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 describe an implementation of a system that utilizes the A3C reinforcement learning model for handling and optimizing the demand response in a smart grid, with the aim of maximizing overall benefits for both the demand and the supply sides. All of this is done by adjusting the prices, which is done by taking into account the consumer's recent usage patterns, the supply trend, and other relevant factors. We have also implemented an environment for simulating a smart grid to evaluate and fine-tune reinforcement learning models for this task.**

***Index Terms*—Reinforcement Learning, Demand Response, Smart Grid, REINFORCE, Asynchronous Advantageous Actor Critic Algorithm, LSTMs, OpenAI Gym, PyTorch**

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 that takes 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[3][12][4].

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 [2]. This is done in order to reduce energy losses.

A. Understanding the environment

Our environment 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.

B. Building Blocks

Here we describe the fundamental technical building blocks of the models we create and implement in this paper.

1) REINFORCE: REINFORCE is a policy gradient reinforcement learning algorithm[18]. 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.

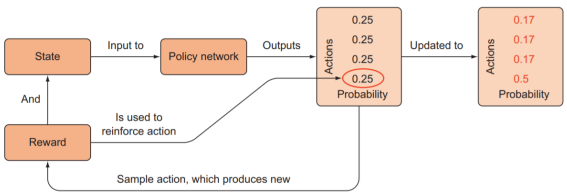

Figure 1. Overview of REINFORCE algorithm [7]

The working of REINFORCE algorithm [19] relies on the fact that the expectation of the sample gradient is equal to the actual gradient. (See appendix for algorithm)

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

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 [19].

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

2) LSTMs: LSTMs are a variant of recurrent neural networks that can keep track of arbitrary long-term dependencies in input sequences[11]. This makes them the perfect neural network to model the time series data generated by the smart grid environment. We leverage this ability by making the LSTM a part of a simulated environment.

3) OpenAI Gym: OpenAI is a not-for-profit artificial intelligence research and deployment company which conducts research in the field of AI with the goal of creating a General AI which won't be a threat to humanity.
In the context of reinforcement learning, OpenAI Gym is a set of tool-kits and libraries for developing, testing and comparing reinforcement learning algorithms[6]. The goal of Gym is to standardize the way environments are created in reinforcement learning and to provide a easy-to use general framework for bench marking and evaluating RL algorithms on different environments.

4) A3C: Asynchronous Advantage Actor-Critic[13], short for A3C, is a classic policy gradient method with a main focus on parallel training. Now, there are two core innovations of A3C that we focus on and use: the first is n-step bootstrapping, and the second is the use of concurrent actors to generate a varied set of experience samples in parallel.

In A3C, we create multiple workers, and each of these workers traditionally creates an instance of the environment and the policy and critic network which it uses for generating experience. Once a batch of experience is collected, each worker shall update the global copy of the policy and the critic network asynchronously. Once done, they reload the copy of the networks, and repeat.

The most important aspect of A3C is that the updates to its policy and critic networks are asynchronous and do not require the use of locks. This is accomplished using a style of network updates known as Hogwild[17]. Hogwild outperforms alternative schemes that use locking by an order of magnitude, and to achieve a near-optimal rate of convergence, which is why we have decided to use it as part of our implementation[14].

5) n-step bootstrapping: Bootstrapping is a method of estimating returns. It involves using the actual return for one state, which is the immediate reward , and then an estimate of the return from the next state onwards.

The actual returns are high variance. This is because they accumulate many random events in the trajectory. The effect of this randomness is compounded over the course of a trajectory. Bootstrapping is useful because it reduces the variance during training, when used instead of the actual return.

n-step bootstrapping method is simply estimating returns using n actual steps of rewards.

II. Related Work

There has been significant work int the domain of demand response in smart grids. Some methodologies have focused on directly changing the demand at the consumer's end and others have focused on changing demand indirectly by modifying a factor or a signal like price. In this section we discuss about a few methods such as demand-side management using Monte Carlo tree search, multi-agent residential Demand Response and demand response using a Day Ahead pricing model.

A. Demand-side management using monte carlo tree search

Monte Carlo Tree search (MCTS) is a RL algorithm that does random sampling to find optimal decisions in the decision space and builds a tree based on partial results. Originally, the Monte Carlo Tree Search (MCTS) was implemented for games and is now used as a method for optimizing demand response in smart grid[10]. The demand is managed by controlling the demand at consumer's end where the devices take autonomous decisions with the use of Monte Carlo tree search[10]. The algorithm decides when the device should run or should not based on the current price and also considers the behaviour of other devices at the time when the decision is taken. The main idea is that the devices compete for resources but at the same time have an objective to minimize the cost of electricity.

In this method, nodes represent states and edges represent actions. For the algorithm to take an decision, it needs to generate a decision tree first which is created using four major steps, which are: selection, expansion, simulation and backpropagation[10]. The authors of the paper use two variations of MCTS - one being max^n approach and second one being two agent method.

In the max^n approach, all the devices are assigned to one tree and they compete among themselves so as to maximise their own payoffs (their rewards) and achieve their objective by considering what other devices are doing and what might they do. In the two agent approach, we divide the devices and assign them to two trees in which devices assigned to one tree compete with themselves and try to achieve their objective of minimizing the cost.

The experiment in the paper[10] considered four devices that will compete by deciding the best time for them to use electricity which emphasized the shifting of demand. The experiment was run for nearly 100 days and each day was divided into 30 minute slots. Both approaches were tested with the four devices and at the end of testing it was observed that max^n approach was able to significantly reduce electricity costs.

The paper[10] showed that Monte Carlo Tree Search could be used to reduce and / or shift demand with the ultimate goal of learning optimal times to decide when to turn the device on or off.

B. Multi-agent residential demand response

The paper talks about using multiple agents to manage residential demand response through load prediction. The idea is to use reinforcement learning agents that will decide whether a device should be used or not, based on the current price and price predicted based on predicted future load. Each device on the consumer end is controlled by an RL agent which learns how to meet the energy goal of the device by a given target time at a minimum possible price.

The agents are implemented using the W learning algorithm, which is similar to Q learning but the agents have multiple Q learning policies that have different objectives and every action is associated with a W value[9]. In this multiple agent approach we have three types of agents: the load prediction agent that predicts future load, the price agent that predicts price for future load and the device agent that controls the devices.

The device agent is capable of implementing 3 policies: one for deciding whether to use the device or not, another for keeping the load of the electricity source in check, and the last one makes sure that the devices are used in lowest possible demand periods. These policies are used to make sure that the device agent shifts high demand during peak load time to off peak load time.

The experiment was done for 9 homes and devices used were electric vehicles. This experiment was done for 55 days, out of which roughly 44 days were for training and exploration while the remaining days were used for testing[9]. There were 5 scenarios created to try a combination of the three policies in the device agent and to see what the outcome would be. It was observed that using all three policies showed that the device agents, to some extent, were able to spread their charging time for the electric vehicles over 24 hours. The results[9] presented show that reinforcement learning is a suitable technique for residential demand response.

C. Demand response using a day ahead pricing model

This method unlike the other two methods mentioned doesn't directly affect the demand at the consumers end. The paper[8] proposes a pricing model that calculates day-ahead prices so that the consumer can plan when to meet their energy requirement. The authors[8] talk about real-time pricing being implemented in the real world such as a day-ahead real-time pricing (DA-RTP) tariff used by Illinois Power Company in the United States.

The method proposed in this paper[8] is based on the smart grid concept. So every consumer is equipped with a smart meter and energy management or energy consumption control system which schedules energy consumption based on the real-time prices. The smart meter communicates with the energy provider via a communication network where the energy provider sends the real-time prices and the smart meter sends current energy consumption. Using the day-ahead market energy prices, the electricity distribution network data and the consumers' energy consumption data, the proposed pricing model generates optimal day-ahead real-time prices.

The proposed pricing model aims to maximize the profit of the retailer which is defined as the difference

between the cost of selling energy to consumers and cost of purchasing energy from energy providers at market prices[8]. The model, while reaching its objective, takes several constraints into consideration, such as consumer response, flexibility of demand, minimum energy consumption, price cap, preventing energy providers from offering relatively high prices, adequate energy levels, and electricity network constraints. The model uses an optimization technique called Benders Decomposition[5] which divides the problem into two levels - one being the master problem and other being the subproblem. In this context, the master problem is responsible for real-time pricing while not considering network constraints. The solution of the master problem is sent to the subproblem which verifies feasibility of the solution on network constraints. This is iterated until a solution to the master problem is feasible.

The experiment focused on testing the proposed pricing model on 32 node radial distribution system and consumers were divided into three types: residential, commercial and industrial. The total number of consumers were 320: 10 for each node. Consumers of same type are present in the nodes. The day was divided into 3 time intervals. The market prices and load forecasts were used from electricity market data of Ontario, Canada. Three cases were considered: first using regulated pricing mechanism in Ontario, Canada, second was sending the day-ahead market energy prices directly to the consumers and third was sending the optimal day-ahead pricing calculated by the proposed model. The results showed that the use of the proposed pricing model enhanced the reliability and efficiency of the grid. It was also seen that posting optimal day-ahead real-time prices encouraged the consumers to shift their demand to periods of low prices.

This paper[8] shows that the proposed model is able to shift demand using optimal day-ahead prices.

Our approach is focused on using price as a signal that will affect the change in demand, but this is done by the agent acting as broker between distribution companies and consumers. Our model of consumers also takes into account consumers who are small-time producers of electricity.

III. Dataset

In order to train the simulated environment that we use to train the reinforcement learning agents, so that it mimics the real smart grid environment as close as possible, we use a dataset derived from the public data available at the Independent Electricity System Operator (IESO). The IESO contains various reports generated by the Ontario's power grid systems, and provides detailed reports indicating the demand, supply, tariffs, and other relevant parameters. The dataset we used had the data of the time ranging from 2010-01-01 to 2019-12-20, with each data point sampled at five minute intervals[1].

We mainly required hourly electricity consumption by the consumers and the historical tariff price so we could use it to train our LSTM model to better simulate the future parameters of the smart grid, including demand, supply, and other parameters, based on the changes to the tariffs made by the agent.

The first modification we did to our dataset is that we removed the columns that majorly consisted of NaN values, and then used a "backward fill" to fill in NaN values that remained. Next we reduced the number of columns from 47 to 13 columns by removing the columns which had a negligible correlation (whether positive or negative) with the demand and price. This was necessary since the greater the number of columns, especially the ones whose values had negligible effect on the core variables (demand and price), the harder it would be to train the LSTM to accurately simulate a smart grid in response to the agent's actions.

Finally, since our dataset was of a time range for which we couldn't get the relevant supply data, we analyzed the latest supply and demand data from IESO and analysed its correlations and calculated a supply column whose correlations with the other variables were equal to that of the latest data. This was necessary to calculate the reward, especially since our model of the smart grid assumed that most consumers were also small-time producers of electricity, which meant their demand of electricty would fluctuate. A bad scenario for the producer running this agent would be when their supply of electricity is greater than the demand, which means the agent (and therefore the producer) would have to pay the consumers to consume their electricity. Without a supply value, there is just no way to model this.

In addition, we normalized the dataset to speed up the process of learning for the LSTM to simulate the environment, and indirectly, the learning for the policy and critic networks, leading them to faster convergence. The environment and the agents all use the normalized data, with the exception of the reward function and the logging facilities - the former to give more finetuned reward signals to the agent, and the latter for debugging purposes.

IV. 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 the below criteria:

- Whether *newPrice* is within bounds.
- Whether $demand - supply$ is negative, or *newPrice* is negative.

newPrice is the non-normalized value of the price that the action has sent to the environment. We are also using non-normalized values of demand and supply for this function. Since we are using the demand and supply values of the same time step as that of the *newPrice*, we are therefore using the demand and supply values of the next time step, that is, the one that will be returned alongside the reward, to calculate the reward.

correction is simply a factor used to ensure that the *newPrice* is within the defined bounds, which ensures no exploding price or vanishing price problems. This occurs by modifying the reward to punish the agent for getting too far out of bounds.

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.

V. REINFORCE

We implemented a vanilla policy gradient algorithm conventionally called a REINFORCE algorithm using PyTorch. This reinforcement learning agent interfaced with the simulated environment, which provided us with valuable feedback about the baseline performance of policy methods for our problem 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, even for better policy gradient algorithms. 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.

VI. Agent

A. General Architecture

1) Manager Agents and Worker Agents: In a standard A3C implementation, there are no distinctions between the different kinds of agents - while they have their own copy of the policy network, the critic network, and an instance of the environment, they all do the exact same thing, which is that they update the global model (the policy network and the critic network) asynchronously.

In contrast, in our implementation, we have different kinds of agents, described as manager agents and worker agents. The manager agents hold the sole policy and critic network and is responsible for updating these networks, whereas the worker agents are actually responsible for interacting with the environment, gathering the trajectories, and calculating the losses, and signalling the master agent to update the network.

The worker agents do not have local instances of the policy and critic networks. Instead, when they calculate the loss, they signal the manager agent to asynchronously update the sole policy and critic networks.

2) Offline Workflow: In the offline workflow we are training the sole policy and critic networks to have a headstart when they will operate in the online mode on the real environment. The environment we use is the simulated one that consists of the LSTM, and the worker agents act on that environment and attempts to improve the network.

3) Headstart Modification: In the online mode where the agent interacts with a real grid, if it starts from a random policy, it will result in horrendous losses and an abyssmal performance. This means it would take a very long time for the networks to achieve an acceptable level of performance.

The headstart modification alleviates this problem by pre-training the policy and critic networks on a simulated environment in the offline mode so that the online mode doesn't start from a completely random policy and the parameters of the policy and critic networks roughly point in the right direction and therefore learns the right parameters for the actual environment much faster than if it was initiated without the headstart modification.

4) Online Workflow: In the online workflow, the agent interacts with the actual environment, that is the smart grid, and in contrast to the offline workflow, the agent waits for some number of timesteps to gather a trajectory, then uses this to update the network, flushes this trajectory and repeates this indefinitely.

In this mode the agent keeps on learning and adapting to the environment and tries to take the most optimal actions possible.

This online workflow can also be deployed on multiple smart grids simultaneously with each worker agent controlling the price of one grid and utilizing the same actor and critic networks with asynchronous updates.

B. The PyTorch multiprocessing issue

In order to implement the multi-processed nature of training, we have used the fork method in the python multiprocessing module because it is more robust for this approach of having a single copy of the networks that are being updated. When "forking" a child process, it virtually shares the policy and critic networks via a shared memory. This allows us to have a single policy and critic network that we use instead of having multiple copies. It is also necessary to implement Hogwild, which is an update style that has a near-optimal rate of convergence for the networks and outperforms any alternative methods of updating the networks that use locking.[15]

But this has a problem, in that this method of multiprocessing doesn't work on Windows operating systems.

The other issue was that loading the model along with multiprocessing had a problem that the code would get stuck. In order to solve this issue, we load the environment for every worker agent inside the child process. Only after this, we load the environment in the parent process, which contains the manager agent.

As of writing, this is an unfixed issue in the PyTorch codebase.[16]

C. The problem with Sliding Window model updates

We had devised two approaches to implement the online real-time training of the agent. One of them is the sliding window model update workflow, which has some theoretical shortcomings. To make this clear, we have provided an informal proof, given below.

We first create a queue of size n. This queue is used in the calculation of V_{target}, $V_{predicted}$, and the advantage. With the first n steps, the queue is filled with the SAR (state, action, reward) values for these first n states. Once the n^{th} state occurs, we use the entire queue to calculate the n-step advantage and the n-step TD error.

The n-step advantage uses n steps of discounted reward, plus a discounted value network output of the $(n+1)^{th}$ state (aka V_{target}) in the queue, minus the value network output for the oldest state in the queue (aka the $V_{predicted}$). The advantage formula is the same as the $V_{target} - V_{predicted}$ because we have chosen to use n-step advantage to calculate the advantage, and we are using n-step bootstrapping to calculate the loss.

Now we update the policy and the critic networks based on the gradient descent of both the networks based on the calculated loss.

Now, for the $(n+1)_{th}$ state, we remove the SAR value of the first state and append the value of the $(n+1)_{th}$ state's SAR values into the queue. Now we do another calculation of loss of policy network and the critic network.

The actor-critic algorithms are on-policy algorithms. This means the learning is specifically on-policy: the critic network must learn about and critique whatever policy is currently being followed by the actor network. And how does the critic network learn about the policy that is being currently used? It is by the n-step bootstrapped returns. Specifically, this means the trajectory of rewards (and therefore the sequence of SAR tuples) is the indirect indicator of the policy to the critic network.

For the $(n+1)_{th}$ state, however, the trajectory is not an accurate indicator of the policy being followed. The SAR tuples $1_{st}, ..., n_{th}$ are generated by following the π policy, while the $(n+1)_{th}$ SAR tuple is generated by following the π' policy, that is, a newer policy.

Since the loss function of the critic network relies on the reward trajectory to calculate V_{target}, we can say that the gradient descent is inaccurate since the V_{target} isn't based on the π' policy. Since the advantage function also relies on the reward trajectory to calculate the n-step advantage, we can say that the calculated advantage is incorrect. To clarify: the advantage function is the difference between the action-value function (Q function) and the state-value function (which is the critic network here). Since we are approximating the output of the action-value function using n-step bootstrapping, we are again using the V_{target} value here, which is an inaccurate signal of the current policy function. This means the advantage calculated is incorrect.

Since the loss function of the policy network uses the advantage function value in its calculations, we can also conclude that the optimization of the policy network is done incorrectly.

And this is just for the $(n+1)_{th}$ step of optimization using the sliding window policy.

For the $(2n)_{th}$ step, all the SAR tuples in the queue are ones generated by different policies and are used to calculate V_{target}, making it an extremely inaccurate measure, which is being used to optimize both the policy function and the loss function.

Hence, all the steps of the optimization using the sliding window policy incorrectly optimize the networks. Therefore we conclude that the sliding window policy is incorrect.

D. Episodic model updates

The episodic model updates is an alternative to the sliding window model update workflow that doesn't have the issues it does and therefore is usable with an on-policy reinforcement learning algorithm like A3C.

Here we create a trajectory using the same policy, and in an episodic manner, update the networks using the trajectory, flush the trajectory, and repeat. This ensures that the same policy is used for generating a particular trajectory, making the algorithm on-policy as it should be.

VII. Experiment Setup

A. Environment

In order for us to test the agents, we need to set up a proper simulated environment which mimics the real environment as close as possible. To achieve this, we trained our custom LSTM model on the Ontario dataset and tracked its performance, making sure that it performs as optimally as possible. As is evident from Figure 3, we can clearly see the minimization of loss as the LSTM's training progresses, which implies that the LSTM has achieved near optimal performance on the given dataset.

We considered using an environment that directly iterated over the dataset itself, however that very idea is flawed since the aim is to model the effects of the agent's actions on the environment, and one cannot do that if by simply iterating over the original dataset.

B. Training the LSTM to simulate the smart grid

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 2 shows the output of the loss function reducing as the LSTM is trained over a subset of the dataset.

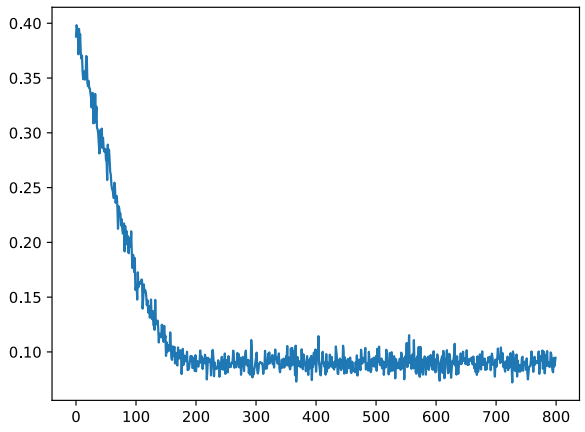

Figure 2. LSTM Training Loss per 100 episodes, using L1

After changing the loss function to MSE (mean squared error), as can be seen in Figure 3, there was a clear increase in efficiency of training, which is why we decided to use MSE as the loss function for the LSTM.

C. Action Space

Our environment supports a discrete action space in which the agent increases or decreases the current price by a fraction of the current price. For example, if the action space is [-5, -3, 0, 3, 5], then the agent can increase or decrease the price by 3 or 5 percent of the current price, or it can keep it the same by choosing the third action, which returns the value of 0 to the environment.

This method of defining the action space has allowed us to significantly reduce the training time as opposed to using a continuous action space, which allows the agent extreme flexibility and choice when modifying the price.

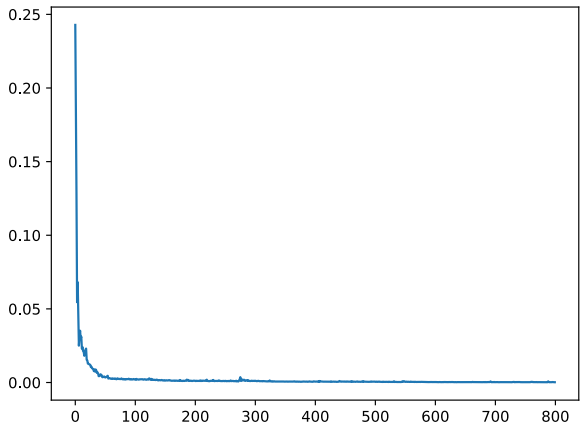

Figure 3. LSTM Training Loss per 100 episodes, using MSE

Choosing a discrete action space allows us to trade extreme flexibility in modifying the price for a reduced training time.

For the tests specifically, we use the following action space: [-20, -15, -10, -5, 5, 10, 15, 20]. By removing the action of keeping the same current price, we force the agent to learn faster since it cannot spam the action without getting useful feedback.

D. Training

The REINFORCE algorithm is a purely policy gradient algorithm, which uses the reward function directly as a reinforcing signal. For the policy network, the first layer has the shape [13, 150] units, where 13 is the input size. We then apply a TanH activation function. Next we have another linear layer of the shape [150, 8], where 8 is the size of the action space currently being used.

We trained this policy network with the Adam optimizer, with a learning rate of 0.009. The discount factor is 0.99 and an experience buffer with a maximum length of 800.

For training of the A3C agent, we have deployed it on our custom simulated LSTM enabled environment.

For the policy network, we use an input layer with a shape [13], which is the state size. We then apply a HardTanH activation function. The hidden layer consists of a linear layer of size 20 units. We then have applied a SeLU activation function, and then again a linear layer of size 30 units, and finally a softmax activation function with output whose length is equal to the size of the action space.

For the critic network, we again use an input layer with a shape [13], which is the state size. We then apply a HardTanH activation function. Then we apply a linear layer of size 15 units. We then use a SeLU activation function. Next we have a linear layer of size 10 units. Finally a HardTanH function, which gives output as a scalar, which is the value of the state.

We trained this policy and critic network with the Adam optimizer too, with a learning rate of $1e^{-3}$, the discount factor is 0.99, and the trajectory length of 50.

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 4, over the course of 5000 episodes, the agent clearly has its policy function converging to an ideal policy.

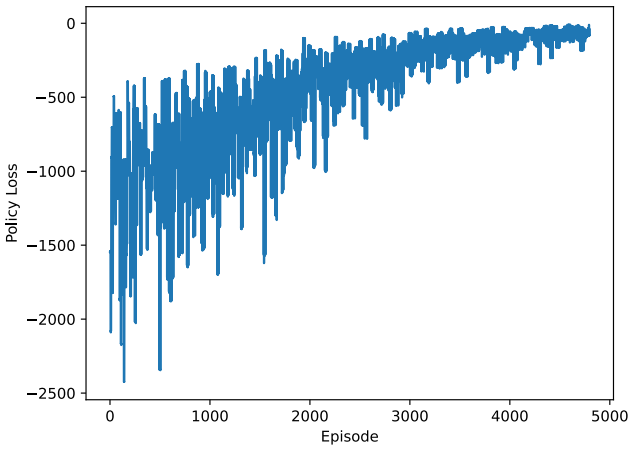

Figure 4. Maximization of objective function

The performance of the critic network gets better as the A3C model is trained more, as seen in Figure 5: the output of the loss function decreases steadily and after 4000 episodes hovers close to $0.01 * 10^6$.

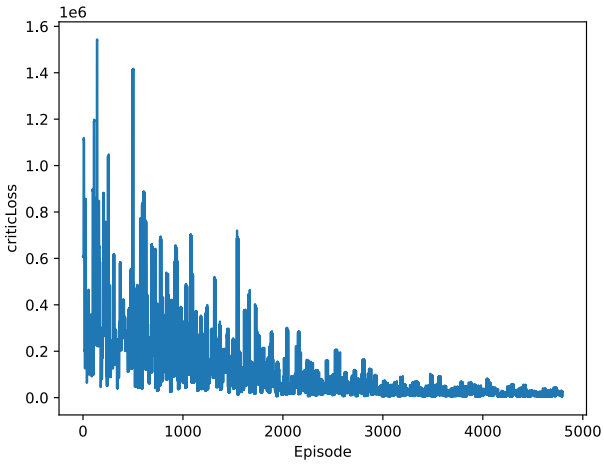

Figure 5. Critic loss

VIII. Results

A. REINFORCE performance

The performance of the REINFORCE algorithm serves as a baseline with which we compare the A3C algorithm with. Figure 6 shows the average model price set by the trained REINFORCE agent, versus the exchange (that is the $demand - supply$). As is evident from the plot, the agent does not create a buffer of exchange.

Figure 7 shows the profits accumulated by the REINFORCE agent. The plot makes it clear that the profits

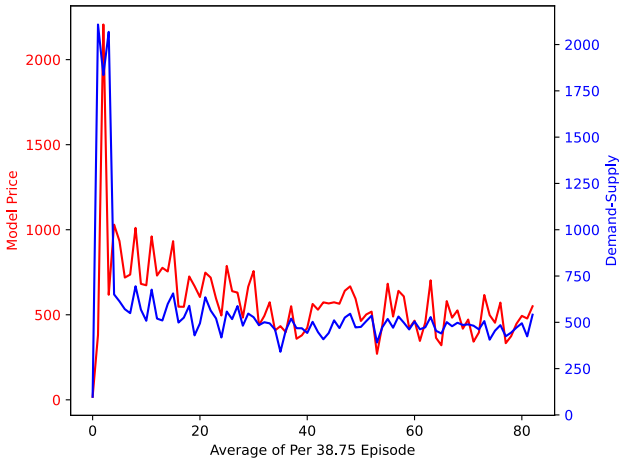

Figure 6. Average model price versus exchange (demand - supply) of the trained REINFORCE model

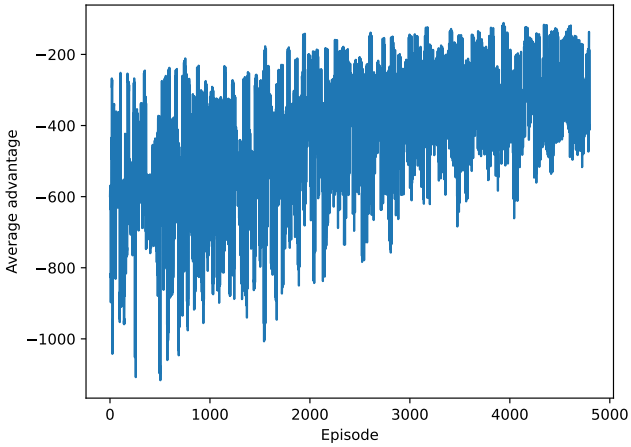

Figure 8. Maximization of average advantage

generated by the agent are better than the profits that would be generated if the agent hadn't been used. However, the performance is far from adequate, since even the maximum profits generated by the model is half that of the maximum profits generated if the agent wasn't even used. Since the order of magnitude of the profit is 1e6, this is an immense amount of profit being left at the table, so to speak.

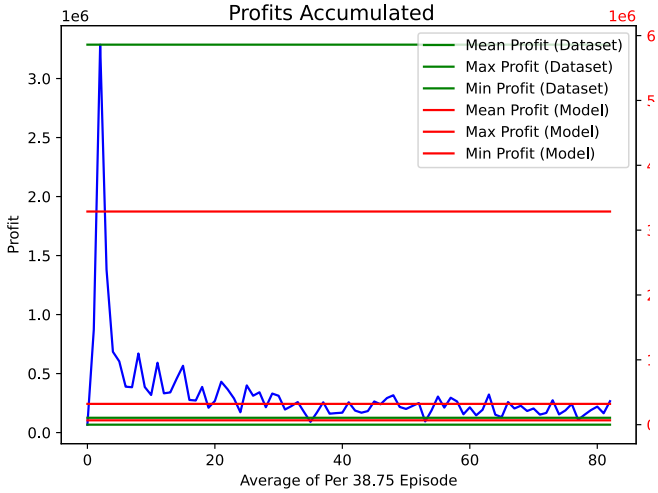

Figure 7. Profits accumulated by a trained REINFORCE agent compared with that made by a producer that did not use an agent at all

- 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.

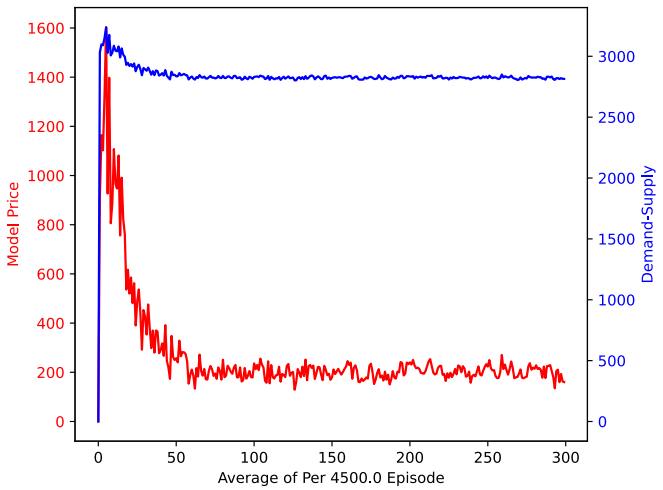

Figure 9. Demand-supply vs Model price curve predicted by our RL model keeping their product to be maximized for maximum profit based on our assumptions

B. A3C performance

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. Figure 8 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.

Figure 9 shows the results of training the agent. This has multiple implications, all positive:

C. Online performance of our A3C model

In the online real-time training of the agent, Figure 10, 11, and 12 show the results of the online training of the agent with the episodic method of model updates (that is, we do not use the sliding window approach here) and use a pre-trained network that uses the head-start approach described earlier.

The implications of the graphs are:

- From Figure 10 we can observe that the agent constantly keeps the demand greater than supply, keeping a buffer between the demand and supply, ensuring the profitability of the producer.
- While maintaining the buffer between Demand and supply, The agent tries to minimize the Price , ensuring the profitability of consumers.

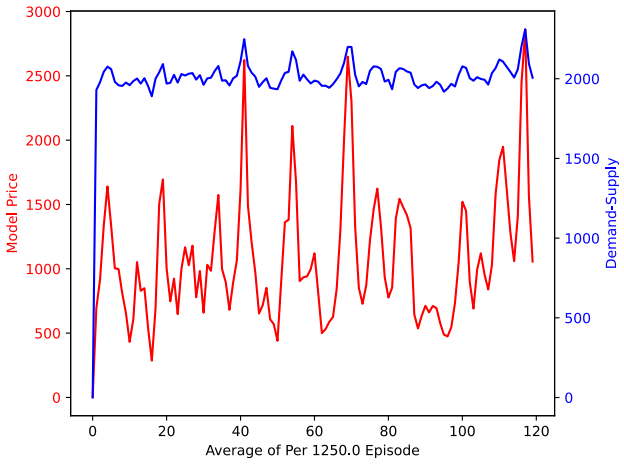

Figure 10. Average Model Price vs Exchange(Demand-Supply) curve predicted by our RL model keeping their product to be maximized for maximum profit based on our assumptions

- Figure 11 shows that the mean profit accumulated by the agent is much higher than the dataset profit, implying that the agent learns to maximize the profits while keeping the price within acceptable limits.

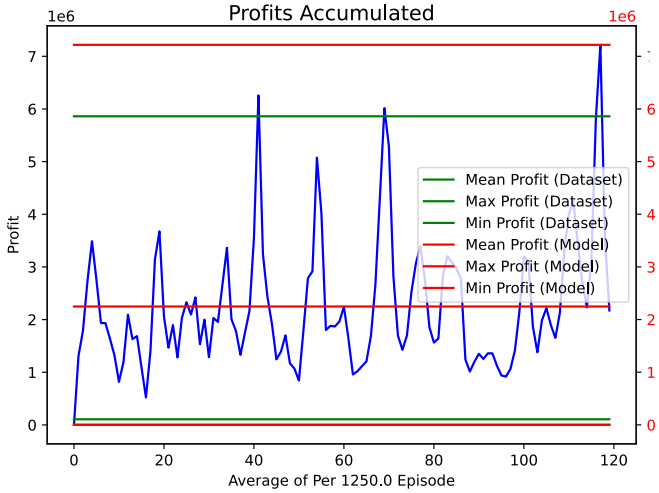

Figure 11. Profit curve along with the model's and original dataset's mean,max and min profits

- Figure 13 shows that the critic loss constantly decreases , implying that the value network is able to predict correct Value of the states, when compared to Offline mode, we can see that after 2000 episodes the Critic loss in Online mode is of the order 0.01 , as compared to $0.1 * 10^6$ in the Offline mode.
- Figure 14 shows that in online mode, policy loss becomes almost optimal after 2000 episodes, whereas it takes 5000 episodes in Offline mode for it to become optimal, this implies a better convergence rate to optimal policy in Online mode due to the usage of the Pre trained Networks (Headstart Modifications)
- It is clear from the Figure 15 that the average advantage in Online is much better than in Offline mode , and also quickly converges, implying that in Online mode the agent is taking comparatively better actions than it's Offline counterpart , and is quick to identify the most optimal actions to be taken in a state.

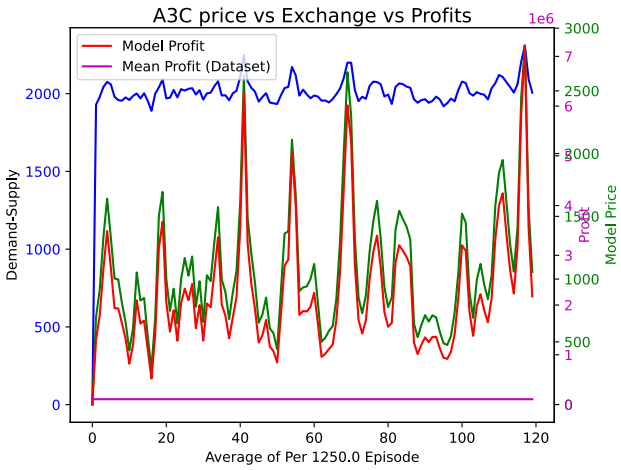

Figure 12. Average Model Price vs Exchange vs Profit curve predicted by our RL model keeping the product of price and exchange to be as maximized and thereafter ensuring optimum profit

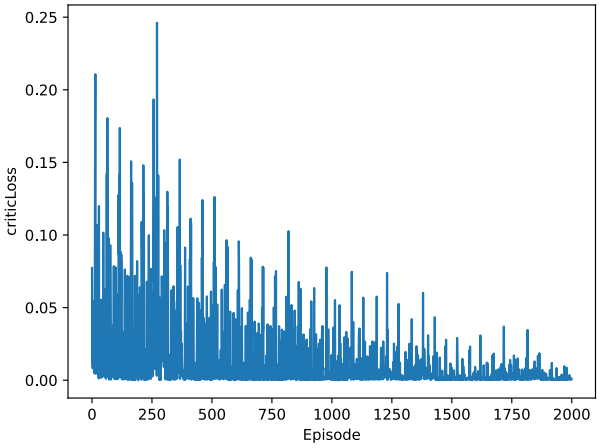

Figure 13. Critic loss curve based on our RL model

IX. Conclusion

In this paper, we presented a deep reinforcement learning approach for optimizing demand for smart grids. Our approach began with creating a custom simulated environment using LSTMs and OpenAI to train and evalute the RL algorithms we implement. For this, we used the Ontario dataset to train the LSTM model and simulate a smart grid. We then implemented the REINFORCE RL algorithm as a baseline for the performance of reinforcement learning algorithms. With the performance of the REINFORCE as a baseline, we next implemented the A3C algorithm which makes use of deep neural networks - the policy and the

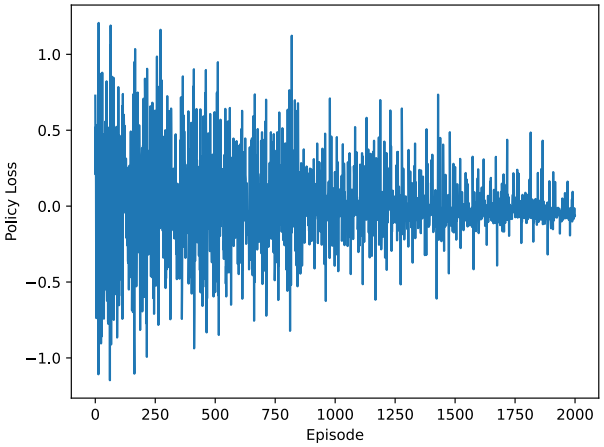

Figure 14. Policy loss curve based on our RL model

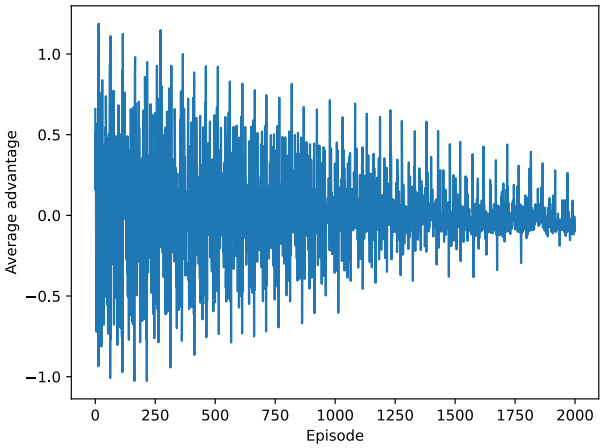

Figure 15. Average advantage

critic network - and uses the policy network to adjust the price in the smart grid, and uses the critic network to calculate the advantage, which acts as a reinforcing signal for the policy network. In order to make the online training of the algorithm more efficient, we have used a headstart modification which makes the agent use a pre-trained set of networks instead of completely random ones, thereby improving the initial performance and rate at which it achieves acceptable performance. We tested two approaches to updating the model in an online mode: the first involved a sliding window trajectory which was used to update the networks at every timestep, which was incorrect due to involvement of multiple policies in the trajectory, making the algorithm off-policy. The second solution that we use, involves creating a trajectory, updating the networks based on the trajectory, flushing the trajectory, and repeating; this solution is on-policy and produces good results. We tested both the REINFORCE and the A3C agents with the aforementioned hyperparameters and plotted our findings. The results showed that our approach outperforms that of REINFORCE by a considerable order of magnitude; it not only optimizes the profit, but also maximizes the difference between the demand and the supply, while keeping the price within the stated bounds. In future work, we intend to modify our A3C implementation to use Generalized Advantage Estimation, and implement the deep reinforcement algorithms PPO and DPPG for optimizing the demand response to gain better performance in comparison to the A3C implementation.

X. References

[1] Independent Electricity System Operator (IESO). *Power Data*. URL: https://www.ieso.ca/power-data.

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

[3] Ashfaq Ahmad et al. "Demand Response: From Classification to Optimization Techniques in Smart Grid". In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*. 2015, pp. 229–235. DOI: 10.1109/WAINA.2015.128.

[4] Mohammed Albadi and Ehab El-Saadany. "A summary of demand response in electricity markets". In: *Electric Power Systems Research* 78 (Nov. 2008), pp. 1989–1996. DOI: 10.1016/j.epsr.2008.04.002.

[5] J.F. Benders. "Partitioning procedures for solving mixed-variables programming problems". English. In: *Numerische Mathematik* 4 (1962), pp. 238–252. ISSN: 0029-599X. DOI: 10.1007/BF01386316.

[6] Greg Brockman et al. "OpenAI Gym". In: (June 2016).

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

[8] Meysam Doostizadeh and Hassan Ghasemi. "A Day-Ahead Electricity Pricing Model Based on Smart Metering and Demand-Side Management". In: *Energy* 46 (Oct. 2012). DOI: 10.1016/j.energy.2012.08.029.

[9] Ivana Dusparic et al. "Multi-agent residential demand response based on load forecasting". In: *2013 1st IEEE Conference on Technologies for Sustainability (SusTech)*. 2013, pp. 90–96. DOI: 10.1109/SusTech.2013.6617303.

[10] Edgar Galván-López et al. "Autonomous Demand-Side Management system based on Monte Carlo Tree Search". In: *2014 IEEE International Energy Conference (ENERGYCON)*. 2014, pp. 1263–1270. DOI: 10.1109/ENERGYCON.2014.6850585.

[11] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[12] K. Kuroda, T. Ichimura, and R. Yokoyama. "An effective evaluation approach of demand response programs for residential side". In: *9th IET International Conference on Advances in Power System Control, Operation and Management (APSCOM 2012)*. 2012, pp. 1–6. DOI: 10.1049/cp.2012.2175.

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

[14] Miguel Morales. *Grokking Deep Reinforcement Learning*. New York, United States: Manning Publication, 2020. ISBN: 978-1-6172-9545-4.

[15] *Multiprocessing best practices — PyTorch 1.8.1 documentation*. URL: https://pytorch.org/docs/stable/notes/multiprocessing.html?highlight=fork#asynchronous-multiprocess-training-e-g-hogwild (visited on 05/23/2021).

[16] *Multiprocessing map gets stuck if doing inference on loaded model · Issue 35472 · pytorch/pytorch · GitHub*. URL: https://github.com/pytorch/pytorch/issues/35472#issuecomment-826298121 (visited on 05/23/2021).

[17] Feng Niu et al. "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *NIPS* 24 (June 2011).

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

[19] 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.

XI. Appendix

Algorithm 1: 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: $\theta \leftarrow \theta + \alpha\gamma' G_t \nabla_\theta \log(\pi_\theta(a_t)|S_t)$
end

Algorithm 2: Reward Function

Initialize the hyperparameters *maxAllowedPrice* and *minAllowedPrice*
correction $\leftarrow 1$
if $((demand - supply) < 0)$ or $(new_price < minAllowed)$ or $(new_price > maxAllowed)$:
correction=0-abs(correction)
reward $\leftarrow$((demand - supply)^3) $*$ $(abs(new_price)^2) * correction$
profit $\leftarrow$(demand - supply)*new$_price$
return reward

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: Asynchronous Advantage Actor-Critic (A3C) Offline + Online (Episodic)

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

$$R = \begin{cases} 0, & S_t = TERMINAL \\ V_{w'}(S_t), & otherwise \end{cases}$$

  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 5: Asynchronous Advantage Actor-Critic (A3C) Online mode Sliding Window

Global parameters:- θ, w
Initialise thread-specific parameters:- θ' and w'
Initialize time step $t = 1$
Initialize deques trajectoryReward , trajectoryState , trajectoryAction
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 $T = T + 1$
  append state to trajectoryState
  append action to trajectoryAction
  append reward to trajectoryReward
end
while *True* **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.
  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 $T = T + 1$
  Pop the trajectoryState
  Pop the trajectoryAction
  Pop the trajectoryReward
  append newState to trajectoryState
  append newAction to trajectoryAction
  append newReward to trajectoryReward
  Initialize the variable that holds the return estimation

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

  Calculate$V_{predicted}, V_{target}$ and Advantage
  Accumulate gradients w.r.t. :
  $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i|S_i) * (Advantage(S_i)$;
  Accumulate gradients w.r.t. w':
  $dw \leftarrow dw + \nabla_{w'}(v_{predicted} - v_{target})^2$.
  Update asynchronously θ using $d\theta$, and w using dw.
end

Algorithm 6: 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