. We use the following formula to normalize the dataset values:

$$value_{row,col} = \frac{value_{row,col} - min(valueS_{col})}{max(valueS_{col}) - min(valueS_{col})}$$

By default, we use this normalized data for every operation of the environment and the reinforcement learning agent. This also means that the data generated by the LSTM is normalized, and the data that the policy network and the critic network expects is normalized. The only exception to this is the reward function. The input to the reward function is not normalized, and therefore the reward output is therefore based on the non normalized values. This reward is passed to the agent, and therefore even the agent, in an indirect sense, uses the non-normalized values.

VI. Agent

A. REINFORCE model

We have used the REINFORCE algorithm as the RL agent that interfaces with our custom environment.

REINFORCE is a policy gradient reinforcement learning algorithm. It is a Monte-Carlo variant of policy gradient, which means that it takes random samples of the environment and then does gradient descent based on the reward that it gets. Simply put, it uses a policy network that it updates using gradient descent.

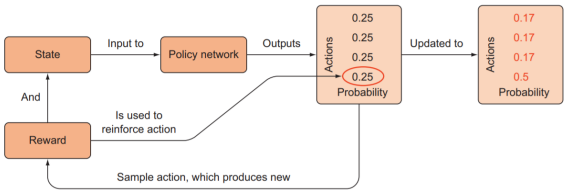

Figure 3. Overview of REINFORCE algorithm [5]

The working of REINFORCE algorithm [17] relies on the fact that the expectation of the sample gradient is equal to the actual gradient.(See appendix Algorithm 7)

$$\nabla_\theta J(\theta) = E_\pi[G_t \nabla_\theta \log(\pi_\theta(a_t|S_t))]$$

The main idea of reinforce algorithm relies on the fact that Policy based reinforcement learning is an optimization problem and can be solved using gradient descent. For this we consider the policy objective function J_θ which represents the most expected accumulative reward. The goal is to find parameters θ such that J_θ is maximized through which we can get the optimal policy. Through the use of gradient descent we have the REINFORCE algorithm [17].

$$\textbf{update rule : } \theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

1) Testing and results: We programmed a vanilla policy gradient algorithm conventionally called a REINFORCE algorithm using PyTorch. This reinforcement learning agent interfaced with the OpenAI custom Gym environment, which provided us with valuable feedback about the Gym and helped us to refine the design and fine-tune the parameters of the environment.

This series of rigorous testing also exposed the limitations of REINFORCE: the greater the action space, the more computational resources it would take for the agent to learn a sensible policy. Using a continuous action space, as we found out, was entirely out of the question. In the end, we had to balance the amount of actions by minimizing it, while also maximizing the impact the agent's actions had on the environment. (See appendix Algorithm 8)

B. Actor Critic Algorithm

There are two types of reinforcement learning methods one being policy based and other being value based. Instead of learning both methods separately, Actor-Critic [14] method tries to learn both policy and value networks simultaneously, using critic (value based) to refine the Actor (policy based). It is much more beneficial to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in vanilla policy gradients, (See appendix Algorithm 3)

Actor-critic methods [10] consist of two models, which may optionally share parameters:

- **Critic** updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|S)$ or state value $V_w(S)$.
- **Actor** updates the policy parameters θ for $\pi_\theta(a|S)$

C. Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic, short for A3C, is a classic policy gradient method with a main focus on parallel training. The critics learn the value function in A3C while multiple actors are trained in parallel and are synchronized from time to time with the use of global parameters. Hence, for parallel training, A3C is built to function well on modern CPU which have multi threading capabilities. (See appendix Algorithm 5)

In multiple agent preparation, A3C allows for parallelism. The step of gradient accumulation (6.2) can be considered as a parallelized reformation of the stochastic gradient update based on mini-batch: the values of w or θ are corrected independently by a little bit in the direction of each training thread. [10]

1) Policy Network: Policy is the mapping of states to action with a certain probability, i.e it maps states to action probabilities. A good policy maximizes the discounted total rewards and is referred to as the Optimal/ideal policy. In the Actor-Critic class of algorithms learning a good policy is crucial, and since the policy is a probabilistic function , it can be approximated really well by using neural networks which are natural function approximators. Whenever a deep neural network is used to approximate the policy of the agent , it can be parameterized by some parameters θ , the corresponding Policy network created is called π_θ. The network is created so that it's input is the action with several hidden layers in between and output is a ndarray giving probability for each action in action space.

2) Critic Network: Similar to the policy network, the Value/Q network can also be parameterized and can be approximated through the use of deep neural networks. Here also the input will be state and the output will be V value in case of value network and , Q value for each action for Q network. If the state space is very small we can also use tabular method of storing Q/V values.

3) Advantage Function: For S_i is the current state and a_j is the action being taken, thus advantage is the difference of Q value of a state for a action and the value of the state. This tells us how good a particular action is for the state. The Advantage Function:

$$A(S_i|a_j) = Q(S_i|a_j) - V(S_i)$$

VII. Results

A. LSTM performance

The LSTM used to simulate the environment is trained on the aforementioned smart grid historical dataset before use as part of the environment for the reinforcement learning agent. Figure 6 shows the output of the loss functions reducing as the LSTM is trained over a subset of the dataset.

After changing the loss function to MSE (mean squared error), as can be seen in Figure 7, there was a clear increase in efficiency of training, which is why we decided to use MSE as the loss function for the LSTM.

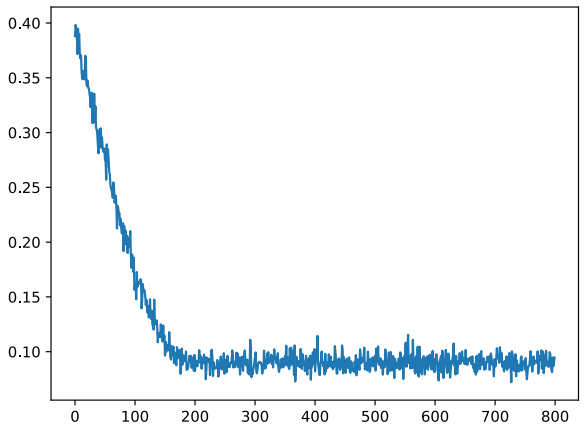

Figure 4. LSTM Training Loss per 100 episodes, using L1

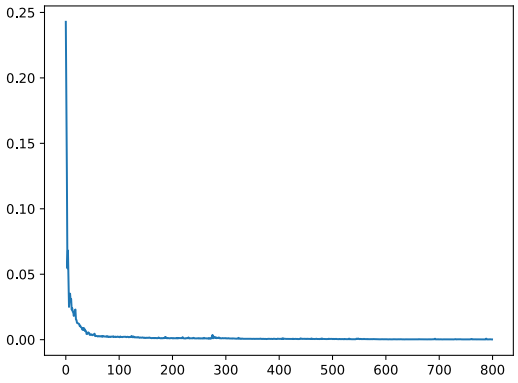

Figure 5. LSTM Training Loss per 100 episodes, using MSE

B. A3C performance

One of our main goals is to converge the policy network to the ideal policy. We do this using the policy gradient algorithm that maximizes the objective function. As seen in Figure 8, over the course of 5000 episodes, the agent clearly has its policy function converging to an ideal policy.

The performance of the critic network gets better as the A3C model is trained more, as seen in Figure 9: the output of the loss function decreases steadily and after 4000 episodes hovers close to $0.01 * 10^6$.

The advantage is a measure of how good a particular action is compared to all the other actions possible in a specific state. The average advantage is the average of all advantages for all actions taken in a given episode.
The below graph shows how, as the agent is trained, the average advantage increases, which implies that, on average, the actions taken by the agent is getting better and better in comparison to other possible actions it could take. The improvement is constant, even past 5000 episodes of training.

The graph below shows the results of training the agent. This has multiple implications, all positive:

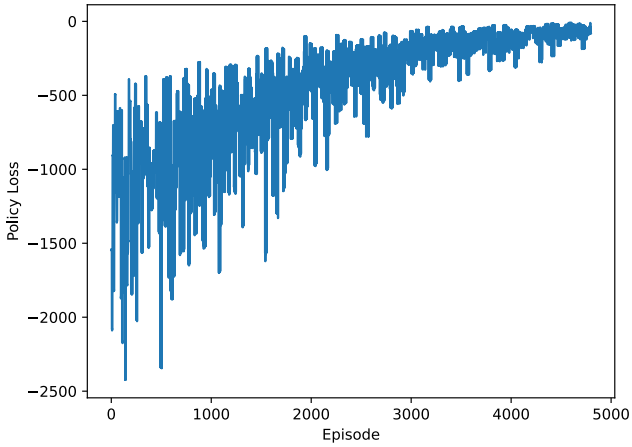

Figure 6. Maximization of objective function

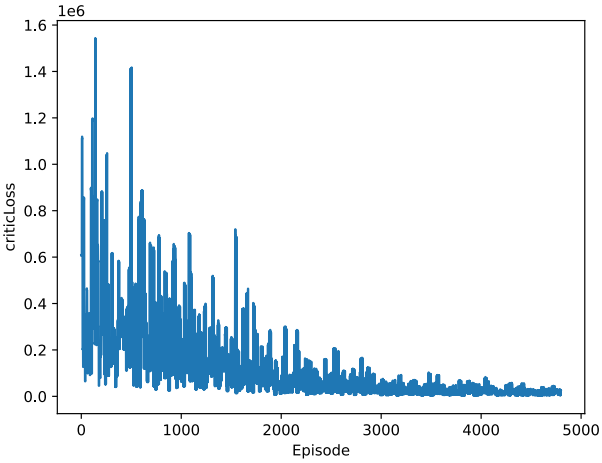

Figure 7. Critic loss

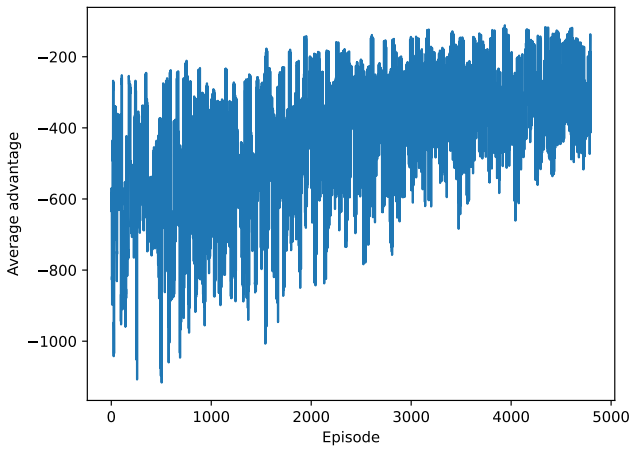

Figure 8. Maximization of average advantage

- The agent consistently keeps the demand greater than the supply. This ensures that the producer is profitable.
- The agent consistently keeps a buffer between the demand and supply values. This ensures that in the occasion of sudden increases in supply, there is still demand for electricity to absorb the increase in supply, ensuring that the producer is still profitable.
- The agent endeavours to decrease the price in such a manner as to keep profits high. This benefits both the producer and and consumers.

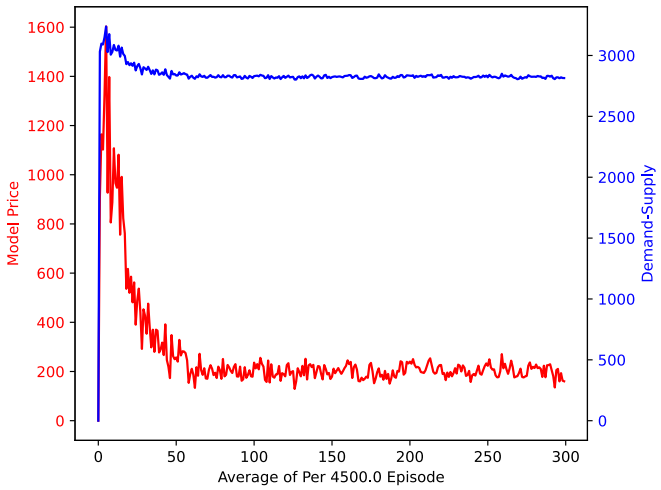

Figure 9. Demand-supply vs Model price curve predicted by our RL model keeping their product to be maximized for maximum profit based on our assumptions

VIII. References

[1] Sashank Mishra Agam Dwivedi Ruchin Agrawal. *Autonomous Decision Making in Smart Grids*.

[2] Sashank Mishra Agam Dwivedi Ruchin Agrawal. *House Hold Load Prediction, LSTM*.

[3] Sashank Mishra Agam Dwivedi Ruchin Agrawal. *House Hold Load Prediction, VAR-CNN-LSTM*.

[4] J. Aghaei and Mohammad Iman Alizadeh. "Demand response in smart electricity grids equipped with renewable energy sources: A review". In: *Renewable and Sustainable Energy Reviews* 18 (Feb. 2013), pp. 64–72. DOI: 10.1016/j.rser.2012.09.019.

[5] Brandon Brown. *Deep Reinforcement Learning in Action*. City: Manning Publications, 2020. ISBN: 978-1-61729-543-0.

[6] Benjamin Dupont et al. "Impact of residential demand response on power system operation: A Belgian case study". In: *Applied Energy* 122 (June 2014), pp. 1–10. DOI: 10.1016/j.apenergy.2014.02.022.

[7] Ivana Dusparic et al. "Residential demand response: Experimental evaluation and comparison of self-organizing techniques". In: *Renewable and Sustainable Energy Reviews* 80 (Dec. 2017), pp. 1528–1536. DOI: 10.1016/j.rser.2017.07.033.

[8] Sayon Dutta. *Reinforcement Learning with TensorFlow*.

[9] Yee Wei Law et al. "Demand Response Architectures and Load Management Algorithms for Energy-Efficient Power Grids: A Survey". In: Nov. 2012, pp. 134–141. ISBN: 978-1-4673-4564-4. DOI: 10.1109/KICSS.2012.45.

[10] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: 1602.01783 [cs.LG].

[11] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[12] Prashant Reddy and Manuela Veloso. "Strategy Learning for Autonomous Agents in Smart Grid Markets." In: July 2011, pp. 1446–1451. DOI: 10.5591/978-1-57735-516-8/IJCAI11-244.

[13] Pierluigi Siano. "Demand response and smart grids—A survey". In: *Renewable and Sustainable Energy Reviews* 30.C (2014), pp. 461–478. DOI: 10.1016/j.rser.2013.10.02. URL: https://ideas.repec.org/a/eee/rensus/v30y2014icp461-478.html.

[14] Richard Sutton. *Reinforcement Learning*. Boston, MA: Springer US, 1992. ISBN: 978-1-4615-3618-5.

[15] Christopher J.C.H. Watkins and Peter Dayan. In: *Machine Learning* 8.3/4 (1992), pp. 279–292. DOI: 10.1023/a:1022676722315. URL: https://doi.org/10.1023/a:1022676722315.

[16] Lilian Weng. *A Long Peek Into Reinforcement Learning*. URL: https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html.

[17] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[18] Bo Zeng et al. "Impact of behavior-driven demand response on supply adequacy in smart distribution systems". In: *Applied Energy* 202 (Sept. 2017), pp. 125–137. DOI: 10.1016/j.apenergy.2017.05.098.

IX. Appendix

Algorithm 1: Q-Learning

Initialize $t = 0$

Start with S_0

At time step t, we pick the action according to Q values, $a_t = \arg\max_{a \in \mathscr{A}} Q(S_t, a)$ and ε-greedy is commonly applied.

After applying action a_t, we observe reward R_{t+1} and get into the next state S_{t+1}.

Updated Q-value: $Q(S_t, a_t) \leftarrow Q(S_t, a_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathscr{A}} Q(S_{t+1}, a) - Q(S_t, a_t))$

$t = t + 1$ and repeat from step 3.

Algorithm 2: Deep Q-Learning

Initialize replay memory D to capacity N.
Initialize action-value function Q with random weights
for $episode = 1, M$ **do**
  Initialise sequence $S_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(S_1)$
  for $t = 1, T$ **do**
    With probability ε select a random action at otherwise select $a_t = max_a Q\ (\phi(S_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    $S_{t+1} = S_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = st + 1$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from D

$$\text{Set } y_j = \left\{ \begin{array}{ll} r_j, & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{for non terminal } \phi_{j+1} \end{array} \right\}$$

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end
end

Algorithm 3: Actor Critic

Initialize s, θ, w at random;
sample $a \sim P(s^{'}|s,a)$;
for $t = 1$ *to* T **do**
  Sample reward $r_t \sim R(s,a)$ and next state $s^{'} \simeq P(s^{'}|s,a)$
  Then sample the next action $a^{'} \sim \pi_\theta(a^{'}|s^{'})$.
  Update the policy parameters $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta ln \pi_\theta(a|s)$
  Compute the correction (TD error) for action value at time t as : $\delta_t = r_t +_w (s^{'}, a^{'}) - Q_w(s,a)$ and use it to update the parameters of action-value function as : $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$.
  Update $a \leftarrow a^{'} and s \leftarrow s^{'}$
end
Two learning rates, namely α_θ and α_w , are predefined for policy and value function parameter updates respectively.

Algorithm 4: Advantage Actor-Critic (A2C)

Initialize step counter $t \leftarrow 1$
Initialize episode counter $E \leftarrow 1$
while $E > E_{max}$ **do**
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta \leftarrow 0$
  $t_{start} \leftarrow t$
  Get state S_t
  while *terminal* S_t ***or*** $t - t_{start} == t_{max}$ **do**
    perform a_t according to policy $\pi(a_t|S_t;\theta)$
    receive reward r_t and new state S_{t+1}
    $t \leftarrow t + 1$
  end

$$\text{r} = \left\{ \begin{array}{ll} 0, & \text{terminal } \text{S}_t \\ \text{V}(\text{S}_t, \theta_v), & \text{for non-terminal } \text{S}_t \end{array} \right\}$$

  for $i \in \{t-1, ..., t_{start}\}$ **do**
    $r \leftarrow r_i + \gamma r$
    Accumulate gradients wrt $\theta : d\theta \leftarrow d\theta \nabla_\theta \log_\pi(a_i|S_i;\theta)(r - V(S_i;\theta_v) + \beta_e \partial H(\pi(a_i|S_i;\theta))/\partial\theta$
    Accumulate gradients wrt $\theta : d\theta \leftarrow d\theta + \beta_v(r - V(S_i;\theta_v))(V(S_i;\theta_v)/\partial\theta_v)$
  end
  Perform update of θ using $d\theta$ and of θ_v using $d\theta_v$ $E \leftarrow E + 1$
end

Algorithm 5: Asynchronous Advantage Actor-Critic (A3C)

Global parameters:- θ, w
Initialise thread-specific parameters:- θ' and w'
Initialize time step $t = 1$
while $T \leq T_{max}$ **do**
  Reset gradient: $d\theta = 0$ and $dw = 0$.
  Synchronize thread-specific parameters with global ones: $\theta_{'=\theta}$ and $w^{'} = w$.
  $tstart = t$ and sample a starting state st.
  while *(st != TERMINAL) and* $t - t_{start} \leq t_{max}$ **do**
    Pick the action $a_t \sim \pi_{\theta'}(a_t|S_t)$ and receive a new reward R_t and a new state S_{t+1}.
    Update $t = t+1$ and $T = T+1$
  end
  Initialize the variable that holds the return estimation

$$R = \left\{ \begin{array}{ll} 0, & S_t = TERMINAL \\ V_{w'}(S_t), & otherwise \end{array} \right\}$$

  for $i = t-1, \ldots, t_{start}$ **do**
    $R \leftarrow \gamma R + R_i$; here R is a MC measure of G_i.
    Accumulate gradients w.r.t. : $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i|S_i)(R - V_{w'}(S_i))$;
    Accumulate gradients w.r.t. w': $dw \leftarrow dw + 2(R - V_{w'}(S_i))\nabla_{w'}(R - V_{w'}(S_i))$.
  end
  Update asynchronously θ using $d\theta$, and w using dw.
end

Algorithm 6: Phasic Policy Gradient (PPG)

for *phase = 1, 2,...* **do**
  Initialize empty buffer B
  for *iteration = 1, 2, ..., N_π* **do**
    Perform rollouts under current policy π
    Compute value function target $\hat{V}^{targ}$ for each state S_t
    for *epoch = 1, 2, ..., E_π* **do**
      Optimize L_{clip} + $\beta_S S[\pi]$ wrt θ_π
    end
    for *epoch = 1, 2, ..., E_V* **do**
      Optimize L_{value} wrt θ_V
    end
    Add all (S_t, $\hat{V}_t^{targ}$) to B
  end
  Compute and store current policy $\pi_{\theta_{old}}(.|S_t)$ for all states S_t in B
  for *epoch = 1, 2, ..., E_{aux}* **do**
    Optimize L^{joint} wrt θ_π, on all data in B
    Optimize L^{value} wrt θ_V , on all data in B
  end
end

Algorithm 7: REINFORCE

Initialize the policy parameter θ at random
Generate one trajectory on policy $\pi_\theta : S_1, a_1, R_2, S_2, a_2 ..., S_t$;
for $t = 1$ *to* T **do**
  Estimate the return G
  Update policy parameters: $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \log(\pi_\theta(a_t)|S_t)$
end

Algorithm 8: Custom Environment

Set Hyper-parameters : $\alpha, \beta, \gamma, \mu, N$
Initialize: Demand D_1,Supply S_1, Price P_1
for $\forall$ episode E **do**
  for Time step *t=1,2....* **do**
    $S_{t+1} = S_t + \gamma * (D_t - S_t)$
    $D_{t+1} = movingAverage(D,N) - \alpha * (observedPrice_{t+1} - P_t)$
    $Reward = \frac{\mu * P_t * min(D_t, S_t)}{\beta * ((D_t - S_t)^2) + 1}$
  end
end

Algorithm 9: Update Price

Set hyper-parameters: $\zeta, totalNumActions, priceUpperBound, priceLowerBound$
for $\forall$ timestep t **do**
  $action = a \in [0, totalNumActions - 1]$
  $maxChange = (priceUpperBound - priceLowerBound)/2$
  $correctingFactor = 2 * (maxChange^{1/\zeta})/totalNumActions$
  $correctedAction = action - (totalNumActions/2)$
  $price_t = price_{t-1} + (correctingFactor * correctedAction)^{\zeta}$
end