Understanding Deep Deterministic Policy Gradient (DDPG) used in air-fuel sensors for Proton Exchange Membrane (PEM) fuel cells

Pavan datta Reddy, *ECE*

***Abstract*—This term paper seeks to delve into the various algorithms, specifically related to those of Deterministic Policy Gradient algorithms (DDPG), and their relation to helping maintain an optimal oxygen excess ration (OER) within the Proton Exchange Membrane (PEM) fuel cells. There are two types of algorithms we will explore deeply related to DDPG, which are fuzzy PI controllers, also known as SIT2-FPI controllers that have a DDPG architecture alongside that fine-tunes the gains, and multi-role exploration strategy distributed deep deterministic policy gradient (MESD-DDPG) algorithm which fine-tunes a normal PI controller. In general, the goal of this paper is just to establish the general algorithms used for both these DDPG algorithms and then explore their specific characteristics. We notice that these DDPG-related algorithms have their own architectures, but they use fundamental Reinforcement Learning (RL) concepts, such as action-values methods, such as Q-learning. Specifically, we will be looking at these action-value methods under a given policy, as many of these DDPG-related algorithms utilize these fundamental concepts. Also, we will establish the basic system of the PEM fuel cells, so there can be a high-level understanding of what parameters are necessary to consider.**

CONTENTS

I. INTRODUCTION

A. Basic Concepts

THERE are certain concepts that are essential to DDPG algorithm, such that they need to be considered. Based on the concepts introduced in class, such as Q-learning algorithms, the DDPG algorithm uses parts of these algorithms and extends them to help optimize training and results. Let us start off with the Bellman equation when it comes to Q-learning. In general, just how we can formulate a Hamiltionian-Jacobi-Bellman equation for a specific control systems problem, which we try to optimize by getting an optimal control by using certain methods depending on the problem, Q-learning can also be formulated using the Bellman equation.

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma * maxQ'(s',a') - Q(s,a)] \tag{1}$$

In general, in (1), α is the learning rate, at which the error is multiplied, while γ is a value between [0, 1] that helps the agent not focus too much on future rewards, and this value is known as the discount rate. But, let us explore this idea of these action-value functions further. First, let us introduce this concept of expected return. In general, when it comes to reinforcement learning, we want to maximize the reward in every time step we take, so we can define the expected return as the cumulative sum of all of the rewards at each of the time steps.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2}$$

In general, we want to look at a continuous case, so we assume that we take infinite time steps; however, if the problem were discrete, we would take a finite amount of time steps. Hence, in the continuous case, T is infinite or $T \to \infty$. However, since we do not want our agent to focus too much on these future rewards, but the rewards we get at that time instant, we should introduce a discount rate for the expected return.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3}$$

The expected return equation in (3) can be simplified to the following in (4).

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{4}$$

But, before we go further on explaining Q-learning, let us talk about another concept in Reinforcement Learning called the policy. In general, the policy can be defined as an agent's general strategy. In general, it is basically what an agent does to get and maximize its reward. But, if we are more technical about this definition, we can say that the policy is just a mapping of a certain state to its action. So, if you were in a certain state what is the probability that you would do this action to get this reward. In Q-learning, we have Q-learning function under a given policy.

But, to introduce some nuance and to understand the significance of having a policy in the first place, let us define a Q-learning function that does not have a policy (off-policy). We could utilize the form in (5).

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma * maxQ(S_{t+1}, A_{t+1}) - Q(S_t, At)] \quad (5)$$

This specific algorithm is an off-policy algorithm as we directly approximate the optimal q, which we can call $q*$. We will talk about $q*$ in a moment. But, besides the essence of the off-policy algorithm, let us also note that the expected return for Q-learning is defined in (6).

$$G_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) \quad (6)$$

So, with this consideration of the expected return for Q-learning, we could say that the form of the off-policy Q-learning algorithm in (5) can have equivalent form in (7).

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)] \quad (7)$$

Now, instead of exploring the off-policy algorithm, let us now discuss a Q-learning function under a given policy π. But, let us also explore these questions: Why are we focusing on model-free methods rather than using functions that require a model? Why should we use an algorithm under a given policy $\pi(s, a)$?

When it comes to the problem of finding the optimal oxygen excess ratio within the PEM system, the model is complicated within itself that it requires a high-dimensional state-space. To even create a model like that is computationally expensive and challenging to do so in the first place. As a result, for this particular problem regarding the PEM system, we have a model-free system, as it is not as complicated and gets us quick results in continuous time. In general, model-free systems are known as Monte-Carlo, and we prefer them over the modeled systems or Dynamic Programming.

But, let us also answer the question of why we use a policy in the first place. In general, we want to introduce stochastic processes using the policy. In general, if we did not have a policy, we are able to predict an action given a state perfectly, which makes the system deterministic. However, in real life, such accurate prediction is not possible in the first place. Instead, we want to look at the probabilities of a certain event happening. As a result, stochastic processes, specifically Markov Decisions processes in which a policy, such as $\pi(s, a)$ is based on, introduces this idea of complexity which closely resembles the real world. Hence, we want probability distribution of the actions given we are in a specific state. For example, let us say there is an excess of oxygen within the PEM system, which is a state of the system. The policy helps us determine what the probability of doing a certain action, such as allowing more reactant to flow in to stabilize the oxygen excess ratio. If the probability is high, then we do the action, and we can reinforce that action through a reward mechanism, which betters the action-value function as well as the given policy. In general a certain policy, or Markov Decision Process has the tuple of the form (S, A, P, R), which indicates the state, action, transition probability, and reward.

Let us now consider the action-value function under a given policy $\pi(a, s)$ in (8). Generally, in (8), we consider the expected value of the expected return given a state and action.

$$q(s, a) = E_\pi(G_t | S_t = s, A_t = a) \quad (8)$$

However, considering what the expected return is equal to, we can also write action-value function under policy π in the form given in (9).

$$q(s, a) = E_\pi(R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a) \quad (9)$$

If we want to find the optimal action-value, which we called $q*$ a few moments ago, we can model it using the Bellman Optimality equation as used in (10).

$$q^*(s, a) = E_\pi(R_{t+1} + \gamma \max_{a+1} q^*(S_{t+1}, a+1) | S_t = s, A_t = a) \quad (10)$$

Though it would be nice to get the optimal action-value and policy, it is computationally expensive to do so. In general, when it comes to the PEM system problem, we want to use equation (9) that captures the action-value function under a given policy.

All in all, Q-learning and the policy in which the agent is under are important components in the DDPG algorithm. Note that this specific form in (1) and (9) is used in DQN learning, which is a suboptimal algorithm for the PEM system, but note that DDPG utilizies parts of DQN, which implies that the basic components of Q-learning are relevant in DDPG-related algorithms. But, besides the general concepts of Q-learning, which is a part of Reinforcement Learning, let us talk about the actual system in which we are trying to implement efficient algorithms for.

B. PEM system overview

In the PEM system, we have several different components that we have to consider and model through the use of differential equations. In general, let us talk about what we have to track and optimize within the PEM system. In general, since we have many different subsystems within a vehicle that interact with the PEM system, there is always a chance that there is always more or less oxygen than needed in the PEM system when a certain subsystem operates. For example, when a car accelerates, the PEM system will heat up more than necessary, so we need a reactant to flow into system as fast as possible to make sure the PEM fuel cells are cooled down and operate properly. We have to make sure that there is enough oxygen in the PEM system, such that we do not have oxygen starvation which could ruin a system or an oxygen excess which could destroy the PEM system completely. More specifically, the oxygen excess problem in PEM system is one of the major problems when it comes to air flow within the PEM fuel cells, so we need controllers and algorithms that could fine-tune these controllers to help direct air-flow accordingly. The oxygen excess is the actual variable we are trying to optimize within our system through the use of the algorithms we are using. But, besides the general problems that we are trying to solve within the PEM system, let us also consider the specific components within the PEM fuel cell system. In general, The PEM system can be divided into the following: the cathode, the anode, the supply pipe, the return pipe, and the air compressor. An example of the PEM fuel cell cell is shown in Fig. 1.

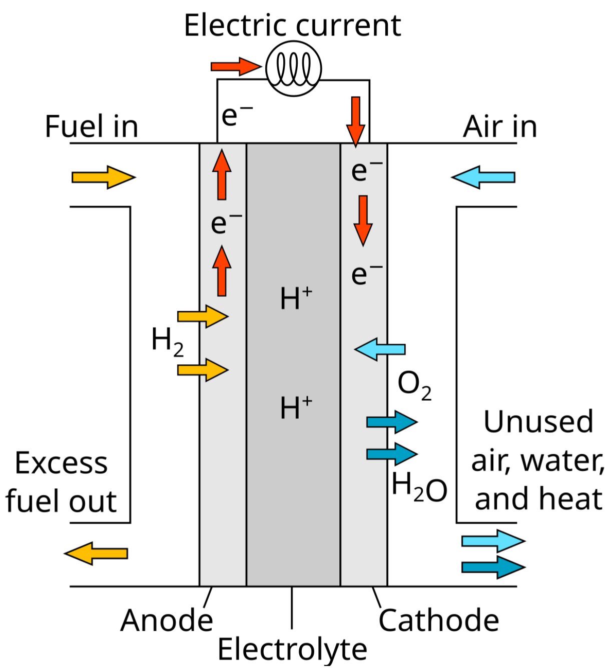

Fig. 1. Here is a model of the PEM Fuel Cell. The anode side considers the flow of H_2 and the cathode side considers the flow of O_2.

The cathode is the part of the system where nitrogen and oxygen are constantly flowing into and flowing out of the cathode, which is the part that provides negative charge in the PEM fuel cell. In general, let us consider the masses of oxygen and nitrogen, and let us call them m_{O_2} and m_{N_2} respectively. Let us call the flow of oxygen and nitrogen into the cathode part of the PEM fuel cell $W_{O_2,in}$ and $W_{N_2,in}$. Let us call the flow of oxygen and nitrogen out of the cathode part of the PEM fuel cell $W_{O_2,out}$ and $W_{N_2,out}$. Let us also consider the oxygen excess ratio (λ_{O_2}), which is the variable we are trying to optimize at the end of the day. In general, following the rules of ideal gas laws as well as the conservation of energy, the rate, which is the amount of mass flowing in the system per unit time, at which the masses of oxygen and nitrogen travel within the cathode of the subsystem can be modeled as follows in (11) and (12). Also, the oxygen exccess ratio can be seen in (13).

$$\frac{dm_{0_2}}{dt} = W_{O_2,in} - W_{O_2,out} - W_{O_2,consumption} \tag{11}$$

$$\frac{dm_{N_2}}{dt} = W_{N_2,in} - W_{N_2,out} \tag{12}$$

$$\lambda_{O_2} = \frac{W_{O_2,in}}{W_{O_2,consumption}} \tag{13}$$

Besides the cathode part of the PEM fuel cell system, let also consider the anode part. Hydrogen is used in anode part of the system to help reduce the pressure flow difference between the anode and cathode. So, in order to control the flow of hydrogen into the anode, we use a valve that regulates the flow of hydrogen into the subsystem. Although, in an ideal world we should just measure the pressure difference between the anode and cathode, the measurements cannot be weighed properly through the sensors, so we instead use pressure controllers for both the anode and cathode part of the PEM system. In general, let us call the pressue of the return pipe, which is part of the cathode part of the system p_{rm} and the pressure in the supply part of the system, which actually directs the hydrogen to the anode p_{sm}. We can call the pressure of the anode and cathode p_{anode} and $p_{cathode}$. Then, the controller measures $W_{an,in}$ in (14), and note that K_1 and K_2 are emprically measured coefficients.

$$W_{an,in} = K_1 * (K_2 * p_{sm} - p_{anode}) \tag{14}$$

The supply pipe which connects to the cathode and further extends to the air compressor also has a flow rate of gases that need to be modeled. The mass flow that enters into the supply pipe is called $W_{compressor}$ and the mass flow that exits out of the supply is called W_{sm}. As a result, then the mass flow of the gasses and the change in pressure of the pipe with respect to time can be modeled as follows in (15) and (16). Note that in (16), θ_1 is a constant and T relates to just the temperatures of the air compressor and the supply pipe.

$$\frac{dm_{sm}}{dt} = W_{compressor} - W_{sm} \tag{15}$$

$$\frac{dp_{sm}}{dt} = \theta_1 (W_{compressor} * T_{compressor} - W_{sm} * T_{sm}) \tag{16}$$

The return pipe can be modeled as follows in (17).

$$\frac{dp_{rm}}{dt} = \theta_2 (W_{cathode} - W_{rm}) \tag{17}$$

The air compressor also has its own specific modeled system, but for the purpose of this scenario of explaining the DDPG algorithm which utilizes Q-learning techniques along with a certain policy, this part is not very important to reveal. The other parts just reveal the general modeled systems of the different parts of the PEM fuel cells. But, the most important variable that we must optimize through the use of our DDPG algorithms is the oxygen excess ratio, as this problem is the most significant when it comes to managing the air flow with the PEM system.

C. Understanding DDPG

In general, let us talk about the general concept used within the DDPG algorithm. There are two neural networks used within the DDPG, which are the policy in which the agent goes through and the Q-network that gives us the best action determined from a state under the given policy. Let us call this policy $\pi(s)$ and the network $Q(s,a)$. Let us say that the parameters for both the policy and the Q-value have the parameters μ_1 and μ_2 respectively, so we have an optimal policy and Q-value went want to approach through infinite steps which we can call μ_1' and μ_2'. In general, we want to maximize the expected return or the Q-value, $Q(s,a)$. Let us call the expected return J as defined in (18) by finding the expected value of the reward.

$$J(\mu_1) = E[R_t] \tag{18}$$

The equation (18) closely resembles the definition of the expected return in (3). In this equation, we do not have a discount rate, but the concept is the same.

Since it is hard to numerically solve the maximal expected return, let us use gradient descent to find the expected return, which we can call ∇J.

$$\nabla J(\mu_1) = E[\nabla Q^{\pi}(s,a) | \nabla \pi(s)] \tag{19}$$

$$Q^{\pi}(s,a) = E[R_t | s, a] \tag{20}$$

Here, we can see that (19) and (20) resemble the definitions introduced about expected return and an action-value function under a given policy. Specifically, these equations closely resemble (9), but in this scenario we are finding the expected return of the action-value from a given state and action rather than just the expected value of the expected return. But, the concept is similar.

Also, to introduce this idea of determining how well our predictions are for the action-value function, we can use loss functions, but in this paper we are mostly focusing on the policy network and the action-value network. In general, we can just say that loss function calculates the difference of the expected value of the difference squared of the expected state-action and the predicted one we calculate.

$$L(\mu_2) = E[(y_t - Q(s_t, a_t))^2] y_t = R_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) \tag{21}$$

where y_t is defined in (22).

$$y_t = R_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) \tag{22}$$

As, we want to approach the targets μ_1 and μ_2 through every time step. In other words $\mu_1' \leftarrow \mu_1$ and $\mu_2' \leftarrow \mu_2$ as T, which is the total time, approaches ∞. Note that, if there is noise in the system, we also have to consider the noise when μ_1 and μ_2 approach their optimal value, but we are just demonstrating the concept here.

Besides, talking about the math behind the DDQG, let us talk about the agents involved in the process. We have both an **actor** and a **critic** in the DDPG algorithm. The actor-network is determined by the policy, and the critic-network is determined by the action-value. In general, we have to have separate networks.

This is similar to our action-value function under a given a policy π, as we have a network where we have an agent that does a specific action in an environment, and we get state S_t and a reward R_t for that state.

Also, when it comes to training on data, we have a data buffer to consider. Just how we store information similar to Markov Decisions Processes with tuples, we have tuples of information about the current and next states, the action and the reward. The data buffer helps the actor and critic networks reach target action-value and target policy, which is $\mu_1' \leftarrow \mu_1$ and $\mu_2' \leftarrow \mu_2$ respectively.

In Fig. 2, we get a big picture overview of how DDPG works, with the actor policy network and the critic action-value network.

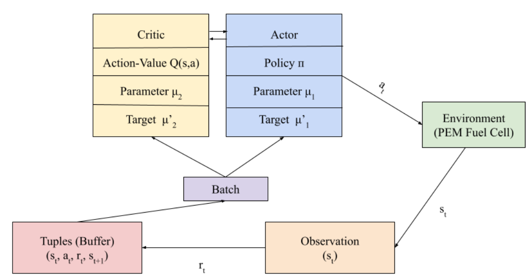

Fig. 2. Here are the components of the DDPG Algorithm with respect to the PEM Fuel Cells.

Now, with this in mind, the main difference between DDPG algorithm and that of the DQN algorithm then, which we have outlined in (1) and (9), is that we have two agents, not just one.

In general, in the DDPG algorithm, the actor does an action within a given environment and gets state feedback S_t along with a reward R_t, which updates the actor network. However, the critic-network also helps update the actor-network. So, the state feedback which helps update the policy π^{μ_1} in the actor-network as well as the state-value from the critic-network help the actor-network overall.

But, although DDPG is simple, it is not the specific algorithm we are using within the PEM system. Instead, we will be using a certain kind of DDPG-inspired controller or fine-tuner.

II. Methods of DDPG-related Algorithms in PEM

A. SIT2-FPI with DDPG adjustments

Before we talk about how DDPG can help us fine-tune a fuzzy-logic PI controller, let us first talk about what fuzzy logic is in the first place.

In general, fuzzy logic deals with problems that do not have a binary true or false. Instead, the result could be closer to true and farther from false, closer to false and farther from true, or equal to false and equal to true. But, the general idea is that their is a certain true or false if the value is closer to true rather than false and vice versa.

Fuzzy logic can be used within the PEM fuel cells because of the fuzziness of determining whether we need to change the oxygen-excess ratio given that something happens within the environment, such as accelerating the vehicle.

It is hard to say whether we need to change the action of the agent based on environmental changes, as it is not a simple true or false. As a result, that is why having fuzzy logic is useful, especially in something like a PI controller.

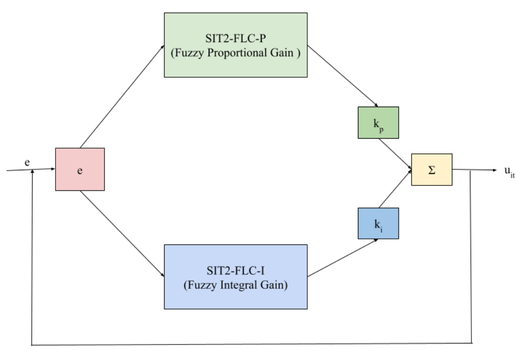

Fig. 3. Here are the components of the Fuzzy PI controller.

A fuzzy PI controller has the sample principles as a normal PI controller. You have normal proportional gain control and integral gain control based on how much error their is between the desired result and the state feedback. However, now since the outputs are fuzzy logic, we want to pass the error into SIT-FLC blocks that help give us the proprtional gain and the integral gain, k_p and k_i respectively. The output of the control can be defined as u_{it} and (23) helps define that result. In (23), e is represented as the error, and k_u is represented as the inverse of the error. Also in Fig. 3, we show the internals of how a fuzzy PI works.

$$u_{it} = k_u(k_p e + k_i \int edt) \tag{23}$$

But, now this question arises: Why even use the DDPG algorithm to help fine-tune the proprtional gain coefficient and the integral gain coefficient? The answer to this question is that just using a PI or a fuzzy PI controller by itself is not enough because you want the ability to update your actions when you iterate through infinite time. You do want to update a given system, such as the PEM system if there is a fluctuation. We want to learn from our previous actions, so we want to better our strategy, or more technically our policy based on previous mappings of actions to states. The system is constantly changing and we want to capture that change by updating our strategy, and a fuzzy PI controller does not help collect all previous actions mapped to states and make decisions based off of that. Generally, what we are referring to here is a stochastic process, and the process of learning from our previous states and actions is called online learning.

In general, online learning deals with datasets that are growing in number as we continue to interact with the system, and we need to fine-tune the fuzzy PI gains, k_p and k_i in such a way where we have an optimal policy or strategy where we make better actions to obtain the best state in the least amount of time based on a growing dataset of information. In general, having this growing dataset and optimizing based on training on the dataset helps make the system more adapatable. The fuzzy PI controller by itself cannot do this adaptation alone.

Also, the dataset information, specifically related to this stochastic process can be stored in a tuple, similarly related to the Markov Decisions Processes for Q-learning. The tuple has the form (s_t, a_t, r_t, s_{t+1}). The form of this dataset is used specifically when implementing DDPG.

The whole concept of DDPG is similar to that mentioned in section I-C, where we start to understand the DDPG algorithm. In general, with respect to the PEM fuel cells, the DDPG algorithm should fine-tune k_p and k_i as much as possible, so we could get the desired action-value that results in a given state, which is stored in the tuple (s_t, a_t, r_t, s_{t+1}) as fast as possible.

In general, this is how the DDPG algorithm is implemented in the SIT2-FPI controller:

Step 1: Let us first initialize the actor-network with policy $\pi(s|\mu_1)$ and the critic-network with action-value $Q(s, a|\mu_2)$. As a reminder μ_1 and μ_2 refer to the weights to corresponding networks.

Step 2: Set up target weights, such that through the infinite iterations $\mu_1' \leftarrow \mu_1$ and $\mu_2' \leftarrow \mu_2$.

Step 3: Set up an empty dataset space buffer that will contain the tuples (s_t, a_t, r_t, s_{t+1}).

Step 4: For each episode receive an initial observation state, which we can call s_1. For each time interval until T, where $T \to \infty$, select an action, a_t that the actor network should perform and observe the next s_{t+1}. Compute the reward, r_t for the particular a_t and s_{t+1}. Store all of this information in

(s_t, a_t, r_t, s_{t+1}). Sample the datasets from the dataset space buffer. Calculate the expected state-value based on optimal weights, and call this calculation y_i. Calculate the loss with the equation in (21). Update the policy gradient as described in equation (19). End the time interval loop and end the episode loop.

Note that our datasets will look like the following in the sample dataset space buffer, which we call R.

$$R = (s_1, a_1, r_1, s_2), (s_2, a_2, r_2, s_3), \\ (s_3, a_3, r_3, s_4), ..., (s_T, a_T, r_T, s_{T+1}) \quad (24)$$

Once we go through algorithm for each time-interval in one episode, we can fine-tune the gains, k_i and k_p respectively, and finally use SIT2-FPI controller to help send an output to the PEM Fuel cells to regulate the λ_{O_2}, which remember is the oxygen excess ratio. Then we send the oxygen excess ratio and compare it with the optimal excess ratio, λ'_{O_2} to get the error, e.

$$e = \lambda'_{O_2} - \lambda_{O_2} \quad (25)$$

Then, we can send this error through the fuzzy PI controller to further update the system when there is a change, but the coefficient gains k_i and k_p will be updated through the use of of the critic network and the actor network represented by the action-state value and the policy in DDPG.

Note that for the sake of simplicity, we did not include any sort of noise within the system, but note that in real life, when considering the PEM Fuel Cell Stack, we need to consider the Ornstein-Uhelnbeck (OU) Noise, which can actually affect how we approach our target weights for the policy and the action-value.

In Fig. 4, we are considering the SIT2-FPI with DDPG without the noise, and you can see how all the components work together, specifically the mechanism of how the DDPG algorithm fine-tunes the fuzzy PI controller.

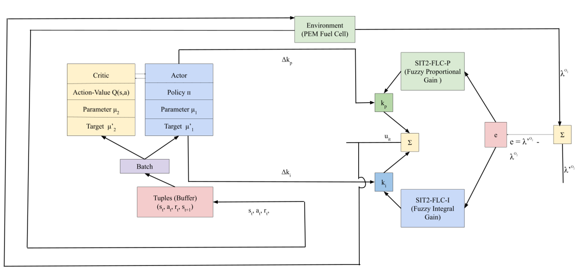

Fig. 4. Here are the components of the SIT2-FPI with DDPG adjustments.

But, let us also find another way to fine-tune a PI controller through another form of DDPG for the same problem of maintaining a desired λ_{O_2} in the PEM Fuel Cells Stack system.

B. MESD-DDPG

This algorithm is similar to that used in SIT2-FPI, where we try to update the gain coefficients, k_p and k_i through a DDPG-related tuner. But, the difference between MESD-DDPG and the SIT2-FPI is that we used a normal PI controller instead of a fuzzy-logic one. But, also in DDPG usually you have one actor that does actions and receives feedback, which results in a lack of more observation of the states. So, the solution to this problem, specifically implemented by MESD-DDPG, is to have multiple RL frameworks and a diverse exploration policy. In general, we have followers, explorers, and learners within the MESD-DDPG that help us brings about more observations about the environment, which brings further diversity into the policy and action.

In general, when it comes to training on the data, we usually have one actor-network and one critic-network. However, in MESD-DDPG, we have one actor-network and three critic-networks, which helps us bring about a more diverse policy. So, each of these three networks are once again updated with the sample batch provided by the tuples given in the form (s_t, a_t, r_t, s_{t+1}). The actor-network action-value is updated with these tuples but also with the outputs of the multiple critics. In other words, with the policy updates of the critic-network, we are able to update the actor-network's action-state value under these critics' policies accordingly. These networks are incorporated under learner, and the difference between the MESD-DDPG and DDPG just from the networks, is that we have more critics that provide a more diverse policy that ever before, as we have more observations of the environments with more critics.

However, now let us talk about the followers and explorers who actually go out of their way to collect the tuples (s_t, a_t, r_t, s_{t+1}), which the learner uses in the actor-network and critic-networks. The explorers and followers observe the PEM system and send the necessary data or tuples into dataset buffers. These reinforcement algorithms that are used are DDPG, OU, SAC, and Gaussian. Each of these algorithms, except SAC which are only used as explorers, have explorers and followers that observe the PEMFC and provide the necessary tuples that would be sent to the buffers. In general, there are two types of explorers, which are DDPG-related and SAC-related, and each of these explorers have their own followers.

The SAC algorithm deals with enhancing randomness by increasing the entropy. In general, we want increase the randomness of picking an action through training, so we could reduce any sort of bias, and increasing the entropy helps with that. The SAC algorithm has its own policy network, two action-value networks, and also two state-value networks. In general the policy network, which can be called π helps map an action based on the state of the system. The state-value functions which we can call $V(s)$ and $V'(s')$ outputs the current state and the next state. The action-value functions which we can call $Q_1(s,a)$ and $Q_2(s,a)$ output the value of the action selected with the policy π. So, the action values are under a given policy π established in equations (1) and (9).

Let us now talk about the DDPG-explorers. These explorers use three different types of exploration, specifically greedy strategy, Gaussian noise, and OU noise. When it comes to the greedy strategy, specifically ϵ-greedy strategy, we want to balance exploration and exploitation, where we could explore to improve our observation of the system, or we could maximize the reward with the current action-value. In general we can define the action selection of ϵ-greedy as follows in (27). Note ϵ is a probability. Note that the maximum action-value is equal to the policy.

$$\max_a g_\pi(s,a) = \pi \tag{26}$$

$$a(t) = \begin{cases} \pi, & 1-\epsilon \\ a\epsilon A, & \epsilon \end{cases} \tag{27}$$

Now, let us consider the OU noise and the Gaussian noise, which are stochastic in nature. We won't explore any sort of noise in this paper, as we are considering the DDPG related algorithms with the policy and action-value to be more significant. But, in real life, not in a system such as that of the PEM Fuel Cells Stack, you have to consider the noise.

Now, let us talk about the general followers. The followers are inspired by this idea of human evolution survival. In general, you have two types of followers, OU-follower and Gaussian-follower. In general these followers help increase the possibility of exploring more value samples.

Therefore, we have DDPG explorers (ϵ-DDPG-explorer, OU-DDPG-explorer, $Gaussian$-DDPG-explorer, and SAC-explorer), and two followers (OU-follower, $Gaussian$ follower) relating to the noise which provide more value samples.

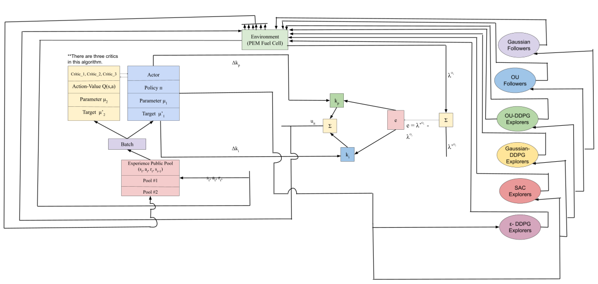

Fig. 5. Here are the components of the MESD-DDPG algorithm

If you look at Fig. 5, you can see all the components of the MESD-DDPG Algorithm, and all of the explorers and followers involved.

Also, note that there are two public experience pools, which are just dataset buffers. There are some rules dictating which pool should be used for learning, but for the sake of this paper we just want to focus on the essence of the DDPG algorithm, with its critic-networks and actor-network.

III. Analytical and Numeral Results of DDPG

A. Analytical Results

Regarding the PEM fuel cells and maintaining the oxygen excess ratio, λ_{O_2}, there are some simulations of using a SIT2-FPI controller that is fine-tuned through the use of the DDPG-controller algorithm. In general, in [1], the experiment went as follows. They used four different kinds of controllers: PI, Model-Free SMC, SIT2-FPI, and DDPG based SIT2-FPI. Then, they compared the controllers' performance along a few different cases. We will be focusing only on the cases regarding λ_{O_2}. Along these different cases, they measured the λ_{O_2}, the Stack Voltage, and the Stack Power. But, in essence, considering how much of a problem oxygen excess is, we only focus on the oxygen excess ratio results with the different controllers:

Case 1 (Constant λ_{O_2}): In this case, the performance of all of the controllers seems familiar, but in the long run the DDPG based SIT2-FPI controller does the best. When it comes to the λ_{O_2}, there is accurate tracking with DDPG based SIT2-FPI controller. Even when there is an overshoot in the ratio, even though we have a constant ratio for the most part, specifically at five-second time interval, the SIT2-FPI detects that change.

Case 2 (Variable λ_{O_2}): Once again the performance of the controllers are subtle. They seem to be the same overall, but overall, you can see the SIT2-FPI controller performs the best along different control surfaces.

Besides, the experiment regarding the PEM fuel cells with the SIT2-FPI controller, let us also look at the experiment regarding the MESD-DDPG-PI in [2]. The experiment is similar in the sense that we use different controllers and compare their performance across different components, such as Stock Voltage, Oxygen Excess Ratio, and the proportional-coefficient, k_p. We only want to focus on λ_{O_2}. The other controllers in which the MESD-DDPG-PI controller is compared with are the TD3-PI, DDPG-PI, DQN-PI, REF-PI, and DDPG.

In general, from the experiment presented in [2] the MESD-DDPG-PI controllers seems to have a better response overall compared to the other controllers. Comparing at different time intervals, the MESD-DDPG-PI seems to stabilize and get to results quicker than the other controllers.

B. Numerical Results

In general, there are not a lot of numerical results given in [1] and [2], regarding the oxygen excess ratio, λ_{O_2} and the controllers. In [2], we do get a chart of the rise time, steady time, and overshoot of the different controllers in regards to λ_{O_2}. In general, in terms of rise time and steady time, the MESD-DDPG-PI performs the quickest, with 241/s and 3.67/s respectively. Finally, the overshoot is the lowest at 17.009%, indicating that the system stabilizes quickly with the MESD-DDPG-PI controller. As a result, we can say the fine-tuning a normal PI controller, with a DDPG algorithm, that utilizes action-value functions and optimal policy which affect these action-value function, performs the best out of all of the other controllers in this experiment, just from these results of stabilization within a short amount of time and minimal overshoot even when there was an excess of oxygen released into the system randomly and quickly.

IV. Conclusion

In general, the PEM fuel cell system has a problem of having excess oxygen, hence resulting in us optimizing the excess oxygen ratio λ_{O_2}. In general, DDPG algorithms help us with fine-tuning PI controllers that help get us to the expected λ_{O_2}. Regarding one way of fine-tuning PI controllers, specifically a fuzzy logic PI controller, we can use DDPG to fine-tune the gain coefficients, k_i and k_p within the fuzzy PI controller. On the other hand, there was also another way to use DDPG when fine-tuning the gains. Instead of using a fuzzy PI controller, we use a normal one, but now we use the MESD-DDPG algorithm to fine tune k_i and k_p, such that we can stabilize the system quickly and reduce the overshoot when there is a change in any of the variables but more specifically with λ_{O_2}. We also determined the DDPG related controllers, in experiments performed in [1] and [2] perform better than other controllers in different fields such detecting λ_{O_2} or voltage and so on. The DDPG algorithm use fundamental concepts in Reinforcement Learning, such as action-state function under a given policy. Also, note that we are doing model-free because the PEM fuel cell system is already complex itself such that it would require a high-dimensional state-space that is computationally intensive and expensive. All in all, the general idea of using the DDPG algorithm to fine-tune the PI controllers are implemented in ing the oxygen excess ratio of λ_{O_2} in the PEM Fuel Cells System.

References

[1] M. Gheisarnejad, J. Boudjadar, and M. Khooban, “A new adaptive type-II fuzzy-based deep reinforcement learning control: Fuel cell and air-feed sensors control,” IEEE Sensors Journal, Vol. 19, 9081-9089, 2019.

[2] J. Li and T. Yu, “A new adaptive controller based on distributed reinforcement learning for PEMFC air supply system, Energy Reports, 1267-1279, 2021.