THE HONG KONG POLYTECHNIC UNIVERSITY
DEPARTMENT OF ELECTRICAL ENGINEERING

Project ID: FYP_34

Machine Learning in Economic Dispatch Problem - Deep Q Network based Reinforcement Learning Approach

by

Fung Chun Yin
13027548D

Final Report

Bachelor of Engineering (Honours)
in
Electrical Engineering

Of

The Hong Kong Polytechnic University

Supervisor: Dr. C.W. Yu

Date: 16/3/2018

ABSTRACT

Computer-based solution Economic Dispatches has already been mature, from short term to long term. However, the methods are just for optimization, rather than a tool for the needs of operations. In operation, there are much more conditions and restrictions, such as emergency outages and emission control. We should have a tool to really understand all the things about the grid, and give us the solution directly when we require, instead of getting a stochastic search in both global and local solution space.

This project proposed an imaginary game, in which the machine agent can play with to know about the cost change for every action it takes. After training with different states and actions, it can remember those actions that bring it to the maximum reward, the total cost reduction from a fixed start. In contrast to the current Metaheuristics based or traditional gradient-based solution, it utilizes the memory of the grid system, and it can give us the actions, i.e. the changes in power dispatches, deterministically. We can also use various expected conditions for the machine to learn to optimize the cost in different situations, such as outages and constraints.

This innovative solution has successfully applied to an IEEE test system with six buses and 3 generators, proving that it is usable for such a machine learning concept in solving economic dispatches in different conditions. In the future, it should apply to a much bigger grid network with a higher complexity environment and neural network to check whether it is practical to use.

ACKNOWLEDGEMENTS

I would like to thank Dr. C.W. Yu, my final year project supervisor, suggested the Final Year Project Title "Solving Economic Dispatch Problems using Neural Network" to us. This let me think about how to adopt the modern neural network to a traditional problem.

Besides, I am thankful that Dr. K.W. Chan, the mid-term progress evaluator, pointed out that not all types of neural networks can provide the fully constrained outputs. This made me re-design all the things of the report, and thus brought an innovative solution to the problem.

I would also like to thank Dr. E.A. Jasmin, Dr. T.P. Imthias Ahamed, and Prof. V.P. Jagathy Raj, who are the Indian scholars in Electrical Engineering, that they have been worked for a long term of using the reinforcement learning approaches to the economic dispatches problem. Their formulation of states and actions inspired me to further explore the relationship between reinforcement learning and engineering optimization, which is not a hot topic in the academic field.

Importantly, I would like to thank DeepMind to develop the modern tools of neural network based reinforcement learning, providing a new insight of how computers can solve challenges.

TABLE OF CONTENTS

CHAPTER 1

Introduction

1.1 Economic Dispatch Problem and Optimization Methods

Economic Dispatch Problem has been a subject undergoing intense study in the field of Electrical Engineering. It involves the various issues in cost optimization, network security, and environmental emission control. In a network, there are usually multiple generators connected to buses supplying the total load demand to the consumers. As the relationship between fuel cost and power generation is quadratic rather than linear, different arrangements of generations form different total operational costs. In addition to the equality constraint that $\sum_j Cost(P_j) = P_{demand}$, this arrangement shall also fulfill different criteria such as the power loss limit and emission limit.

In the early time, computers had not yet advanced so that fuel costs were simplified as $Cost(P_j) = a_j + b_j P_j + c_j P_j^2$. This quadratic form gave the advantage of finding the optimal cost fulfilling constraints using Lagrange multiplier. However, it was getting much more difficult when the valve point effects were introduced to the cost function, which made algebraic expressions became a nonlinear system.

Since the mid-20 century, academics have intensively researched on the global non-convex optimization in various higher dimensional application problems [1], [2]. With valve-point effects and other boundary constraints, Economic Dispatch Problem is actually a typical example of non-convex optimization problems. Engineers use the

Computer Sciences tools (Metaheuristics) to solve the Economic Dispatch Problem in different specific natures. These metaheuristics include Simulated Annealing (SA), Genetic Algorithm (GA), Differential Evolution (DE) and Particle Swarm Optimization (PSO). In contrast to gradient-based algorithms, these metaheuristics borrowing the beauty from nature successfully has performed better in solving economic dispatch problems. However, these metaheuristics usually have many hyperparameters to be set, and these parameters are sensitive to the results [3]. If the system for the EDP changes, these metaheuristics may have to reset by trials and errors. stochastic and there is no guarantee of getting the same optimal solution within a same amount of time.

Nowadays, computers are much powerful than before. Computer Scientists formulate a modelling topology for computers to simulate the brain's working, called Artificial Neural Networks (ANN). Artificial Neural networks provide new insights into the various problems. From image classification to the voice recognition, it has already been applied deeply and greatly [4]–[6].

For engineers, we are also concerned about whether neural networks can effectively solve our application problems or not. There are different approaches using neural networks to power engineering problems. For example, the electric load can be forecasted by artificial neural networks [7]. To solve economic dispatch problems, Hopfield Neural Network (HNN) is a quite common focus in the research field due to its fundamental nature as an energy state representation being able to minimize the energy level [8], [9].

Recently, there is a new powerful neural network framework called Deep Q Network proposed for reinforcement learning. It has been proposed by DeepMind in 2013. [10] With convolution Neural Network layers, it has demonstrated the ability of this reinforcement learning for the gaming experience in seven Atari games. It combines an old theory called Q Learning to teach the machine how to get the maximum rewards from the varying environment.

1.2 Innovative Approach using Deep Learning

Among all of the previous researches, there is no a general solution to deal with the various problem for a grid.

For a power company, it is not expected that there are different grid networks to be controlled. The parameters for the transmission lines and generators are constant unless there is a replacement of electrical power equipment such as transmission line, overhead lines, generators, or transmission power transformers.

However, for the same grid, there are various operational conditions taken into considerations. For example, a generation unit is suddenly tripped and raises an emergency outage for repair; the natural gas supply is insufficient so that its reduction of power output has to be compensated by other generation units.

If we can have a specific solution to our grid with designed scenarios for all possible incidents, the engineers can simply change the conditions with a button to re-calculate the optimal dispatches for the changed conditions.

In this project, an imaginary game is constructed as an environment for a specialized agent to obtain the optimal solution in various conditions. The agent will learn how to minimize the costs with the constraints by itself (Unsupervised Reinforcement Learning) via Deep Deterministic Policy Gradient (DDPG), which is an extended implementation of the original Deep Q Network for Reinforcement Learning.

The objectives for this new approach can be summarized as:

0. The well-trained network can give us a good solution much faster than the stochastic searching for repetitive usages.
0. No re-learning is required for the change of designed conditions

This report is organized as follows. In Section 2, there will be literature reviews for the previous methods in solving economic dispatch problem proposed/mentioned

by other academics and the current reinforcement learning development nowadays. In Section 3, the methodology of implementing DDPG-DQN Reinforcement Learning approach to EDP and evaluating the effectiveness will be presented. In Section 4, the results will be presented with tables and graphs to show the effectiveness of this approach. In Section 6, there will be a conclusion with the further development for this imaginary game as an optimization tool to EDP.

CHAPTER 2

Literature Review

2.1 Formulation of the Economic Dispatch Problem

2.1.1 Economic Dispatches

In the US, the cost spent in the fuels was escalating since 1973 at a rate of 25% annually [11]. As the amount of energy produced was so high, any small percentage change in the fuel usage can significantly reduce the operational cost and quantities of fuels. Nowadays, energy consumption is still continuously increasing [12]. The problem of economic dispatches extends to the issues of environmental concerns, network security, and cooperation of other power sources apart from the fossil-fired generation [11], [13]. These additional constraints make the traditional optimization tools cannot easily solve the economic dispatch problem.

Apart from the short-term economic dispatches, long-term economic dispatches should also be addressed.

In a long run, generation units cannot run indefinitely. It may be also necessary to switch on/off the units during the peak hours or the light load period.

In scheduling, there are several concerns about the operation such as grid synchronization and lifespan of generation equipment, affecting the immediate availability of the generation units.

2.1.2 Lagrange Multiplier Method

Traditionally, there are multiple thermal-generating units in the system to support the electrical load P_L. The fuel costs (as inputs to the system) are denoted as F_1, F_2, ..., F_N. The net real power outputs, after deducing the power feedback to the auxiliary, are denoted as P_1, P_2, ..., P_N.

Mathematically, the economic dispatch problem is

$$\begin{aligned} &\min\{F_T\} \\ \text{where } &F_T = F_1 + F_2 + \cdots + F_N \\ \text{and } &g = P_L - (P_1 + P_2 + \cdots + P_N) = 0 \end{aligned} \tag{2.1}$$

Lagrange Multiplier Method can convert the statement above into:

$$\begin{cases} F_T = F_1 + F_2 + \cdots + F_N \\ g = P_L - (P_1 + P_2 + \cdots + P_N) = 0 \\ \mathcal{L} = F_T + \lambda g \\ \nabla\mathcal{L} = \nabla F_T + \lambda \nabla g = 0 \\ \exists\lambda \in \mathbb{R} \end{cases} \tag{2.2}$$

which are equalities instead of parameters' minimization.

The simplest way to solve the problem is to approximate the fuel cost as a quadratic function without considering valve-point effects.

$F_i = a_i^2 P_i + b_i P_i + c_i$ where a_i, b_i, c_i are positive.

$$\mathcal{L} = F_T + \lambda\Big(P_L - \sum_{i=1}^{N} P_i\Big) \tag{2.3}$$

$$\frac{\partial \mathcal{L}}{\partial P_i} = \frac{\partial F_T}{\partial P_i} - \lambda \frac{\partial}{\partial P_i} \sum_{i=1}^{N} P_i \qquad ,\forall i \tag{2.4}$$

$$0 = \frac{dF_i}{dP_i} - \lambda \tag{2.5}$$

$$\frac{dF_i}{dP_i} = \lambda = 2a_i P_i + b_i \tag{2.6}$$

$$P_i = \frac{\lambda - b_i}{2a_i} \tag{2.7}$$

Therefore the solution is now a linear system of equations.

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ \dots \\ P_N \end{bmatrix} = \begin{bmatrix} \frac{\lambda - b_1}{2a_1} \\ \frac{\lambda - b_2}{2a_2} \\ \dots \\ \frac{\lambda - b_n}{2a_1} \\ P_L \end{bmatrix}$$

Lambda-iteration method The Lambda equation can be solved numerically. Since the power generation increase with the increment cost, the power outputs can match the desired load the by adjusting lambda. However, it is not suitable for multiple equality constraints mentioned above.

Multiple constraints This method can extend to multiple equality constraints denoted as g_1, g_2, ..., g_m.

$$\begin{cases} \nabla f = \lambda_1 \nabla g_1 + \lambda_2 \nabla g_2 + \cdots + \lambda_m \nabla g_m \\ g_1 = g_2 = \cdots = g_m = 0 \end{cases}$$

However, inequality constraints such as generation limits cannot be directly included in the equation. Also, the complexity of the computation greatly depends on the constraint functions. As a result, it will be too difficult to solve if all the constraints in the modern power generation are considered.

Inequality constraints Several mathematical techniques can transform the inequalities to equalities for the Lagrange Multiplier Method. In the following, a slack variable s is introduced.
Consider inequality constraint $h(P_1, P_2, ..., P_N) \leq 0$
$g(P_1, P_2, ..., P_N, s) = h + s^2 = 0$ for any $s \in \mathbb{R}$

2.1.3 Common Constraints related to Economic Dispatch Problem

Ramp Rate Ramp Rate is the generator limit in terms of the change in time. UR_i is the maximum rate of increase of power P_i while DR_i is the maximum rate of decrease of power P_i. Their units are usually expressed as MW/h [14], [15].

Spinning Reserve In order to meet the possible power change for the next time interval, the total power output has to be reserved with an amount greater than SR_t [14], [15].

Unit Commitment When it comes to scheduling, we need to consider the running time, start and stop. The arrangement of the running scheduling can directly affect the operational cost which is a part of the total cost being optimized. It is often based on a daily scheduling or a monthly scheduling. [15], [16]

Transmission Loss As there is a voltage drop in each transmission line with a small resistance, it results in the loss in the total power output delivery to the customer side. The power loss for a grid can be expressed by a B-loss Matrix, formulated by George's equation or Kron's formula. It is proven that the total real power loss can be approximated by the generator outputs only [17].

Emission Control Emission Control is to control the environmental emission within a limit or minimized. For example, the emission of the NOx during the operation of fired-fuel generators shall be taken into consideration. The quantitative measurement is to count the amount per interval with a unit of kg/h [18].

2.1.4 Gradient method

Other methods are first-order gradient method and second-order gradient method [11].

The total fuel cost $F_T : \mathbb{R}^N \to \mathbb{R}$ can be approximated by Taylor Series at an operational point $\mathbf{P} = \begin{bmatrix} P_1 & P_2 & P_3 & ... & P_N \end{bmatrix}^T$.

$$
\begin{aligned}
F_T + \Delta F_T &= \sum_{i=1}^{N} \left(F_i(P_i) + \frac{dF_i}{dP_i}\Delta P_i + \frac{1}{2}\frac{d^2F_i}{dP_i^2}(\Delta P_i)^2 + ... \right) \\
\Delta F_T &= \sum_{i=1}^{N} \left(\frac{dF_i}{dP_i}\Delta P_i + \frac{1}{2}\frac{d^2F_i}{dP_i^2}(\Delta P_i)^2 + ... \right) \qquad (2.8) \\
\text{where } &\sum_{i=1}^{N} \Delta P_i = 0 \text{ as the total power unchanged.}
\end{aligned}
$$

This shows the relationship between the change of total cost and the individual change of outputs at each operational point. By choosing suitable changes in the output according to the coefficients, the lowest cost under generator operational limits can be obtained.

2.2 Nature-inspired Metaheuristics

In the previous section, the conventional methods mostly rely on the gradient to find the optimal solution. However, in non-convex optimization problems, it usually do not work.

Convex Region Non-Convex

Figure 2.2: Feasible Region for Convex and Non-Convex Problems [19]

As the diagrams shown, any two points inside the convex region can be linked with a straight line while any two points inside the non-convex region cannot. By using conventional gradient-based tools, the gradient may lead the candidate solution to the in-feasible points and the algorithm cannot provide further movement in such a situation.

Therefore it is necessary to consider other tools to solve EDP. Nature-inspired Metaheuristics have been rapidly developed since 1991 [3]. They imitate how the species live in a complicated world using simple mathematics. As a result, these methods find a "solution" to the problems without mathematical derivatives.

2.2.1 Simulated Annealing (SA)

The basic idea of Simulated Annealing is raised by N. Metropolis in 1953 as a physics theory for the equipment condition of atoms at a certain temperature; after the basic idea was linked with the optimization problems by Pincus in 1970, Krikpatrick suggested Simulated Annealing, an optimization technique to search the optimal solution of general problems [20].

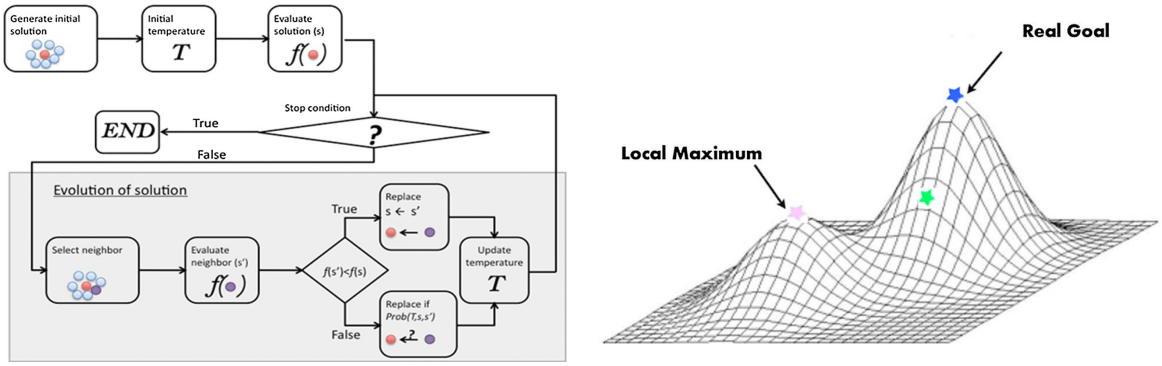


Figure 2.4: Flow Chart and Concept Diagram for simulated annealing (SA) [21]

Simulated Annealing starts at the initial feasible solution, and search the neighbourhood randomly. As the "temperature" is high initially, it will search globally instead of the local optimum. After sufficient iterations, the temperature will be lower and the searching range is reduced. In the end, the temperature will be too low and the solution is obtained.

Instead of the traditional optimization methods, Simulated Annealing is able to search the global or near-global solution with appropriate parameters [22]. There are studies shown that SA can be used for solving economic dispatch problems (e.g. Searching the optimal λ for economic dispatches) [22], [23].

As INGBER noted:

> Simulated annealing presents an optimization technique that can:
>
> (a) process cost functions possessing quite arbitrary degrees of nonlinearities, discontinuities, and stochasticity;
> (b) process quite arbitrary boundary conditions and constraints imposed on these cost functions;
> (c) be implemented quite easily with the degree of coding quite minimal relative to other nonlinear optimization algorithms;
> (d) statistically guarantee finding an optimal solution.
>
> [24, p. 1]

Application to Economic Dispatch Problems -

One of a famous simulated annealing based EDP solution was proposed by K.P. Wong and C.C. Fung [25] in 1993. By dispatching the total demand unit by unit randomly, a feasible solution can be found and the algorithm can be applied. The paper also proposed a technique to transform the B-matrix loss equation to a simple algebraic linear relationship in the dependent generator P_r. Here are the results under different settings of γ:

Table 1: Comparison of dispatch solutions

a	Solutions from Ref. 11		
Gen	Loadings	Fuel cost	
	MW	$/h	λ
1	360.2	1661.956	4.82517
2	406.4	1843.420	4.81883
3	676.8	3137.083	4.82366
Total	1443.4	6642.459	
losses	43.4		
b	Solutions from new algorithm when $\gamma = 0.01$		
Gen	Loadings	Fuel cost	
	MW	$/h	λ
1	359.5459	1659.016	4.82237
2	406.7342	1844.985	4.82583
3	677.1525	3138.656	4.82377
Total	1443.4339	6642.657	
losses	43.4339		
c	Solutions from new algorithm when $\gamma = 0.1$		
Gen	Loadings	Fuel cost	
	MW	$/h	λ
1	376.1226	1733.799	4.88146
2	100.0521	397.030	8.10834
3	986.2728	4508.675	4.85014
Total	1462.448	6639.504	
losses	62.448		

3 generators; Demand = 1400 MW

Table 2: Dispatch solutions with $\gamma = 0.1$

Load demand	Generator loadings			Fuel cost	Losses
	Gen 1	Gen 2	Gen 3		
MW	MW	MW	MW	$/h	MW
500	100.0086	100.0016	306.2836	2379.9622	6.2936
600	100.0597	100.0007	409.9377	2840.6670	9.9983
700	261.5951	100.0081	351.5866	3307.3391	13.1895
800	298.4908	100.3291	418.9266	3772.0664	17.7465
900	330.1606	100.0245	492.8260	4239.6001	23.0118
1000	348.1436	100.7676	580.0187	4715.2363	28.9303
1100	361.1604	100.1610	674.4479	5190.7461	35.7694
1200	381.5067	100.2550	761.6805	5672.4136	43.4424
1300	414.7176	100.6995	836.4216	6160.3027	51.8388
1400	376.1226	100.0521	986.2728	6639.5043	62.4480
1500	368.0314	407.7335	776.1051	7126.6680	51.8714
1600	371.7842	407.9640	881.7268	7613.2847	61.4764
1700	371.1941	407.8110	993.3748	8101.3330	72.3812
1800	457.5262	423.1195	999.9993	8605.0674	80.6463
1900	499.9968	486.9898	999.9942	9190.2998	86.9812

Table 3: Dispatch solutions with $\gamma = 0.01$

Load demand	Generator loadings			Fuel cost	Losses
	Gen 1	Gen 2	Gen 3		
MW	MW	MW	MW	$/h	MW
500	100.0050	100.0007	306.2873	2379.9546	6.2937
600	100.0642	100.0007	409.9324	2840.6648	9.9982
700	282.9852	100.0325	330.3979	3307.7041	13.4150
800	100.0048	397.9365	312.9480	3817.3701	10.8906
900	100.0024	400.8333	414.3950	4281.3169	15.2321
1000	275.1006	399.3372	344.2293	4748.3428	18.6686
1100	309.8751	400.3426	413.5573	5214.6982	23.7763
1200	330.8108	403.5294	495.1281	5686.0342	29.4699
1300	347.6868	405.3712	582.9395	6162.1973	35.9989
1400	359.5459	406.7342	677.1525	6642.6562	43.4339
1500	367.8181	407.5905	776.4725	7126.6680	51.8825
1600	371.7213	408.1905	881.5497	7613.2842	61.4629
1700	371.4536	407.7474	993.1752	8101.3330	72.3777
1800	457.7585	422.9009	999.9979	8605.0693	80.6584
1900	499.9997	486.9839	999.9966	9190.2832	86.9815

Comparing the effects of γ setting

Figure 2.6: Simulation Result in the original paper

As the results shown, SA can be applied in EDP even with the consideration of losses. The authors summarized the pros and cons as follows [25]:

- Advantages
  - both inequality and equality constraints can be matched
  - without gradient computation like Lagrange multipliers
  - no penalty factor is required
- Disadvantages
  - time-consuming computation

Indeed, the computational speed is now much higher than before (billions of transistors per CPU chip today[1]), and parallel computing is rapidly developing in these few years. The effects of cost and power loss due to scaling factor γ should be avoidable using multiple processes with different initial power dispatches.

A recent study has even shown that it can also apply to the dynamic economic dispatch scheduling problem with valve-point effects cost [14].

2.2.2 Genetic Algorithm (GA)

Genetic Algorithm was proposed after the middle of 20^{th} century by many researchers. The main idea in GA is the variation of "chromosome". The integral variables $b_i \in \mathbb{R}$ can be treated as a bit-string by analogy with a chromosome. An objective function is required to evaluate the fitness of the 'individuals' (possible solutions), determining the survival of the individuals. In GA, three features are involved in the process [20], [26]:

- 'Roulette Wheel Selection' - the selection of the next generation from parents are determined by a probability proportional to the fitness values.
- 'Crossover' - to form a new individual by mixing parents' chromosome.
- 'Mutation' - to form a new individual by changing the gene randomly at a certain probability.

[1]https://maria.gorlatova.com/2015/10/speculated-ending-of-moores-law-can-engineering-creativity-sustain-the-rate-of-technology-progress/

Genetic Algorithm allows the searching with nonlinearities and discontinuities [27], but its convergence requires a huge number of generations [26].

Application to Economic Dispatch Problems -

GA was applied in an EDP with valve points effects in 1993. Walters and Sheble encoded the power generations into ten digits per unit [28], and combined both load constraint and cost into the fitness function. The result showed that GA can effectively optimize the result using different binary coding.

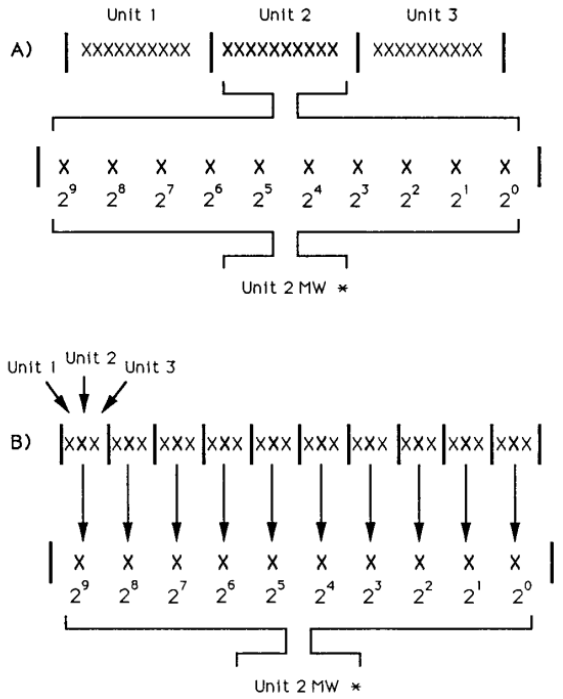


Figure 2.7: Encoding Schemes in an early GA solution to EDP

2.2.3 Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) is a famous and robust method to optimize a problem. By adjusting the parameters, it can perform both global searching and

local searching. It is to simulate how the birds in the sky fly to search the food.

There are N particles randomly distributed in the solution space. They can communicate with each other to alter their direction and speed. They will fly around their own best position (local best) and the best position among all the particles (global best).

$$v_{i,d}^{(t+1)} = w * v_i^{(t)} + c_1 * rand() * x_{pbest,d} - x_{i,d}^{(t)} + c_2 * rand() * x_{gbest,d} - x_{i,d}^{(t)}$$
$$v_{i,d}^{(t+1)} = x_{i,d}^{(t)} + v_{i,d}^{(t)}$$
$$v_{i,d}^{(t+1)} = w * v_i^{(t)} + c_1 * rand() * x_{pbest,d} - x_{i,d}^{(t)} + c_2 * rand() * x_{gbest,d} - x_{i,d}^{(t)}$$

where d is the feature dimension and i represent the particle i

With the control of parameters, it can enhance the ability of global searching or local searching, but it will take a longer time to converge.

Application to Economic Dispatch Problems -

As mentioned before, it is an algorithm designed for minimization/maximization. Therefore, there is an intensive use of PSO in economic dispatch related problems.

Zwe-Lee Gaing applied PSO to a 40-unit system with the comparison with GA. PSO was searching the global space initially leading to a lower convergence compared with GA. However, it did beat the better fitness value in the end under the local searching stage. Besides, the standard derivation diagram also shows the key difference between GA and PSO [29].

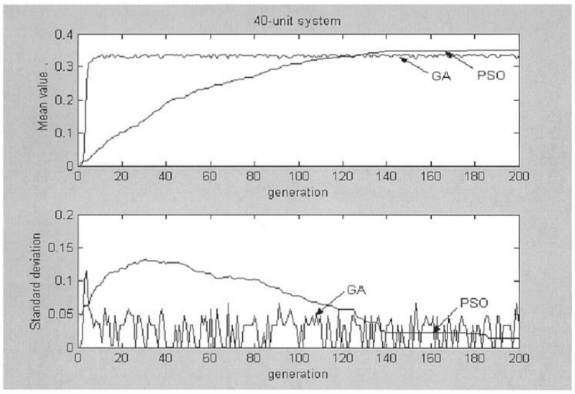


Figure 2.8: Differences in the searching behavior between PSO and GA [29]

2.3 Artificial Neural Networks (ANNs)

Apart from Metaheuristics, Artificial Neural Networks (ANNs) are also inspired by nature [20].

Neurons are the basic elements in our central nervous system. 5 billion neurons consisting of the central nervous system are responsible for receiving, integrating, and transmitting the signals in a human being [30].

McCulloch and Pitts modelled 'neurons' in the 1940s based on biological neurons: a synthetic neuron accepts action potentials and gets activated once their sum is higher than the threshold; the activation makes a response from the neuron transmitting to the next level as an action potential [30]. It is then defined with five components:

- inputs' nature
- outputs' nature
- an input function
- an output function

- a threshold activation function (state)

There are six types of challenging problems can be solved via neural networks [31].

- Pattern Recognition
- Data Clustering
- Approximation of Functions
- Forecasting and Predication
- Optimization
- Content-addressable Memory
- Control of Dynamic System

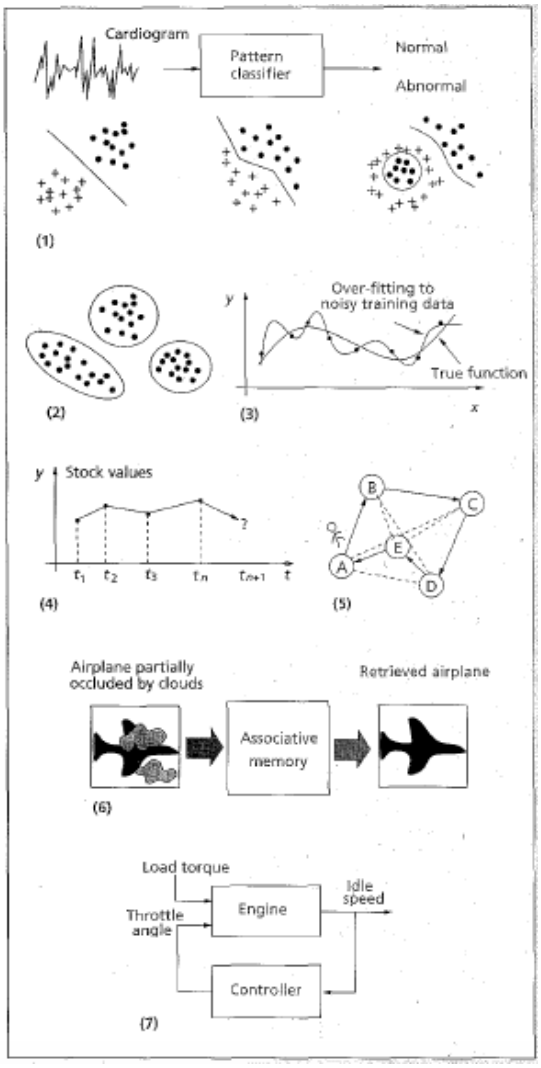


Figure 2.9: Application of Neural Networks [31]

By using neural networks, the problems mentioned above can be solved with distributed parallel self-learning system and give robust solutions without complicated problem-specific pre-coding [31].

Different types of neural networks are developed based on different nature of

applications.

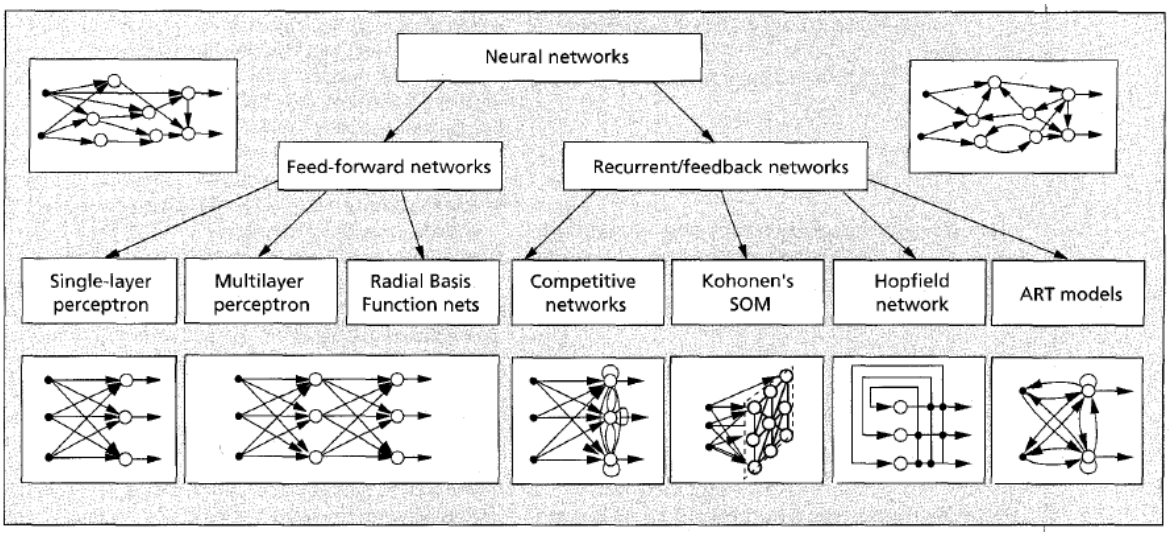


Figure 2.10: Types of Neural Networks [31]

In the following content, Hopfield Neural Network and Radial Basis Function Neural Network are being discussed, representing the two major types of neural networks, namely Recurrent Neural Network and Feedforward Neural Network.

2.3.1 Hopfield Neural Network

Hopfield Neural Network, the simplest neural network was proposed by John Hopfield in 1982 [30], is an analogy to the nervous system searching a stable state by using the neighbour states. It can also correct the errors. It is also equivalent to the energy minimization in statistical physics.

Hopfield Neural Network contains no hidden neurons but fully connected with other neurons; it is an associative memory system with unsupervised learning and good at optimization and pattern recognition [32].

In Hopfield Neural Network, there are three elements [33]:

- activation values $\mathbf{X} \in \mathbb{R}^n$ which are the stored neurons as memory
- weights $\mathbf{W} \in \mathbb{R}^n\mathbb{R}^n$ representing the connection between neurons
- thresholds for each unit $\mathbf{T} \in \mathbb{R}^n$ determining the sensitivity to the noisy data for correction

By computing $\mathbf{X} = sign(\mathbf{XW} - \mathbf{T})$, the data will try to get closer to the stored state. After repetitions, the result will converge to the ḿemory.

It is proven that Hopfield Neural Network is robust to obtain a stable state from the dynamic inputs, being able to memory words or shapes and thus recognize the similar images [30], [34].

Application to Economic Dispatch Problems -

Silva, Nepomuceno, and Bastos implemented a modified Hopfield network for the economic dispatch optimization with the nonlinear generation cost. In their approach, it minimization target also included the constraints to achieve feasible candidate solutions; their work showed that Hopfield Network can be applied to EDP with various constraints in a fast convergence speed [35], [36].

2.3.2 Radial Basis Function (RBF) Neural Network

Radial Basis Function Neural Network (RBFNN) is "a special class of multi-layer feed-forward networks" with two layers [31].

RBF Network contains one layer of hidden neurons for nonlinear transformation;

it is to perform function mapping and good at modelling for dynamic systems and time series [32].
Gaussian functions (radial basis functions) are used as the kernels to transformer the input data to non-linear outputs. The effect of the neurons in the hidden layers is determined by the radial distance from the 'centres', the weighting of the neurons, and the variance of the RBF.

Consider there is a function mapping the inputs to a output perfectly $f : \mathbb{R}^n \to \mathbb{R}$

And the function can be approximated by $s : \mathbb{R}^n \to \mathbb{R}$.

$$s(\mathbf{x}) = \sum_{i=1}^{m} w_i \phi(\|\mathbf{x} - \mathbf{C_i}\|) \tag{2.9}$$

As the training data $\{\mathbf{x_1}, y_1\}$, $\{\mathbf{x_2}, y_2\}$, ..., and $\{\mathbf{x_n}, y_n\}$ are given where $\{\mathbf{x_i}, y_i\} \in \mathbb{R}^d\mathbb{R}$, centres can be determined by K means clustering [37] by specifying the number of centres $m < n$.

The number of equalities is the same as the number of unknown weights.

$$s(\mathbf{x_i}) = f(\mathbf{x_i}) = y_i \tag{2.10}$$

The weights can be found by solving the linear system above.

However, this only gives the exact matching to the training data. To 'enable' the prediction ability, the number of centres and the variances of the Gaussian functions should be adjusted [37].

Application to Economic Dispatch Problems -

For economic dispatch problems, RBF is seldom used for its optimization, due to the fact that RBF is a nonlinear function approximation instead of a minimization tool.

0. approximation of λ: As λ is the required parameter in cost minimization, RBF was applied in approximating λ to solve EDP [38].
0. approximation of

Jasmin, Ahamed, and Jagathiraj were a few to implement RBF with economic dispatch problems. However, the use of RBF is just part of the solution - as an approximator for Q values in their reinforcement Q-learning approach.

Therefore, neural networks are a powerful tool, but they have different usages. Some of them cannot be directly applied in EDPs.

2.4 Reinforcement Learning and its current development

Reinforcement Learning seems to be a field that is completely not related to the economic dispatch problems, but this will be the main focus in the following sections.

The framework of reinforcement learning was first introduced in the early 1990s [39], [40]

The Markov Decision Processes is the fundamental framework for reinforcement learning problems. During the transition of states, there is a reward generated in each epoch and the action and state

As mentioned before, Jasmin, Ahamed, and Jagathiraj had applied reinforcement learning with RBF approximator into the economic dispatch problems. They are almost the only group of people having particular interests in the reinforcement learning approach for the power engineering optimization issues. “

0. A function approximation approach to Reinforcement Learning for solving unit commitment problem with Photo voltaic sources (2016)

0. Reinforcement learning in power system scheduling and control: A unified perspective (2011)
0. Reinforcement learning approaches to economic dispatch problem (2011)
0. Reinforcement learning solution to economic dispatch using pursuit algorithm (2011)
0. A Reinforcement Learning approach to Economic Dispatch using Neural Networks (2008)
0. A Reinforcement Learning algorithm to Economic Dispatch considering transmission losses (2008)
0. Reinforcement learning approaches to power system scheduling (2008)

" [41, p. 1]

In many papers of them, the power generations are expressed as states expressed by (k, D_k). The state means that there are D_k remaining powers to be dispatched and generators $1...k$ have been dispatched. [42] For example, if there are three generators in the problem, there are three state transitions in a single epoch as follows:

$$(0, 420) \rightarrow (1, 290) \rightarrow (2, 155) \rightarrow (3, 0)$$

The example epoch shows how the agent moves from a start point to an end point. D_k decrease along the increase of k, and finally $(3, 0)$ is reached. These state transitions can be equivalent to the power dispatches as follows:

$$\begin{aligned} &\text{Unit 1: } P_1 = 420 - 290 = 130MW \\ &\text{Unit 2: } P_2 = 290 - 155 = 135MW \\ &\text{Unit 3: } P_3 = 155 - 0 = 155MW \end{aligned} \tag{2.11}$$

Therefore, Q-learning method was applied to find the state transitions with optimal total cost.

2.4.1 Q-learning

Q-learning is a traditional algorithm for reinforcement learning. Till now, it is still a popular concept for most reinforcement learning problems.

Dr. Jacob Schrum uses the following Grid World problem as an example of Q-learning theory. [43]

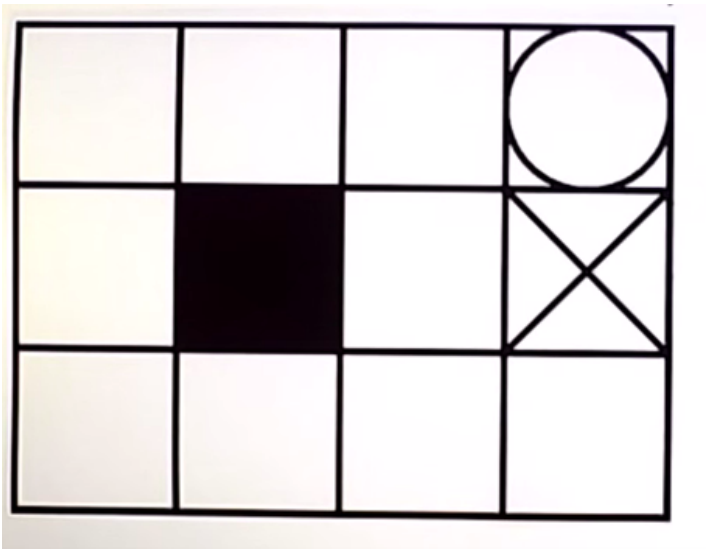


Figure 2.11: Grid World Problem

In the above figure, "Circle" is the positive goal. "Cross" is the negative terminal. The agent may start at any blank grid and take actions one by one to the terminals.

If our agent (player) can be in any of these 9 empty grids, only the right positions near the terminals can immediately get the reward of 1 or -1. However, by playing this game repeatedly, we can know that whether the next grid is the one we win or lose before. By backward thinking, we can start from the positions near the terminals to the far away positions, such that we know which direction should be preferred for a better future reward.

Q value The following equation is the mathematical relationship used in the Q-learning theory:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a_{t+1})$$

where s_t is the current state, a_t is the action we take in s_t,

r is the reward for this action and state, and γ is a parameter $\in (0, 1)$

The term Q value is the representation of expected future reward in state s under the action a. The term γ is also called as a discount factor. It is equivalent to the agent "seeing" how far behind the current state. The typical values are 0.9 and 0.99.

	$\gamma = 0.9$	$\gamma = 0.99$
10 steps	0.349	0.904
20 steps	0.122	0.818
30 steps	0.042	0.740
40 steps	0.015	0.669

From the mathematical proofs, $\gamma = 1.0$ is prohibited due to the infinite sum of future rewards. The future rewards must vanish within a finite number of steps.

Temporal Difference (TD) error During the learning, there is always a mismatch between $Q_t + r$ and Q_{t+1}. This is called TD error. By experiencing sufficient various states and actions, the Q values will be all corrected and finally vanish.

In the mentioned world grid problem, the agent in the bottom left concern (state) should pick the "UP" direction (action) in order to get a higher expected future reward (Q value). By iterations, the Q values are shown as follows

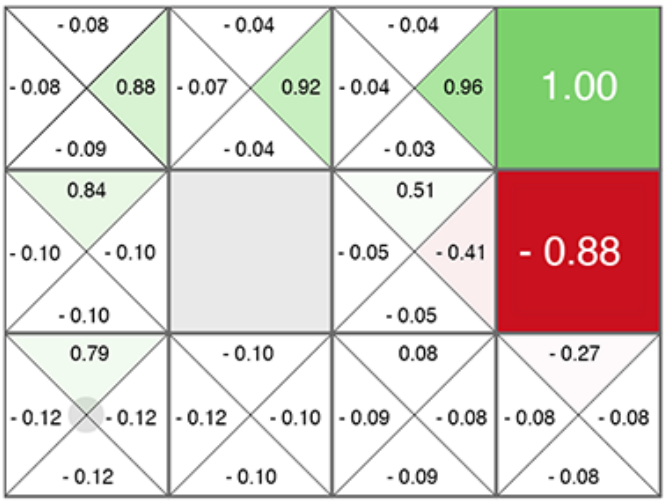


Figure 2.12: Grid World Problem with Q values [43]

Therefore, after the repeatedly playing in random states (reinforcement learning), there will be a set of Q values associated with all actions in any state. By looking-up this Q values table, we can choose the action deterministically and the positive goal is expected to be reached in the end.

2.4.2 Deep Q Network (DQN)

DeepMind, a company focusing on the artificial intelligence in England, published an academic paper in Nature in 2015 illustrating a new reinforcement learning method with an astounding result presented in a real Atari game [10].

This breakthrough in machine learning raise the public attention and has become a key element in the field of machine learning.

In the original Q-learning theory, Q values are described as the data entry in a record table. DeepMind considered the Q values as an output from a neural network, where the input was the state and outputs contained all the Q values for each action. For example, in the previous grid world problem, there are always four neurons as the Q

values for "UP", "DOWN", "LEFT", "RIGHT". As the neural network can approximate any function with sufficient layers and neurons, the Q values thus can be "stored" in the network instead of an impractical large Q table.

There are two major DQN architectures shown below. The left one is the straight-forward approach to implement the $Q(s, a)$ function, while the right one is the advanced architecture proposed by DeepMind for DQN, giving all the actions with one forward feeding of s.

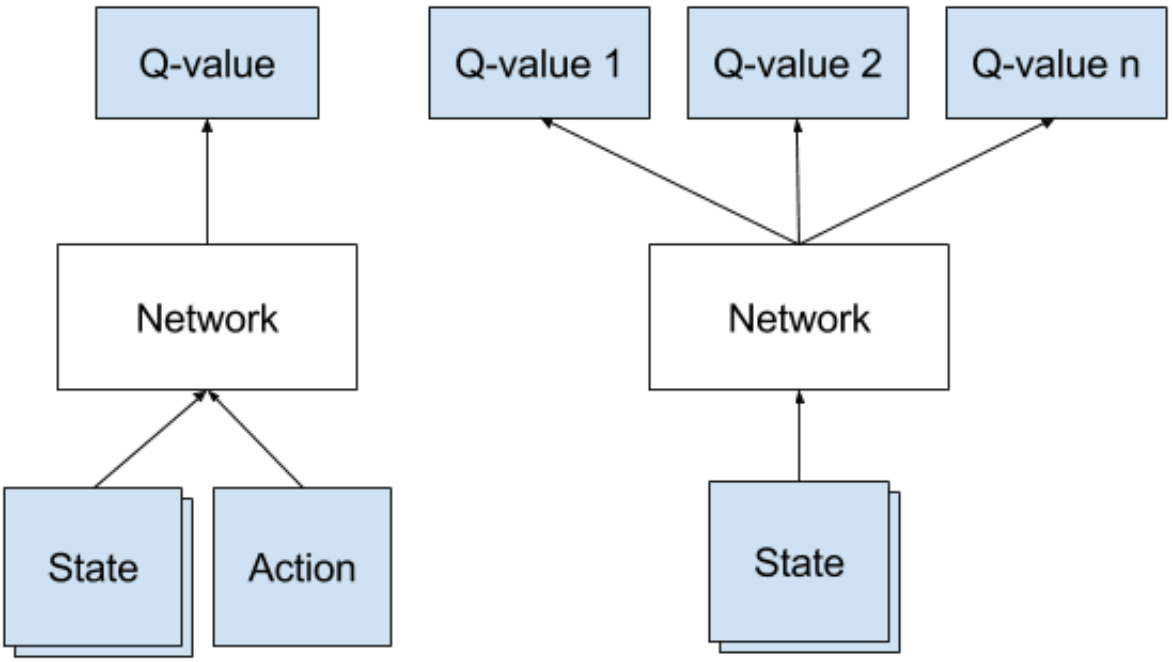


Figure 2.13: Two major DQN architectures [44]

Visualization of DeepMind's DQN by real gaming experiences: [10]

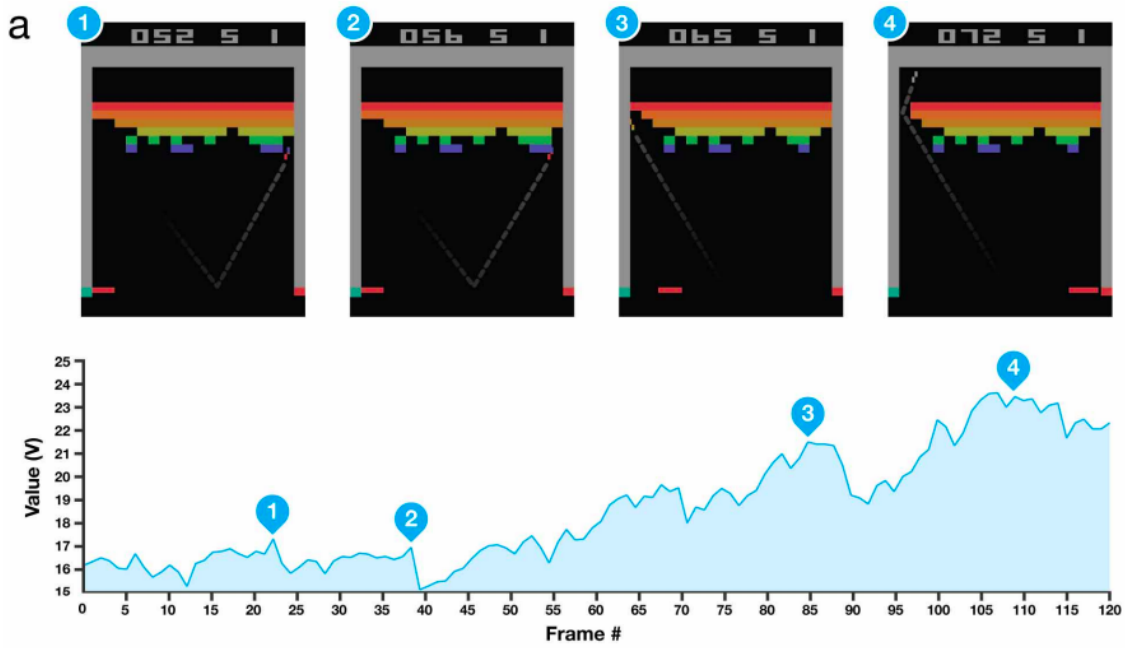


Figure 2.14: The demonstration of how the agent chooses an action in various states

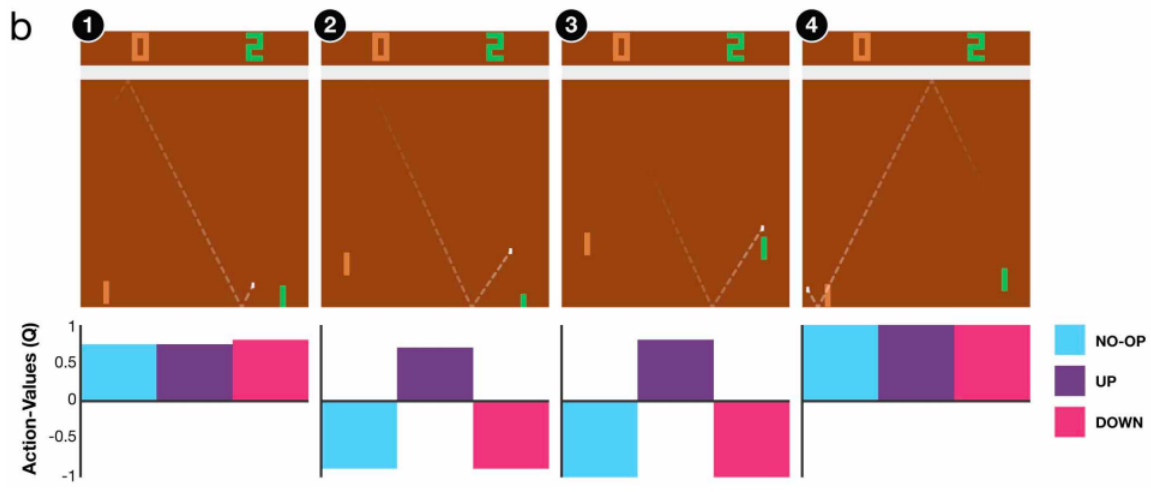


Figure 2.15: The demonstration of how the agent chooses an action in various states

In DQN, the Q values are stored implicitly in the network. As a neural network

is free to feed any input values and get the outputs, it can be very easy to feed the state information and obtain the Q values for each action. As a result, the TD error expressed as

$$\frac{1}{m}\sum_{i=1}^{m}(Q(s_t, a_t) + r_t - \max_a Q(s_{t+1}, a)$$

In the DeepMind's approach, only two forward feedings with s_t and s_{t+1} to the network can give $Q(a|s)$ values for all actions. The backward gradient update is done by the minimization of TD_{loss}, while the action taken is determined by the action with the highest Q value.

The function $Q(s, a)$ was mentioned long before, and people try to use different ways to approximate this function but in vain. The gaming results were the direct evidence that neural network can approximate such an abstract function, and also Q-learning had been no longer a mere theory but become a practical solution for reinforcement training in the huge state spaces.

Neural Network was thought to be infeasible to the updating environment (online learning), and the success of DQN is brought by its implementation of experience reply. When the data are uniformly randomly drawn from the memory, the correlation between data samples are much lower such that the training can be stabilized [45].

After the publication of the paper, many academics step into this new research field. Eventually, three contributions are strongly recognized by DeepMind.

Prioritized Experience Replay To choose samples from a memory database to learn, it is common to use the uniform random distribution which is simple and not time-consuming. In the authors' view, the memory are not equally important; those with significant Temporal Difference (TD) errors should have a higher priority to be sampled in order to explore the unlearned states and actions [46].

Dueling Dueling is a DQN-specific technique specific. From the game design of car playing mechanism, the authors have noticed that sometimes the choice of actions have no direct effect on the expected future rewards. For example, the car in the middle of a straight road can freely choose to move left or right without the concerns of a car crash. Therefore the output of Q(s,a) should be formulated as $Q(s,a) = V(s) + A^*(s,a) = V(s) + A(s,a) - \bar{A}(s)$ [47].

Double Deep Q Network (DDQN) DDQN is a powerful improvement for the DQN architecture. The original DQN uses its own $Q(s',t')$ for the gradient update; it leads to a instability to the network as both Q values are changing (like a snack chasing its own tail). By using a clone of the DQN, the target Q can be stabilized with a fixed-step update of the target DQN [48].

Insights from the DeepMind's DQN In an overview, the author Li summarized 3 important features presented in the DQN for such an astounding result [49]:

0. Action value stabilization can be done by deep nets and experience replay
0. Well-designed features and states which are the minimal knowledge for the agent to learn the relationship
0. A flexible network architecture with the ability to perform various task

2.4.3 Policy Gradients

Back to the year of 2000, the policy gradients had been already mentioned by Sutton et al. [50]. In the paper, it is completely full of mathematical expressions without a single function or a case study. However, since 2000, "policy" can be expressed as an abstract mathematical function π.

p^π is the averaged long-term reward under the policy π; r_t is the reward in step k; d^π is the distribution of states that being stationary; R is the expected reward for

the given current state and taken action.

$$p^{\pi} = \lim_{n\to\infty} \frac{1}{n} E[r_1 + r_2 + \dots r_n|\pi] = \sum_s d^{\pi}(s) \sum_a \pi(s, a; \theta) R(s, a)$$

In the above expression, it is impossible to find the exact form of these parametric functions.

However, its gradient w.r.t. the change of θ depends on the $\pi(s, a; \theta)$ only. [2]

$$\nabla_\theta p^{\pi} = \sum_s d^{\pi}(s) \sum_a \nabla_\theta \pi(s, a; \theta) R(s, a)$$

$$\nabla_\theta p^{\pi} = \sum_s d^{\pi}(s) \sum_a \frac{\pi(s, a; \theta)}{\pi(s, a; \theta)} \nabla_\theta \pi(s, a; \theta) R(s, a)$$

$$\nabla_\theta p^{\pi} = \sum_s d^{\pi}(s) \sum_a \pi(s, a; \theta) \frac{\nabla_\theta \pi(s, a; \theta)}{\pi(s, a; \theta)} R(s, a)$$

$$\nabla_\theta p^{\pi} = \sum_s d^{\pi}(s) \sum_a \pi(s, a; \theta) \nabla_\theta \ln \pi(s, a; \theta) R(s, a)$$

Note that $d^{\pi}(s)$ and $\pi(s, a; \theta)$ are the probability density functions for choosing the state and the action. The following expected value is computable using a mini-batch of samples.

$$\nabla_\theta p^{\pi} = E[\nabla_\theta \ln \pi(s, a; \theta) R(s, a)]$$

Therefore, with a batch of samples, the parameter θ can be updated by

$$\theta_{t+1} = \theta_t + \alpha R_(t + 1) \nabla_\theta \ln \pi(a_t|s_t; \theta)$$

[2] d^{π} is a conditional probability density function with the given s_0 and π

In today machine learning libraries (e.g. TensorFlow, PyTorch), an update of the parameters only requires the gradient change instead of the explicit cost function. Therefore, it can easily get into the current machine learning implementations.

2.4.4 "Actor" and "Critic"

The terms and concepts of "Actor" and "Critic" had been proposed by Konda and Tsitsiklis for almost twenty years [51]. These terms can easily visualize the relationships between valued based learning Q values and policy-based learning policy gradients. Suppose there is a dance performance, the dancer (actor) has to choose the dances to perform in order to get the highest score, but he does not know what the score he can get. While the judges (critic) do not know how to perform a dance, but they know the way to valuate every dance.

Actor and Critic are looking into the same thing in different aspects, but they use different measures to adjust their behaviors (the parameters). Critic is simply a Q-value based learning agent, it is to evaluate and minimize the TD errors during the training. In a long-term, when the TD error is low, Critic can give the accurate scores (Q values) for any given state and action. In contrast, Actor is a policy gradient-based learning agent, it performs the parameter update based on the information provided by Critic (the scoring).

However, in the early 2000s, there were no good ways to present the results. It was just a mathematical paper with the solid mathematical proofs.

After the DQN was applied successfully in reinforcement learning, Actor-Critic algorithm was one of the direction in its development.

One famous algorithm based on Actor-Critic is the Asynchronous Advantage Actor-Critic (A3C) implementation proposed in June 2016 [45].

Asynchronous Advantage Actor-Critic (A3C) DeepMind found that the original Actor-Critic approach contains a flaw making it impractical for the reinforcement learning: Both Actor and Critic are updated using the same set of data, bring the difficulty in the convergence of Actor and Critic's parameters.

A3C is designed to have multiple agents playing different epoch-es at the same time. It increases the variety of the data samples to stabilize the convergence [52].

The term "advantage" refers to the same meaning in DQN's dueling. As the function used in Actor's policy gradient is logarithm, it is necessary to guarantee that the $Q_{max}(s)$ is positive.

By decomposing $Q(s,a)$ into $A(s,a) + V(s)$, advantage $A(s,a)$ has the same gradient w.r.t. θ^{π} as $Q(s,a)$. There must be positive and negative advantage values $A(s,a)$ among all actions, unless $A(s,a)$=constant for all values of a.

The main difference in A3C is that the gradient descent optimization step shares the same gradient change to a common network (global net). The global net is thus updated by multiple agents. It decreases the data correlation and at the same time enhance the convergence stability.

CHAPTER 3

Methodologies

When it comes to a solution that can be flexible to adapt the various conditions(requirements), no traditional tools can serve for such a problem.

As mentioned before, Neural Network reinforcement learning can be a practical choice nowadays. DQN had shown the ability to evaluate the playing strategy for maximizing the game scores in different game mechanisms, while AlphaGo had shown the potential super problem-solving ability by defeating the top players in Go. It is expected that such a machine learning method can provide us an innovative way for the problems we are facing in engineering fields.

In this project, EDP is to be converted into a machine learning problem. As all these successes were presented by game playing, the EDP will be translated into a game mechanism for the implementation of neural network reinforcement learning.

As a start of implementation, we merely consider two changeable conditions - the variety of power outputs/demands and the effect of transmission loss.

As the solution is grid-specific, the settings of the grid are considered as constants. In case there is a need for adjustment of settings, the existing models can be re-learned to update the model parameters to fit the revised settings. This is a

side-benefit of using the reinforcement approach towards an engineering problem.

3.1 Game Mechanism

3.1.1 Parameters - Settings and Conditions

Base on the mentioned considerations, the parameters of the whole grid can be divided into two groups - setting and conditions. Settings are assumed constant for any scenario, while conditions are adjustable by the users.

Table 3.1: My caption

Parameters	
Settings	Conditions
Cost Coefficients	Power Total P_T
Generation Limits	Power Loss P_T
Transmission Loss Matrix	Power Demand P_d
	Mode Selection _ms

For the settings, they are fully implemented in the gaming environment. For example, in Tetris, the shape and position of puzzles are fully controlled by the environment, so that the agent can just passively observe the states.

For the conditions, they are known by the agents. For example, in Tetris, the puzzles shown on the screen are the observable states for the agent, and the agent can take actions according to this known information.

3.1.2 Design of game rewards

For all the machine reinforcement learning methods, a reward is a required item for the agent to learn "good" or "bad". As mentioned in the previous section, the policy

based learning also required the measurable reward to optimize its policy. There are usually two types of rewards presented in a game.

Sequential Rewards for every step In each stage, there is a reward obtained for any action chosen. A typical example is Temple Run. The rewards are obtained and accumulated during an epoch. The accumulated reward is the final score displayed in the game.

Terminal Reward at the end During the playing, it is hard to evaluate whether the playing is good or not. The result can be only known at the end of epochs. Solitaire is an example which is relatively easy to play, while Go is an example which is relatively difficult to play.

In the training, Sequential Rewards are much better for the machine to learn, as it can evaluate the "goodness" of actions taken in each step, while the Terminal Reward can be just obtained once for hundreds of steps.

In this imaginary game, the sequential rewards can be used for the cost optimization.

In the game playing, it is expected that there will be a sequence of transitions of states:

$$s_1 \to s_2 \to s_3 \to .... \to s_n$$

s_1 is the initial state (game start) and s_n is the final state (game end). As the powers presented each state can be treated as the candidate solutions for our cost optimization objective, it is expected that $Cost(s_{t+1}) < Cost(s_t)$. For Q-learning, the agent always want to maximize

$$Q(s_m) = \lim_{n \to \text{inf}} r(s_m, a_m) + \gamma r(s_{m+1}, a_{m+1}) + \cdots + \gamma^{n-m} r(s_n, a_n)$$

Therefore, the $r(s_t, a_t)$ can be expressed as $Cost(s_t) - Cost(s_{t+1})$ s.t.

$$\begin{aligned} Q(s_m) = \lim_{n \to \text{inf}} & (Cost(s_m) - Cost(s_{m+1})) + \\ & \gamma(Cost(s_{m+1}) - Cost(s_{m+2})) + \cdots + \\ & \gamma^{n-m-1}(Cost(s_{n-2}) - Cost(s_{n-1})) + \\ & \gamma^{n-m}(Cost(s_{n-1}) - Cost(s_n)) \end{aligned} \tag{3.1}$$

If there is a action can generate the highest cost reduction in the future, the agent will choose that action and proceed the next action.

For the optimal state, the future expected reward for any action must be negative regardless of the value of γ. In other words, the optimal state will be stationary to keep the reward as zero. And the nearby states will always move towards it.

If we specify γ is sufficiently large and stationary states are treated as the end of the game, the Q value can be approximated as

i.e. $Q(s_m) \approx Cost(s_m) - Cost(s_n) + 0$

Along the path towards to the optimal solution, $Cost(s_n)$ is constant and $Cost(s_m)$ is reducing, thus the Q value is a positive number reducing to 0.

As $Cost(s_m)$ is constant for the state s_m regardless of the choice of action, the $\max Q(s_m)$ is equivalent to $\min Q(s_n)$.

3.1.3 Machine Learning Framework for states and rewards

In the reward design, it is expected that the number of steps should be minimized in order to minimize the effect of discount factor γ. Besides, the conditions (i.e. Power Total P_T, Power Demand P_d, Mode Selection _ms) contains continuous values. If these

variables are quantized, the machine may fail to learn the fine-tuning of the dispatches, that is the most important part of our EDP optimization.

Therefore, the machine learning framework should accept continuous states as inputs and output continuous actions for fine tuning the dispatches.

In the researches of Jasmin et al., they usually discretize the power demands in the states. For example, 5 MW and 10 MW are chosen as the discretization step in one of their papers [42]. However, the original Q-learning has been re-introduced with policy gradient to select the maximum Q, that made a great evolution to the Deep Q Network - Deep Deterministic Policy Gradient.

Deep Deterministic Policy Gradient Deep Deterministic Policy Gradient, or DDPG, is a variation of DQN. It was proposed by DeepMind in 2016. It is a new framework based on the previous work in DQN, policy gradient, and Actor-Critic. It can perfectly fit for the use of continuous states and actions.

Therefore, this DQN extended version is chosen among all the other frameworks. This framework will be discussed in the later part of this report for a much complete picture of the network design.

In the DDPG paper, the games were mostly simulating the physical environment in a real world. The time-related quantity such as acceleration and velocity are usually presented in the states.

As a result, apart from the power dispatches (locations), the rate of change of power (velocity) should be also presented in the states.

By introducing the rate of change of power, it is expected that the agent can gain something from the previous step. For example, the previous step is to increase the amount for achieving the cost reduction, the increase should be kept instead of randomly pick a direction to change the power dispatches.

As the power output for the last generator can be fully determined by the other generators[1], the information for the last generation unit can be fully omitted. As a result, there is only a $(n_{gen} - 1)$ continuous actions output from the DDPG's Actor network.

In order to have the stable direction for the power change, there are some limits to the actions.

0. if the direction for the power change in unit j is positive (negative), the maximum negative (positive) control $action_j$ can only stop the change.
0. if $action_j$ is zero, the current velocity should be kept.
0. As the absolute change of power in each step is limited by V_m, the adjustable range of velocity for $v_j = 0$ shall be also V_m

Therefore the relationship between actions and rate of change can be linked as follow:

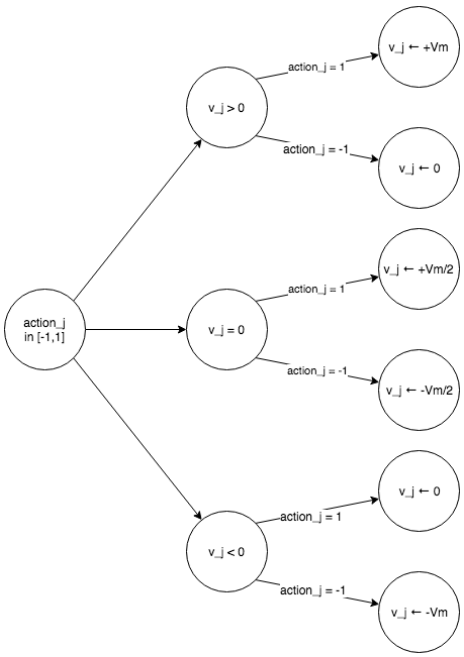


Figure 3.1: Designed behavior for the rate of change (except $v_j = 0$)

[1]A previous research has shown how to determine the power dispatch of dependent generator G_r with transmission loss [25]

As there are always three points $(-1, v_{min})$, $(0., v_{same}), (1, v_{max})$ for all actions and states, the intermediate value can be found by a quadratic/linear polynomial using curve fitting.

For the states, the mode of the environment is represented in bits in $-1, 1$. For example, $[1, 1]$ means both P_t and P_d are frozen (lossless), $[1, 0]$ means the P_t is frozen, $[0, 1]$ means the P_d is frozen.

As mentioned by Li [49], the states shall provide the minimal information which is sufficient for the machine to learn, therefore the mode is represented by bits. Besides, the power demand P_d is not presented in the state as it is just the P_T-P_L, and it is a duplicated value of P_T when the training is set as lossless.

Therefore, for 3 generators case, the states and actions are as follow:

$$\begin{aligned} \text{states} &= (P_1, v_2, P_2, v_2, P_T, P_L, \_ms1, \_ms2) \\ \text{actions} &= (a_1, a_2) \end{aligned} \tag{3.2}$$

By using DDPG neural networks, the states and actions are fed into many layers which adds the nonlinearity to evaluate the Q values.

3.1.4 Environment Response to actions

The pseudo code for the environment to response the action values determined by the Actor network is shown as below:

Algorithm 1: Updating the power dispatches and the state of environment.

function execute (*actions*);

Input : Action values a_1, a_2, ...a_{n-1}

Output: new_states, reward, terminal

for *j = 1 ... n-1* **do**

  $a_j \leftarrow$a$_j$.clip$(-1, 1)$;

  $v_j \leftarrow$ **adjust_power_rate**(a$_j$);

  $P_{t,j} \leftarrow$P$_{t,j} + v_j$;

end

$P_{t,n} \leftarrow$ **calculate_dependent_power**(P$_t, n, P_d$);

if ***is_new_power_feasible***(P_t) **then**

  reward $\leftarrow$ *cost*(P)$-$*cost*(P$_t$);

  P $\leftarrow$P$_t$;

else

  reward $\leftarrow 0$

end

$s_t \leftarrow$ **get_state**()

if $s_t = s$ **then**

  terminal=True;

else

  terminal=False;

end

3.2 Construction of Neural Networks

As mentioned before, Deep Deterministic Policy Gradient (DDPG) is the framework used in this machine learning problem.

There are two online neural networks ($ActorEval$, $CriticEval$) and two offline neural networks ($ActorTarget$, $CriticTarget$) required in DDPG framework.

Similar to the DDQN technique used in DQN, there are online and offline models having the same architecture. $ActorTarget$ and $CriticTarget$ are the references for calculating their backward gradients. A soft update is used for these two target networks to update their parameters.

There are five value-added improvements applied to the original DDPG framework.

3.2.1 Feature 1 - Input State Normalization

For neural networks, it can be classified as CNN-type and MLP-type neural networks [53], [54].

For a CNN-type neural network, it is to deal with the pixel using several convolution layers to extract the features represented by the image. For an MLP-type neural network, the feature vectors are not in the same units and scales, such that the parameters may have to well-defined [55].

In DDPG, as there are so many data stored before the first training, we can evaluate the features of the original data, and construct a data transformation to normalize the state input. For 1-D data, it is common to adopt the $t = \frac{x-\mu}{\sigma}$ to normalize the data.

for an m-dimensional data, we can also apply a similar concept to it. After the

transformation, the mean and covariance of the input state become 0_m and I_{mxm}. The mean is simply the average vector of all sample vector.

For covariance, the transformation is called Whitening transformation. Let X is an mxn matrix where m is the feature dimensions and n is the number of samples. Suppose there is a matrix M_{mxm} such that

$$E[MX(MX)^T] = I$$

$$E[MXX^TM^T] = I$$

$$E[M\Sigma M^T] = I$$

By choosing $M = \Sigma^{-1/2}$ which is a symmetrical matrix, $E[M\Sigma M^T] = E[M\Sigma M] = I$

Therefore, the sample bias vector μ_m and whitening transformation matrix M_{mxm} can be obtained. By comparison to the layers in neural networks, it can be treated as a fully-connected layer with fixed weight W and bias b.

MLP equivalent layer - "Whitening Layer" In MLP problems, fully-connected (FC) layers are the most common elements in MLP neural networks. The sample bias vector μ_m and whitening transformation matrix M_{mxm} actually do the same thing as the FC layers. However, the parameters μ and M are not updated by the gradient. The implementation of the layer-based whitening transformer in PyTorch is attached in Appendix A.

Similar to the DQN target net if periodic updates present For DDPG after training, the states are no longer randomly drawn. If required, the "whitening layer" can be updated according to the new data in the memory. It is very similar to the situation that the target net's parameters are locked during the gradient descent while getting updates at a fixed interval. For DQN target net, the action outputs are changing due to the parameter updated. For "Whitening Layer", the state input may be changing due to the action performed.

3.2.2 Feature 2 - LSUV Weight Initialization

After Mishkin and Matas comprehensively analyzing different kinds of networks and activation functions [56], From [56], [57], the authors mentioned that the weight initialization is important for a deep network. As long as there are sufficient amount of data in the memory, we can do the proposed LUSV Initialization for each layer. This initialization is to normalize the weights using the input data. From the input side to the output side, the author shows that it is practical and worth to do this initialization.

3.2.3 Feature 3 - Batch Normalization

It is not sufficient to have a good initialization for the sample and layers. Batch normalization is a new technology that regularizes the neurons' outputs during the training. There is already intense use of it in order to avoid the gradient vanishing problem in a deep neural network. It was also mentioned by the DDPG's paper and the result can be much improved by using Batch Normalization Layer.

3.2.4 Feature 4 - Replacement Strategy for Experience Replay

In the DDPG's paper, the authors gave up the implementation of Prioritized Experience Replay. Instead, the replacement strategy is just to replace the oldest data.

Prioritized replay has so many parameters to be set, and the exponential function is time-consuming for every mini-batch sampling.

To strive a balance between simple and sophisticated strategy for the use of

experience replay, a new replacement strategy is thus proposed.

Algorithm 2: utilizing the experience in memory more effectively

function learn ($mini_batch$);
....;
mini_batch[:,replay_count_idx] $\leftarrow mini\_batch[:, replay\_count\_idx] + 1$
//When a mini-batch is taken from the memory, the replay counts for the corresponding samples increase.
....;
....;
function store ($datum$);
....;
memory.store(datum);
memory.pointer $\leftarrow memory.pointer + 1$
if $memory.pointer \geq memory.capacity$ **then**
    //avoid the sorted order in the data
    memory.shuffle();
    //sort the data by replay counts in ascending order
    memory.sort_by_replay_count();
    $memory.pointer \leftarrow memory.capacity//2$
else
end
....;

In DDPG's idea, the memory size should be very large so that the correlation of sampled data can be broken. As the sampling of the data is still uniformly random, the number of replays is independent of the experience. Also, this new strategy can increase the chance of data being replayed before they get replaced. It also breaks the time-correlation among data. The mini-batch can draw both old records and new records for replay.

3.2.5 Feature 5 - Storage Strategy for Experience Replay

One quantity should be carefully considered is the reward. It is not related to the network inputs or outputs, but it does affect the quantity of experiences replay. If the agent can seldom get "good" rewards and the stored memory are all biased to the "bad" rewards, the replacement of experience replay may easily ease such a precious experience replay. As a result, we should classify what are the experience getting "good" rewards and the experience getting "bad" rewards.

However, we usually do not know the range, distribution, or frequency of the rewards. It is completely unknown to the agent. For example, for the actions, we can apply the activation functions to control the range. For rewards, we do not have the range or other information for it - we do not know it is a good experience or bad experience in general.

For a common OpenAI Gym environment "Pendulum-v0" [2], the agent can always get the negative rewards as the best reward is 0.0. In contrast, for a Flappy Bird game[3], there is no maximum (accumulated) reward as long as the agent play it so well that never die.

A simple measure to deal with it is to divide the whole memory into "positive set" and "negative set" along with a record of the sum of rewards during the store. the new records will be put into either "positive set" or "negative set" which is determined by the reward is above or below the average.

If the rewards are normally distributed without a bias, the new entry should have equal chances to enter both sets of memory, so both sets of data are balance in size with similar replacement period.

When a mini-batch is required for training, the agent will sample the amount

[2]OpenAI Gym provides free game environments in the web; Pendulum-v0 is available on https://gym.openai.com/envs/Pendulum-v0/.

[3]For readers who do not know Flappy Bird, you can visit http://flappybird.io/ to play it.

number of experiences from both memory sets. Therefore the agent should be able to learn both "bad" and "good" experiences, while the large of the two memory sets can still break the correlation between data.

3.2.6 Layers' Diagram

Here are the network architecture for Actor net and Critic net. Both online evaluation and offline target adopt the same network architecture.

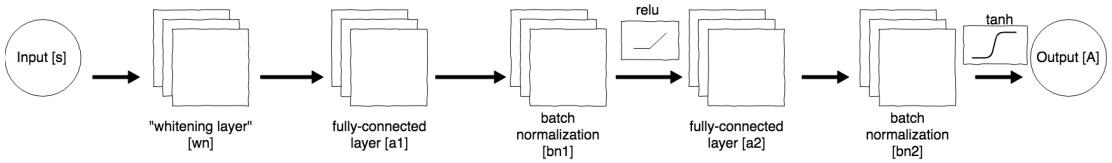


Figure 3.2: Neural Network for Actor: $A(s) = a; s \in \mathbb{R}^{n_s}, a \in (-1, 1)^{n_a}$

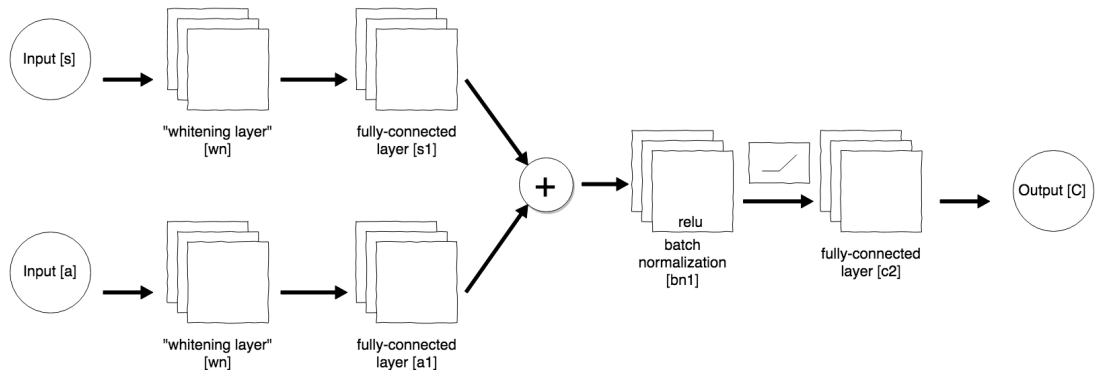


Figure 3.3: Neural Network for Critic: $C(s, a) = Q; s \in \mathbb{R}^{n_s}, a \in (-1, 1)^{n_a}, Q \in \mathbb{R}^1$

In the architecture, only the fully-connected layers change the number of features during the feeding. For a n_{gen} problem, there are $(n_{gen} - 1) * 2 + 4$ features as the state.

In the example case shown later, the number of features in the intermediate layers is 300. The inputs are "expanded" into 300 neurons and then combine into smaller outputs after the nonlinearities added via activation functions.

3.3 Flow charts for the program

In the program, there are two stages - training and evaluation.

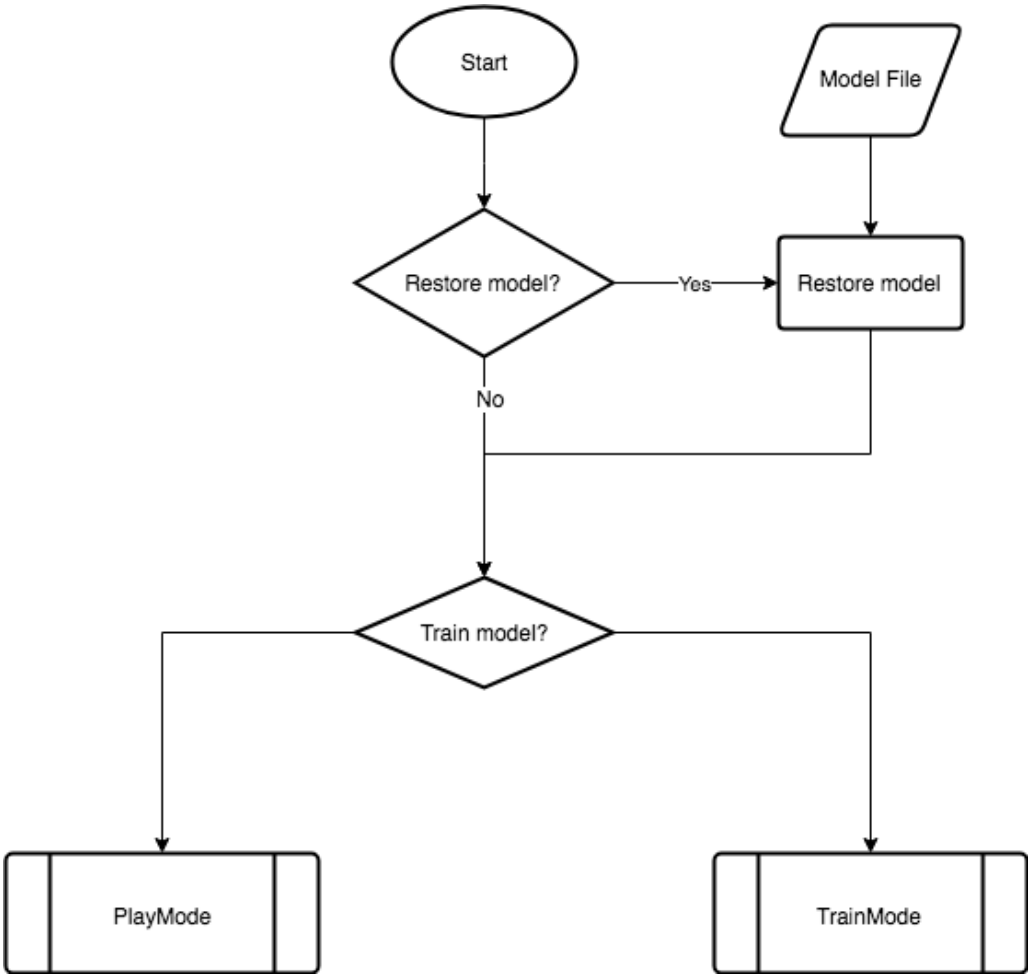


Figure 3.4: Flowchart - overall

The gaming part is just simply getting the action for the agent.

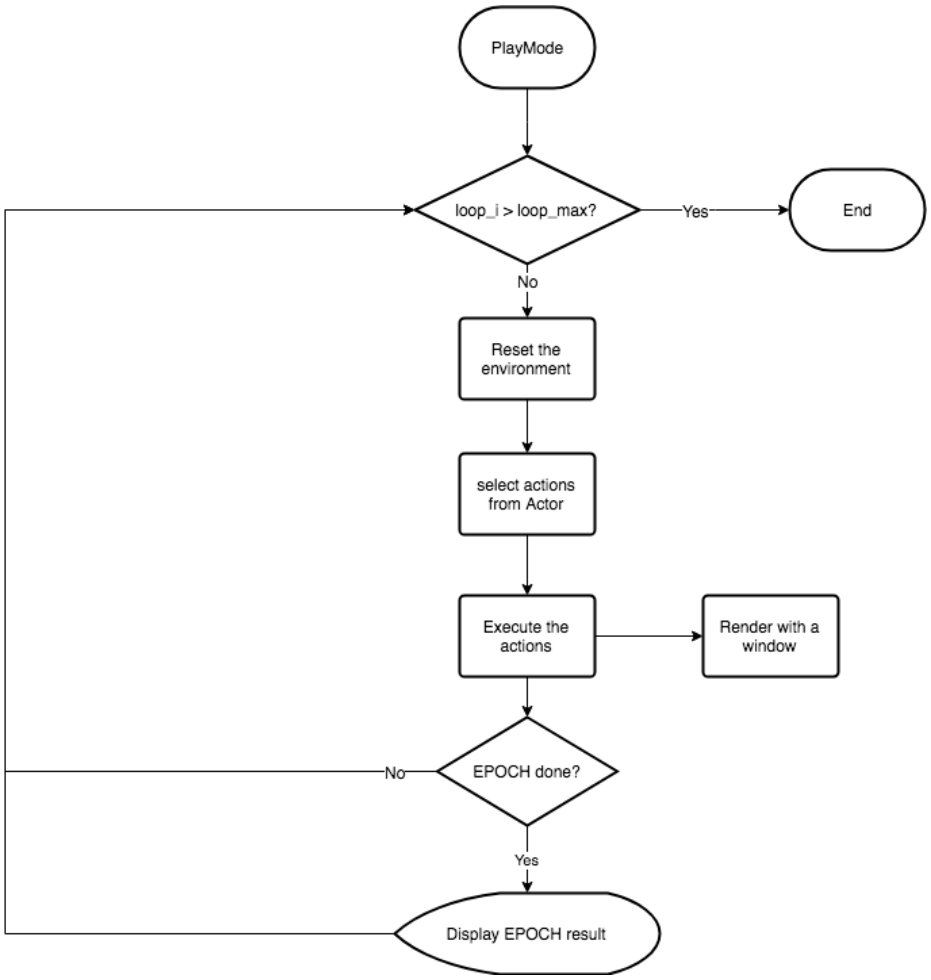


Figure 3.5: Flowchart - PlayMode

There are additional two sub-processes in the TrainMode - memorizing and learning

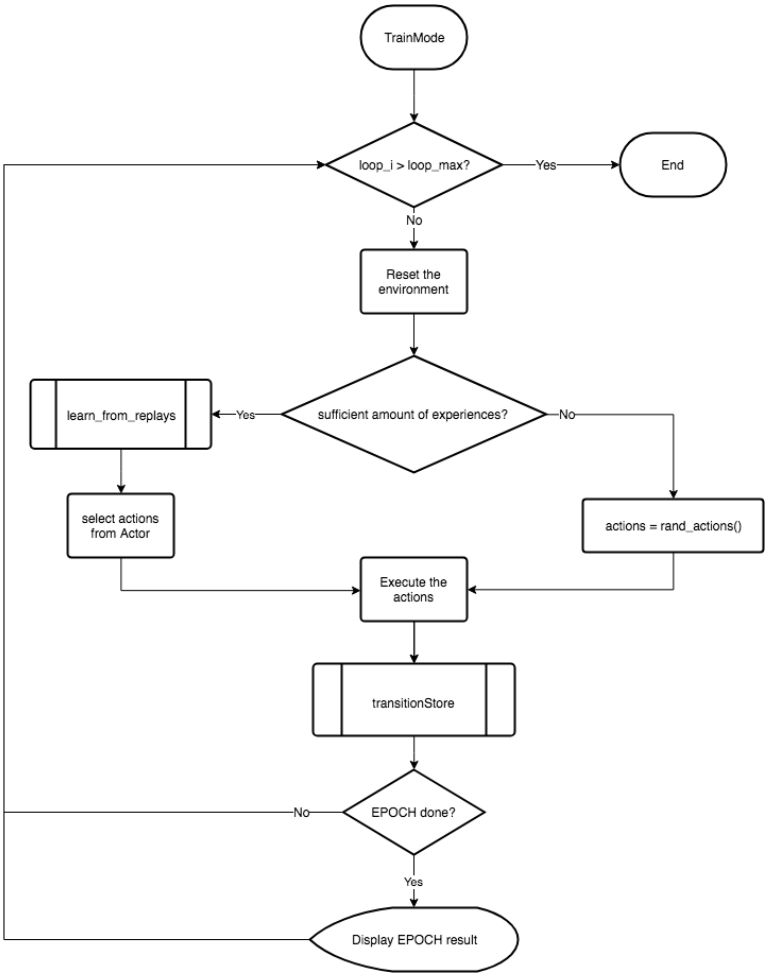


Figure 3.6: Flowchart - TrainMode

When a new datum is added to the memory (either "positive sets" or "negative sets"), it may replace some memory.

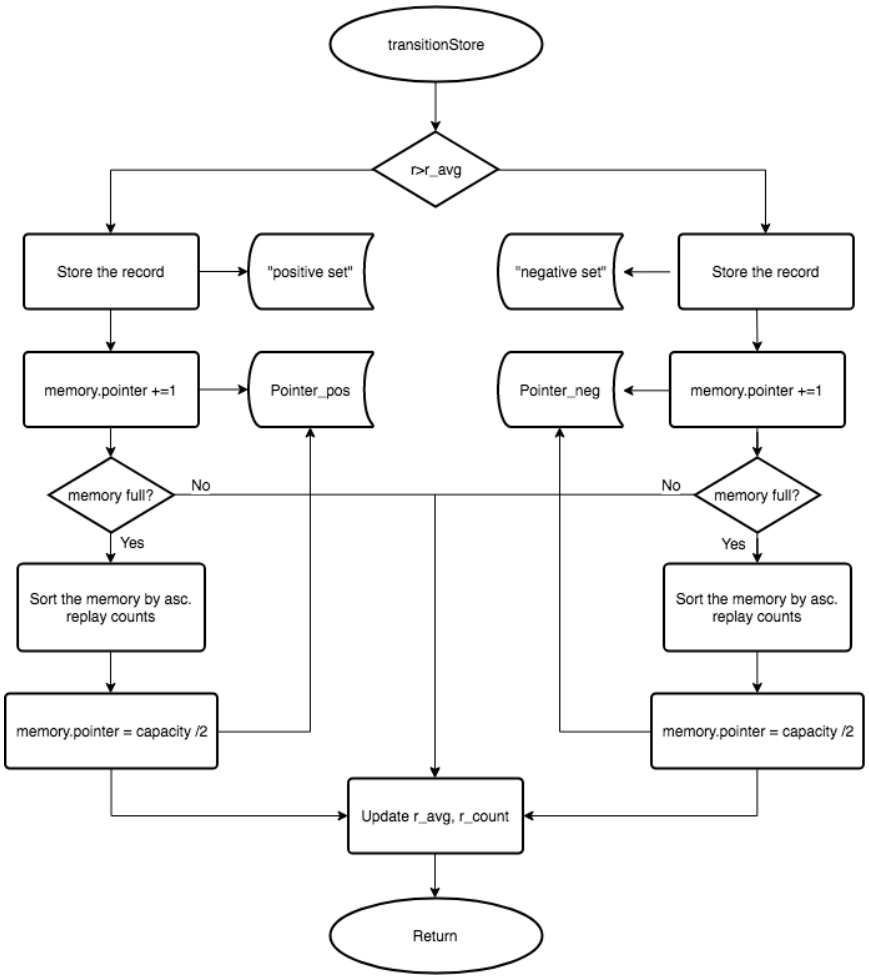


Figure 3.7: Flowchart - Memorizing

The actor is first updated by the gradient

$$\nabla_{\theta^A}\left(-\left[Q(s_k, A(s_k)) - \bar{Q}\right]\right)$$

The critic is then updated by the gradient

$$\nabla_{\theta^Q}\left(\frac{1}{n}\sum_{k=1}^{n}\left(Q(s_k, a_k) - Q_k^t\right)\right) \text{ where } Q_k^t = r_k + \gamma Q'(s'_k, A'(s'_k))$$

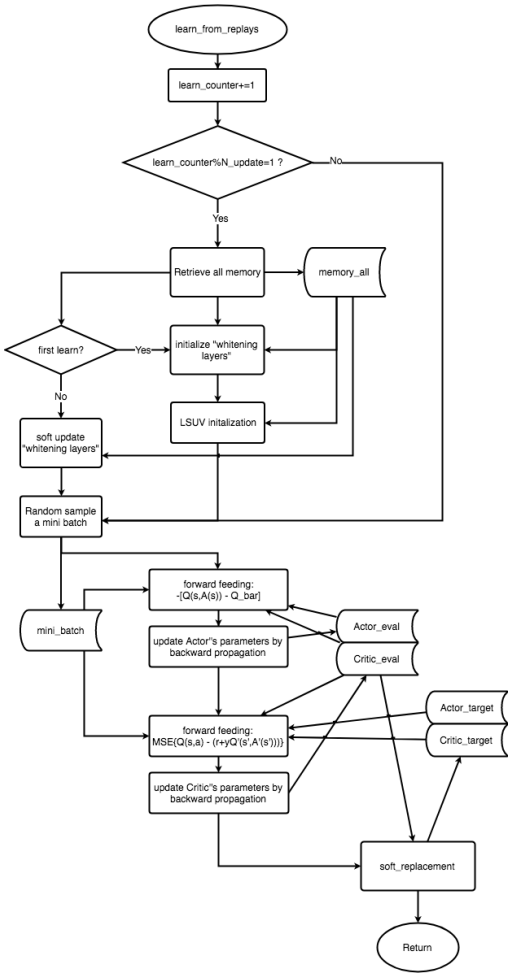


Figure 3.8: Flowchart - Learning

3.4 Methodology for Result Analysis

To evaluate the functionality and see whether it is successful to solve EDP, a simple case consisted of three generators is used. This test system was originally a 6-bus system. In this approach, the information of buses was not required, as the transmission loss matrix was sufficient to evaluate the generation loss due to the transmission lines.

Besides, a simple case can be easily trained with few neurons and layers in the neural network. It can be easily implemented on any personal PC without NVIDIA graphic cards.

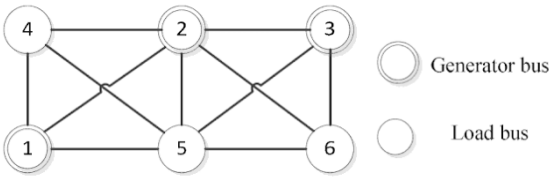


Fig. 1. The communication graph of the test system.

Figure 3.9: IEEE 6-bus system [58]

Table 3.2: Parameters for Economic Dispatches Problem

	Generation Limits		Cost Coefficients			Transmission Loss (Kron's formula)				
	P^{LB}	P^{UB}	c0	c1	c2	B			B0	B00
Unit 1	50.	200.	213.1	11.669	0.00533	0.0676	0.00953	-0.00507	-0.0766	0.040357
Unit 2	37.5	150.	200	10.333	0.00889	-	0.0521	0.00901	-0.00342	S_{base}
Unit 3	45.	180.	240.	10.833	0.00741	-	-	0.0294	0.0189	100 MW

$$P_j^{LB} \leq P_j \leq P_j^{UB} \text{ in MW}$$

$$Cost(P_j) = c0_j + c1_j * P_j + c2_j * P_j^2$$

$$P_{loss} = P * B * P' + B0 * P' + B00 \text{ in p.u.; } B^T = B$$

There are four plots for the training performance:

- "Actor Loss", $\left(-\left[Q(s_k, A(s_k)) - \bar{Q}\right]\right)$, is the measure of how well the agent picks the action.
- "Critic Loss", $\left(\frac{1}{n}\sum_{k=1}^{n}\left(Q(s_k, a_k) - (r_k + \gamma Q'(s'_k, A'(s'_k)))\right)\right)$, is the measure of how well the agent predicts the total cost reduction for the actions.
- Accumulated Reward - it is monotonic increasing against steps if the agent can reduce the cost for any conditions.
- Step Reward - the cost reduction per step

It was expected that both "Actor Loss" and "Critic Loss" attain their minimum and the accumulated reward was increasing with a high steepness. The Step Reward can check how well it plays for each epoch.

The DDPG training was performed in python 3.6.4 and PyTorch 0.3.1 with the following parameters. The parameters follow in common practices in neural network training.

Table 3.3: Hyperparameters for training the neural network

Hyperparameters			
Type of Optimizer	*AdamOptimizer*	Noise for Actor's output value	*None*
Learning Rate for "Actor Loss" (LR_A)	0.001	MEMORY_CAPACITY	40000 entries
Learning Rate for "Critic Loss" (LR_C)	0.001	BATCH_SIZE	32 entries
Discount Factor (γ)	0.9	Whitening Layer Update Interval	3000 steps
Soft Replacement Rate (τ)	0.01	Whitening Layer Update Rate	0.1

Besides, the activation function used for adding non-linearity was LeakyReLU ($a = 0.25$) instead of the common ReLU, as LeakyReLU shows a better training performance [59]. In the DDPG's paper, it mentioned an additional item "Noise" should be added for the agent to explore during the training. However, it did not give any detail of the selection of noise. For simplicity, the noise is chosen to be zero.

The training model and the graph were saved at an interval of 200 epochs.

After the training, there was a GUI window for users to run the agent.

The saved model was also used for evaluating the actual performance of cost

optimization for different power demand P_d. Five set of random initial powers are generated for each of the following pre-defined power demands: 140 MW, 240 MW, 340 MW and 440 MW.

CHAPTER 4

Results

The random samples were collecting for the first 47 seconds, and whitening and LSUV initialization took 6 seconds to complete. Then the models and graphs were saved at the interval of 11 minutes. As we do not know what is the optimal cost, the agent run 200 steps per epoch whenever the cost is optimal or not.

4.1 Training

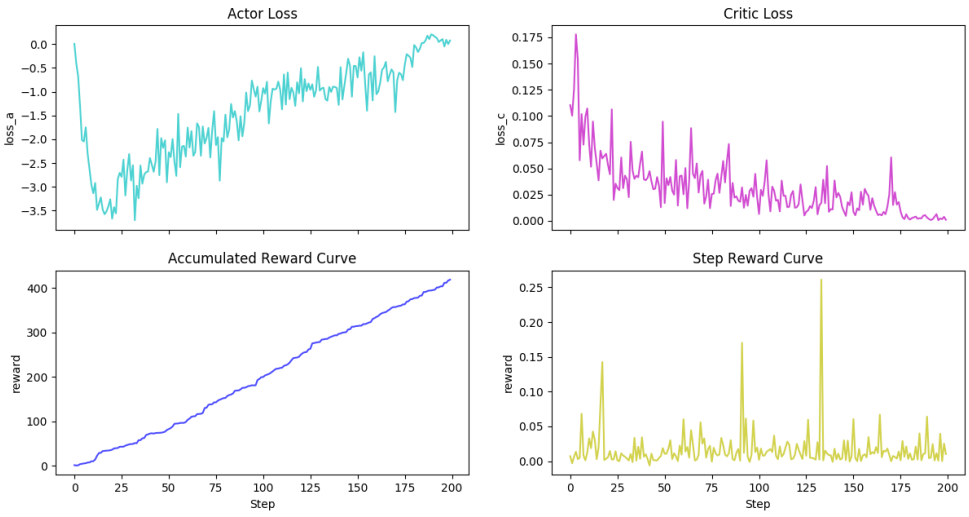


Figure 4.1: Result at the 200-th epoch

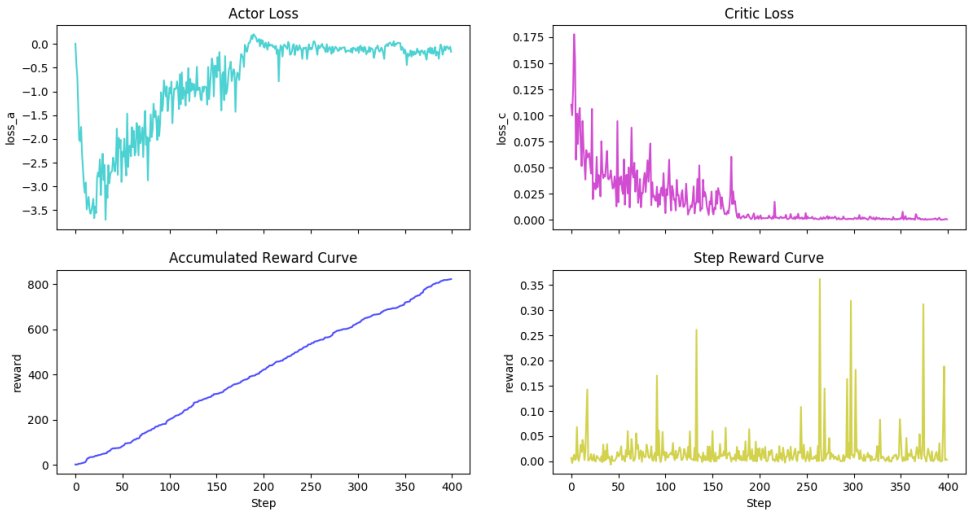


Figure 4.2: Result at the 400-th epoch

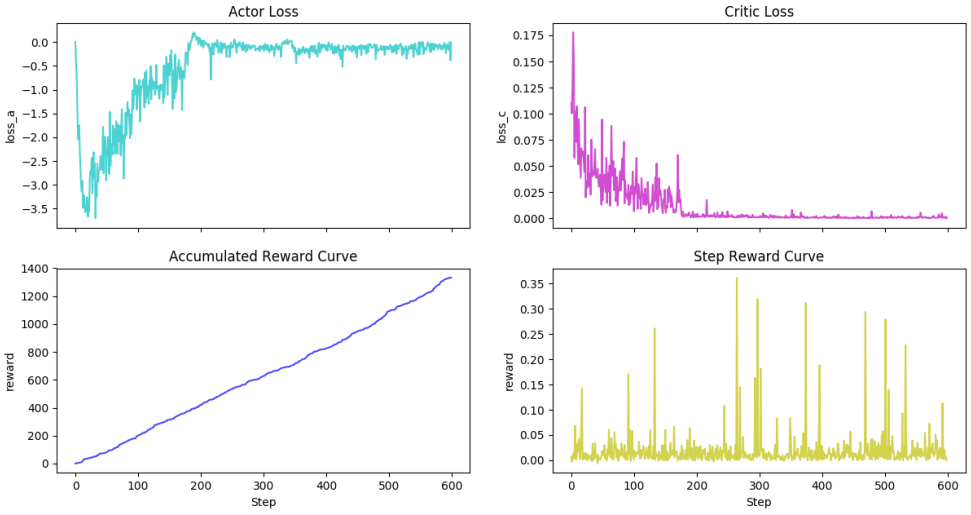


Figure 4.3: Result at the 600-th epoch

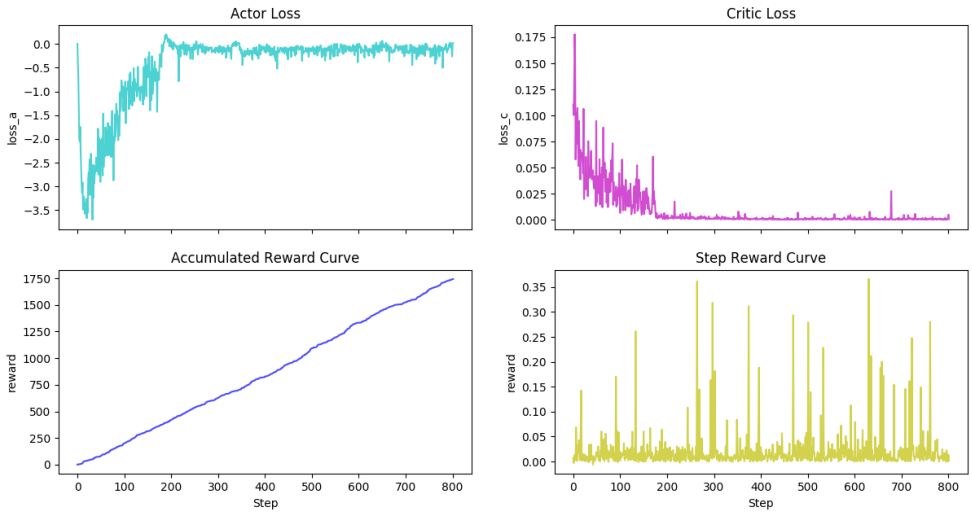


Figure 4.4: Result at the 800-th epoch

As shown, the parameters are stable starting from the 400-th epoch.

4.2 Evaluating

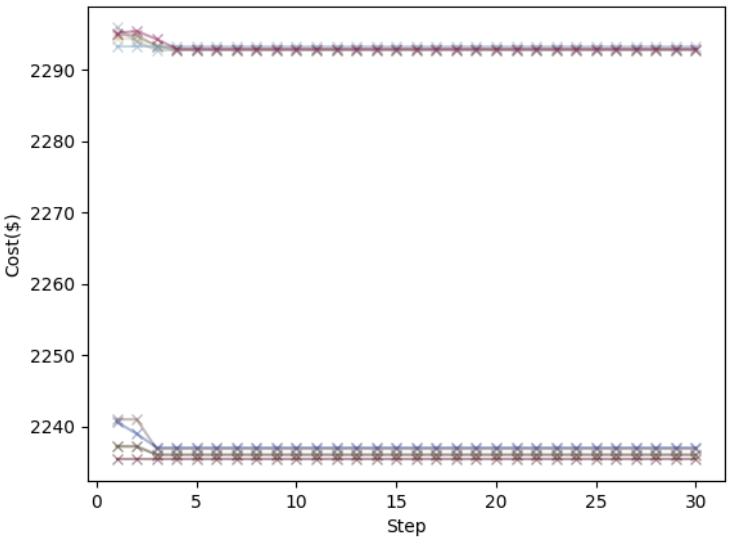


Figure 4.5: Optimizing the cost for $P_d = 140$ MW (lossy/lossless)

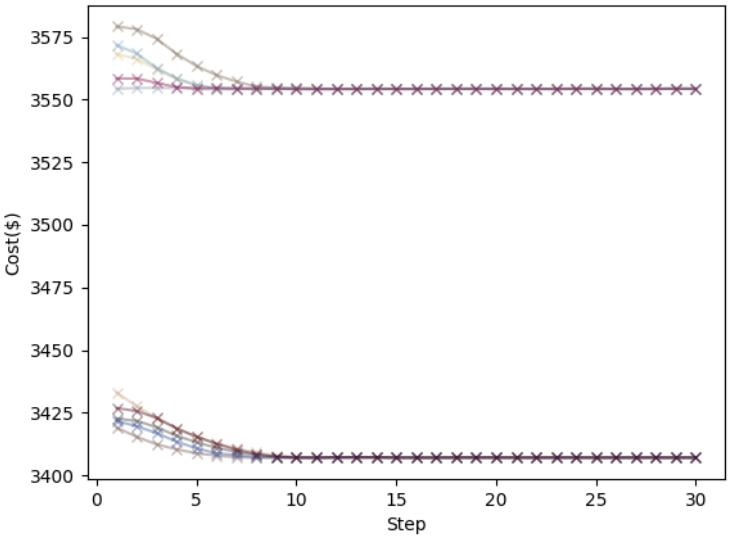


Figure 4.6: Optimizing the cost for $P_d = 240$ MW (lossy/lossless)

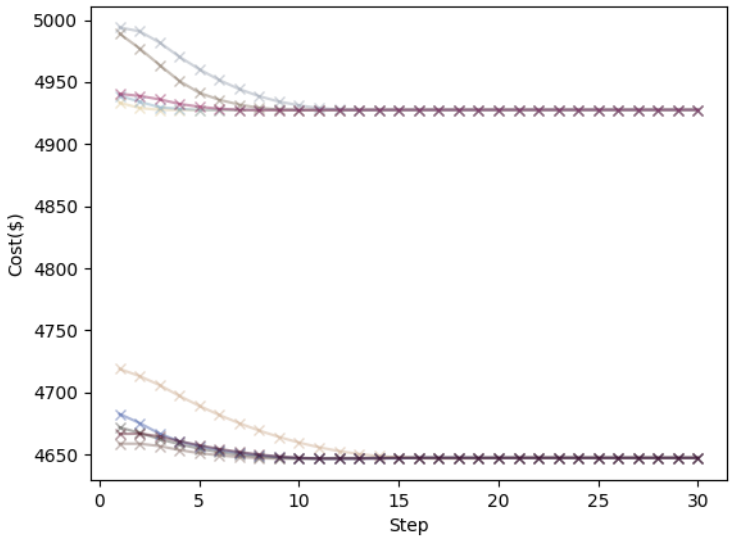


Figure 4.7: Optimizing the cost for $P_d = 340$ MW (lossy/lossless)

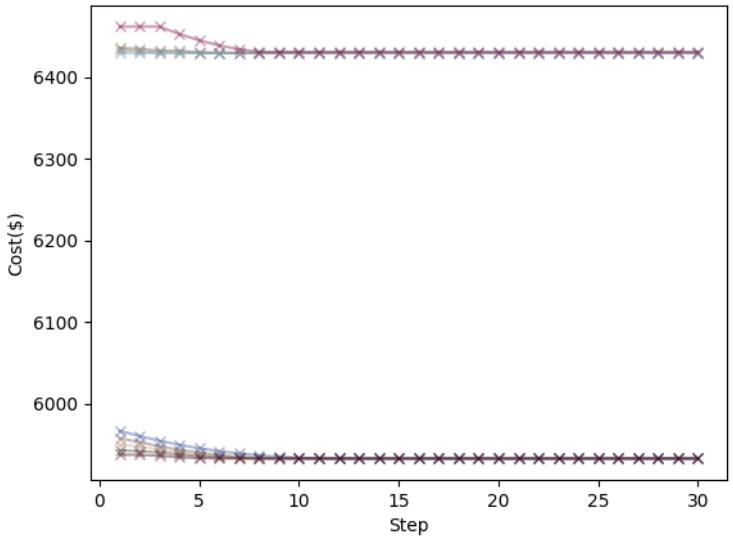


Figure 4.8: Optimizing the cost for $P_d = 440$ MW (lossy/lossless)

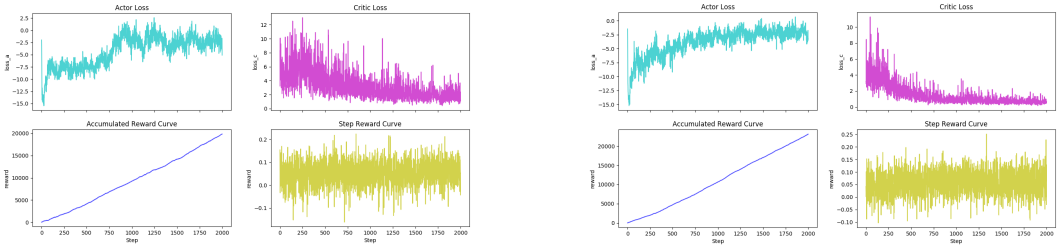
These results suggested that the power dispatches can converge to an optimal solution at a very fast rate (less than 30 iterations of the total cost). However, in the

lossy case with demand 140 MW, it cannot effectively converge to a single optimal solution. The game mechanism should be improved for the better constraint handling when the power dispatches are near to their generation limits.

4.3 Effects on the scale of a network

In the simple case, 300 neurons are sufficient. However, if the system adds the consideration of valve-point effects, the training is unstable with the noisy "critic loss". After adding one more layer in between the hidden neurons and the final output layer, the stability of "critic loss" is much better.

The coefficients and other settings are referred to the IEEE-6-bus case 2 in Kherfane et al.'s article. [60]



without modifying the network with an additional FC layer

Figure 4.10: Adding a complicity to the system (e.g. valve point effects)

The difference is just an additional fully connected layer with 300 input features and 300 output features with ReLU activation, but the result is completely different.

From the research carried by Jasmin et al., Q-learning can apply to the grids of a large scale too [42]. Perhaps a much better game mechanism which is similar to their research can achieve a better result and performance.

CHAPTER 5

Conclusion and Future Development

In this project, the neural network based reinforcement learning method has been successfully applied in a very simple case study. It can be treated as a good start although it is not complicated enough. The result of demand 140 MW also suggested that the game mechanism shall have a better design to jump over the infeasible solutions if present.

In the future development, the neural network must be on a large scale to solve different tasks, such as emission control and outages. As the number of minimal states increases, the number of neurons and layers will be also increased. We should carry more analysis on the scale of the neural networks when the game has to deal with more challenging issues.

However, the current game mechanism may be too complicated. The little variation can greatly impact the training as shown in the last section.

In the Jasmin et al.'s researches [42], the state included only the target demand instead of the full set of power dispatches. Perhaps the game can be redesigned like that to achieve the similar performance from their study with 20 generators system.

REFERENCES

[1] P. M. Pardalos, A. Žilinskas, and J. Žilinskas, *Non-Convex Multi-Objective Optimization*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-61007-8. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-61007-8.

[2] P. Jain and P. Kar, "Non-convex optimization for machine learning", 2017. DOI: 10.1561/2200000058. arXiv: 1712.07897.

[3] D. Jones, S. Mirrazavi, and M Tamiz, "Multi-objective meta-heuristics: An overview of the current state-of-the-art", *European Journal of Operational Research*, vol. 137, no. 1, pp. 1 –9, 2002, ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(01)00123-0. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221701001230.

[4] A. V. Savchenko, *Search techniques in intelligent classification systems*. Springer, 2016.

[5] P. Melin and W. Pedrycz, *Soft Computing for Recognition Based on Biometrics*. Springer, 2010, vol. 312.

[6] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups", *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[7] D. Park, M. El-Sharkawi, R. Marks, L. Atlas, and M. Damborg, "Electric load forecasting using an artificial neural network", *IEEE Transactions on Power Systems*, vol. 6, no. 2, pp. 442–449, 1991. DOI: 10.1109/59.76685. [Online]. Available: http://dx.doi.org/10.1109/59.76685.

[8] P. Vasant, *Innovation in power, control, and optimization : emerging energy technologies*. Hershey, PA: Engineering Science Reference, 2012, ISBN: 1613501390.

[9] .

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level

control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, p. 529, 2015.

[11] A. J. Wood, B. F. Wollenberg, and S. G. B., *Power generation, operation, and control*. J. Wiley & Sons, 2014.

[12] *U.s. energy information administration - eia - independent statistics and analysis*. [Online]. Available: `https://www.eia.gov/outlooks/ieo/index.php`.

[13] R. Yokoyama, S. Bae, T. Morita, and H. Sasaki, "Multiobjective optimal generation dispatch based on probability security criteria", *IEEE Transactions on Power Systems*, vol. 3, no. 1, pp. 317–324, 1988. DOI: `10.1109/59.43217`.

[14] C. Panigrahi, P. Chattopadhyay, R. Chakrabarti, and M Basu, "Simulated annealing technique for dynamic economic dispatch", *Electric power components and systems*, vol. 34, no. 5, pp. 577–586, 2006.

[15] C. Yingvivatanapong, "Multi-area unit commitment and economic dispatch with market operation components", 2007.

[16] J. P. Catalão, *Electric power systems: advanced forecasting techniques and optimal generation scheduling*. CRC Press, 2016.

[17] N. Ramana, *Power System Operation & Control*. Pearson Education India, 2010.

[18] Y. Song, G. Wang, P. Wang, and A. Johns, "Environmental/economic dispatch using fuzzy logic controlled genetic algorithms", *IEE Proceedings-Generation, Transmission and Distribution*, vol. 144, no. 4, pp. 377–382, 1997.

[19] 2018. [Online]. Available: `https://www.solver.com/convex-optimization`.

[20] L. N. d. Castro, *Fundamentals of natural computing: basic concepts, algorithms, and applications*. Chapman & Hall, 2006.

[21] I. Ziane, F. Benhamida, and A. Graa, "Simulated annealing algorithm for combined economic and emission power dispatch using max/max price penalty factor", *Neural Computing and Applications*, 2016. DOI: `10.1007/s00521-016-2335-3`.

[22] K. K. Vishwakarma and H. M. Dubey, "Simulated annealing based optimization for solving large scale economic load dispatch problems", *International Journal of Engineering Research & Technology (IJERT)*, vol. 1, no. 3, 2012.

[23] J Sasikala. and M Ramaswamy., "Optimal based economic emission dispatch using simulated annealing", *International Journal of Computer Applications*, vol. 1, no. 10, pp. 69–83, 2010. DOI: 10.5120/221-371.

[24] L. Ingber, "Simulated annealing: Practice versus theory", *Mathematical and computer modelling*, vol. 18, no. 11, pp. 29–57, 1993.

[25] K. Wong and C. Fung, "Simulated annealing based economic dispatch algorithm", *IEE Proceedings C Generation, Transmission and Distribution*, vol. 140, no. 6, p. 509, 1993. DOI: 10.1049/ip-c.1993.0074.

[26] A. Bakirtzis, "Genetic algorithm solution to the economic dispatch problem", *IEE Proceedings - Generation, Transmission and Distribution*, vol. 141, no. 4, p. 377, 1994. DOI: 10.1049/ip-gtd:19941211.

[27] D. Walters and G. Sheble, "Genetic algorithm solution of economic dispatch with valve point loading", *IEEE Transactions on Power Systems*, vol. 8, no. 3, pp. 1325–1332, 1993. DOI: 10.1109/59.260861.

[28] D. C. Walters and G. B. Sheble, "Genetic algorithm solution of economic dispatch with valve point loading", *IEEE transactions on Power Systems*, vol. 8, no. 3, pp. 1325–1332, 1993.

[29] Z.-L. Gaing, "Particle swarm optimization to solving the economic dispatch considering the generator constraints", *IEEE transactions on power systems*, vol. 18, no. 3, pp. 1187–1195, 2003.

[30] E. Davalo, N. Patrick, and A. Rawsthorne, *Neural networks*. Macmillan, 1993.

[31] A. Jain, J. Mao, and K. Mohiuddin, "Artificial neural networks: A tutorial", *Computer*, vol. 29, no. 3, pp. 31–44, 1996. DOI: 10.1109/2.485891.

[32] S. Sathasivam, N. Hamadneh, and O. H. Choon, "Comparing neural networks: Hopfield network and rbf network", *Applied Mathematical Sciences*, vol. 5, no. 69, pp. 3439–3452, 2011.

[33] A. J. Laferriere, *Hopfield network*. [Online]. Available: http://perso.ens-lyon.fr/eric.thierry/Graphes2010/alice-julien-laferriere.pdf.

[34] C. G. Y. Lau, *Neural networks: theoretical foundations and analysis*. New York, 1992.

[35] I. N. de Silva, L. Nepomuceno, and T. M. Bastos, "Designing a modified hopfield network to solve an economic dispatch problem with nonlinear cost function", in

Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on, IEEE, vol. 2, 2002, pp. 1160–1165.

[36] I. N. da Silva, L. Nepomuceno, and T. M. Bastos, "An efficient hopfield network to solve economic dispatch problems with transmission system representation", *International Journal of Electrical Power & Energy Systems*, vol. 26, no. 9, pp. 733–738, 2004.

[37] J. Moody and C. J. Darken, "Fast learning in networks of locally-tuned processing units", *Neural Computation*, vol. 1, no. 2, pp. 281–294, 1989. DOI: 10.1162/neco.1989.1.2.281.

[38] P Aravindhababu and K. Nayar, "Economic dispatch based on optimal lambda using radial basis function network", *International journal of electrical power & energy systems*, vol. 24, no. 7, pp. 551–556, 2002.

[39] K. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks", *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990. DOI: 10.1109/72.80202. [Online]. Available: http://dx.doi.org/10.1109/72.80202.

[40] P. J. Werbos, W. T. Miller, and R. S. Sutton, *Neural networks for control*. Cambridge, Mass: MIT press, 1990, ISBN: 0262132613.

[41] [Online]. Available: https://scholar.google.com/citations?user=5eBWoxEAAAAJ&hl=en.

[42] E. Jasmin, T. Imthias Ahamed, and V. Jagathy Raj, "Reinforcement learning approaches to economic dispatch problem", *International Journal of Electrical Power Energy Systems*, vol. 33, no. 4, pp. 836–845, 2011. DOI: 10.1016/j.ijepes.2010.12.008. [Online]. Available: http://dx.doi.org/10.1016/j.ijepes.2010.12.008.

[43] [Online]. Available: https://www.youtube.com/watch?v=bHeeaXgqVig&index=2&list=PL4uSLeZ-ET3xLlkPVEGw9Bn4Z8Mbp-SQc.

[44] T. Matiisen, *Demystifying deep reinforcement learning | computational neuroscience lab*, 2018. [Online]. Available: http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/.

[45] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning", 2016. eprint: arXiv:1602.01783.

[46] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay", *arXiv preprint arXiv:1511.05952*, 2015.

[47] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning", *arXiv preprint arXiv:1511.06581*, 2015.

[48] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.", in *AAAI*, vol. 16, 2016, pp. 2094–2100.

[49] Y. Li, *Deep reinforcement learning: An overview*, 2017. eprint: arXiv:1701.07274.

[50] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation", in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[51] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms", in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[52] K. Lin, S. Wang, and J. Zhou, *Collaborative deep reinforcement learning*, 2017. eprint: arXiv:1702.05796.

[53] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach", *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.

[54] S. B. Driss, M Soua, R. Kachouri, and M. Akil, "A comparison study between mlp and convolutional neural network models for character recognition", in *Real-Time Image and Video Processing 2017*, International Society for Optics and Photonics, vol. 10223, 2017, p. 1 022 306.

[55] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, *Continuous control with deep reinforcement learning*, 2015. eprint: arXiv:1509.02971.

[56] D. Mishkin and J. Matas, *All you need is a good init*, 2015. eprint: arXiv:1511.06422.

[57] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[58] Z. Jiaxing *et al.*, *Energy And Mechanical Engineering-Proceedings Of 2015 International Conference*. World Scientific, 2016.

[59] B. Xu, R. Huang, and M. Li, *Revise saturated activation functions*, 2016. eprint: arXiv:1602.05980.

[60] Y. M.K.R. K. F. Kherfane N., 2016. [Online]. Available: http://www.jee.ro/covers/art.php?issue=WR1449664094W56681e5e35584.

Appendices

A Layer-based Whitening Transformation in PyTorch

```
def whiten(Xnp):
    # this is to whiten the numpy-based array
    MU=np.mean(Xnp,axis=0)
    U,S,V =np.linalg.svd(np.cov(Xnp.transpose()))
    S[np.abs(S)<1e-9]=1.        # avoid 1/0 for constant states
    K=S**(-1/2)

    Z=U@np.diag(K)@V

    # Y = (X -MU) @ Z
    return MU,Z

def whitenTorch(X):
    # this is to whiten the torch-based array
    MUnp,Znp=whiten(X.numpy())
    return torch.from_numpy(MUnp).type_as(X),torch.from_numpy(Znp).type_as(X)

class WhitenLayer(nn.Module):

    def __init__(self, n_features):
        self.n_features = n_features
        super().__init__()

        self.weight = torch.nn.Parameter(torch.Tensor(n_features, n_features),requires_grad=False)
        self.bias = torch.nn.Parameter(torch.Tensor(n_features),requires_grad=False)
        # if bias:
        #     self.bias = Parameter(torch.Tensor(n_features))
        # else:
        #     self.register_parameter('bias', None)
        self.reset_parameters()
        self.weight.underWhitenLayer=True
        self.bias.underWhitenLayer=True

    def reset_parameters(self):
        self.initialized=False
        self.weight.data[:]=torch.eye(self.n_features)
        self.bias.data[:]=0.
        self.weight.detach_()
        self.bias.detach_()

        self._weight=self.weight.data.clone()
        self._bias=self.bias.data.clone()
        self.training = False   # this layer is always untrainable

    def fetch(self,sampleMU, sampleZ,firstLearn=True):

        # Y = (X - MU) @ Z
```

```
    def fetch(self,sampleMU, sampleZ,firstLearn=True):

        # Y = (X - MU) @ Z
        # Y = X @ Z - MU @ Z

        # W = Z.t()
        # b = -MU@Z

        # F.Linear: Y = X @ weight.t()  + bias

        W_new=sampleZ.t()
        b_new= - (sampleMU @ sampleZ).view(self.n_features)

        if not firstLearn:
            W_new=(1-learnCounter_rate)*self.weight.data[:]+learnCounter_rate*W_new
            b_new=(1-learnCounter_rate)*self.bias.data[:]+learnCounter_rate*b_new


        self.weight.data[:] = W_new
        self.bias.data[:] =b_new

        self.update()

    def update(self):
        # call after fetch and load_state_dict
        self._weight=self.weight.data.clone()
        self._bias=self.bias.data.clone()
        self.training = False   # this layer is always untrainable

    def ensure(self):
        self.weight.data[:]=self._weight
        self.bias.data[:]=self._bias
        self.training=False                # this layer is always untrainable
        self.weight.detach_()
        self.bias.detach_()

    def forward(self, input):
        # ensure the weight are the one stored, instead of "trained"
        self.ensure()
        return F.linear(input, self.weight, self.bias)

    def __repr__(self):
        return self.__class__.__name__ + '(' \
            + 'n_features=' + str(self.n_features) \
            + ', initialized=' + str(self.initialized) + ')'
```

Figure A.1: code for the whitening layer