Assignment1 Report

Jianxiao Luo 300123069

September 2020

1 Introduction

This assignment is to use polynomial models (1) to fit a bunch of data, which is generate by a specific equation (2).

$$Y = a_0 + a_1X + a_2X^2 + ... + a_dX^d \tag{1}$$

$$Y = cos(2\pi X) + Z \tag{2}$$

In equation(2), X is a set of random value between (0, 1) and Z is a zero mean Gaussian random variable with variance σ^2 , and Z is independent of X.

2 A

First of all, we need to generate the data, by defining a function ***getData()***. The parameters of this function is the number of data(N), the variance σ^2 and a Boolean value of whether to plot the figure(draw). I used PyTorch package in my program. The functions I used in this part is *torch.rand()*, *torch.normal()* and *torch.cos()*.

The following figure [1] is the generated data when $N = 200$ and $\sigma = 0.01$. It returns two tensors X and Y with size 200.

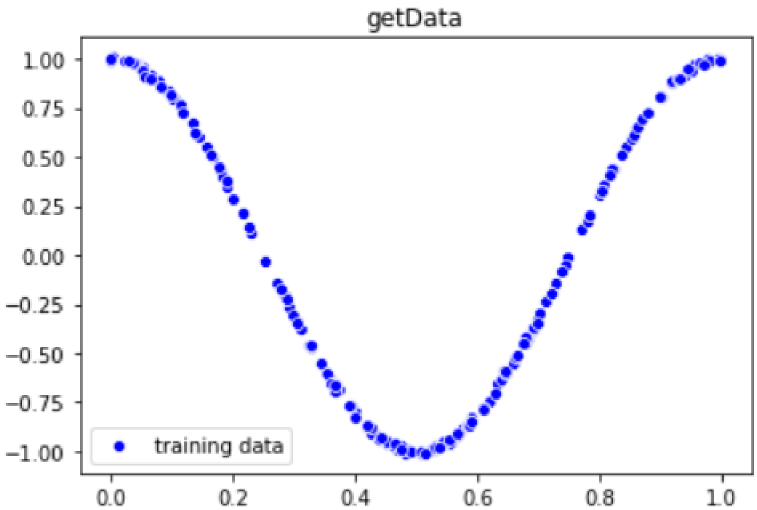

Figure 1: The data generate with $N = 200$, $\sigma = 0.01$

3 B

Define a function ***getMSE()*** to get the Mean Squared Error(MSE) between the true Y and fitted Y (3). The parameters of this function are the data X, Y, and the coefficients vector *ad*.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{3}$$

All we have to do is to get the predicted $\hat{Y}$ and then compute the MSE according to the equation above. To achieve this, I extended the X from $(1 \times N)$ to $((D+1) \times N)$ (4), where D is equal to the degree of polynomial.

$$X = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_N \\ \dots & \dots & \dots & \dots \\ x_1^d & x_2^d & \dots & x_n^d \end{bmatrix} \tag{4}$$

In this case, we can use *torch.matmul(ad, x)* to get $\hat{Y}$. Then return the result of MSE *torch.mean(torch.square(y-y_ex))*.

4 C

To fit data to a degree-d polynomial, another function ***fitData()*** is needed. I used mini-batched SGD here, these are the parameters I need to define:

X, Y - The data generated in ***getData()***
D - The degree of polynomial
batch_size - The size of each small batch
n_epochs - The epochs times

First of all, I need to form the mini batch, which means that I need to pick data randomly from X and should cover all the data. I use a function in torch *torch.randperm(x.size()[0])* to generate a random permutation of integers from 0 to X's size, then each time I pick batch_size of index. The vector of polynomial coeffecients is first generated randomly and updated in each batch. I run a loop of n epochs and a loop to traverse all the mini batches inside it. In each loop, *getMSE()* is used to get the MSE of the current polynomial.

Equation (5) is the way to update coefficient vector. I calculate the mean of the loss in every batch and plot its curve. Figure [2] is the figure of the E_{in} with N=1000, d=20, batch_size=100, n_epoch=3000. E_{in} declined fast in the first 1500 epochs, especially in the first 300 ones. In the last epoch, it stays at 0.024. For the E_{out}, it is 0.025, which is a little bit higher than Ein.

$$\theta^{new} = \eta^{old} + \lambda \cdot \frac{1}{|\beta|} \sum_{(x,y)\in\beta} 2(y - \theta^{old^T} x)x \tag{5}$$

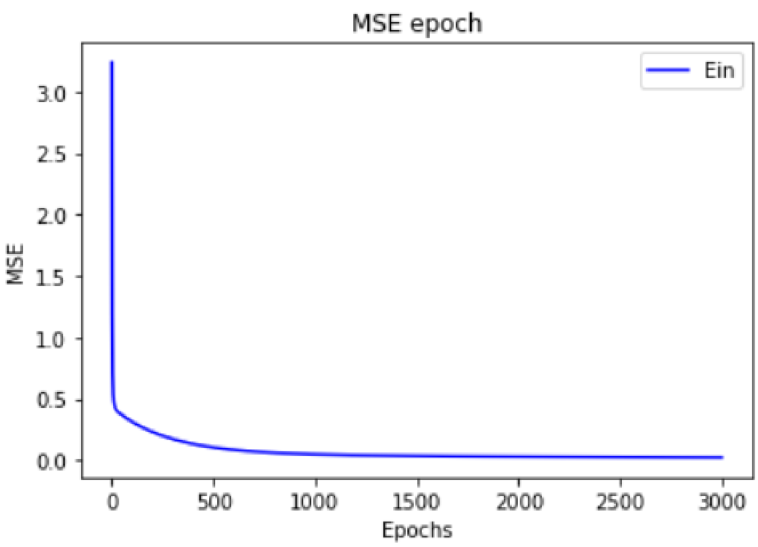

Figure 2: E_{in} with N=1000, d=20, batch_size=100, n_epoch=3000

5 D

The function ***experiment()*** is to run the above functions for M trails and get the average E_{in} and E_{out}. Then for the coefficients, we need to get the average coefficient vector over the M trails, which means that we have a $(M \times D)$ matrix after M trails, we need to do the mean in the column direction, in order to shrink it to $(1 \times D)$ vector. Then we use the new coefficient vector to deal with new generated test data and get E_{bias} as a result.

To test this function, I call it and set its parameter to this: *N=200, sigma=0.01, d=20, batch_size=20, n_epoches=1000, M=30.* After a bit long running time, I get these result:

$$E_{in}\text{: } 0.05068271979689598$$
$$E_{out}\text{: } 0.05154363065958023$$
$$E_{bias}\text{: } 0.052007194608449936$$

Here we can see, by reducing the n_epoches parameter, all of the results are higher than the above function. Within these three results, E_{in} is still the lowest one, with a higher number in E_{out} and a little bit higher number in E_{bias}. It is a little bit surprising that E_{bias} has not the lowest MSE, but even completely opposite. I have done the test for several times, the results did not change.

6 E

Section E is to play *experiment()* function with different parameters, with suggesting values: $N \in \{2, 5, 10, 20, 50, 100, 200\}$, $d \in \{0, 1, 2, ..., 20\}$ and $\sigma \in \{0.01, 0.1, 1\}$.

- The first experiment I did is to see the MSE curve when I increase the **model complexity**. So I set N and to a small number 12, and σ to 0.01. Then I choose a number which can be divided by 12 - 4 to be the batch_size and a normal n_epochs, 800. For the D, I let it be a list from 0 to 30. The figure [3] below shows the MSE curve while the data is relatively simple and the complexity of model increases. For E_{in}, it is going down all along, and reaching 0. However, for E_{out}, it decreased in the first place, but started to increase after d=9. The gap between E_{in} and E_{out} is E_{gen}, which increases along with the model complexity. The green line I draw, it can be seen as a division of under-fitting and over-fitting. If I

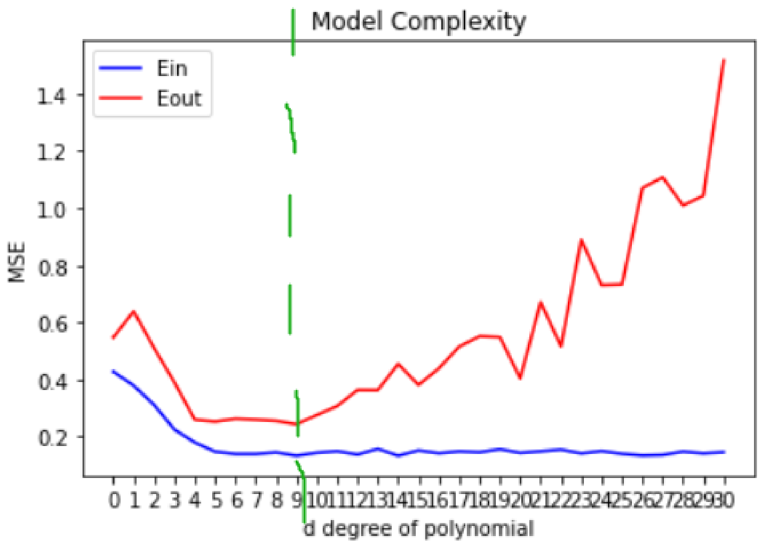

Figure 3: E_{in} and E_{out} when increasing the model complexity

set N to a higher number, then it is less probably to cause overfitting. The following figure 4 is the curve of MSE while N is equal to 100. With a large dataset, the curve of E_{out} does not grow when d increases to a high value.

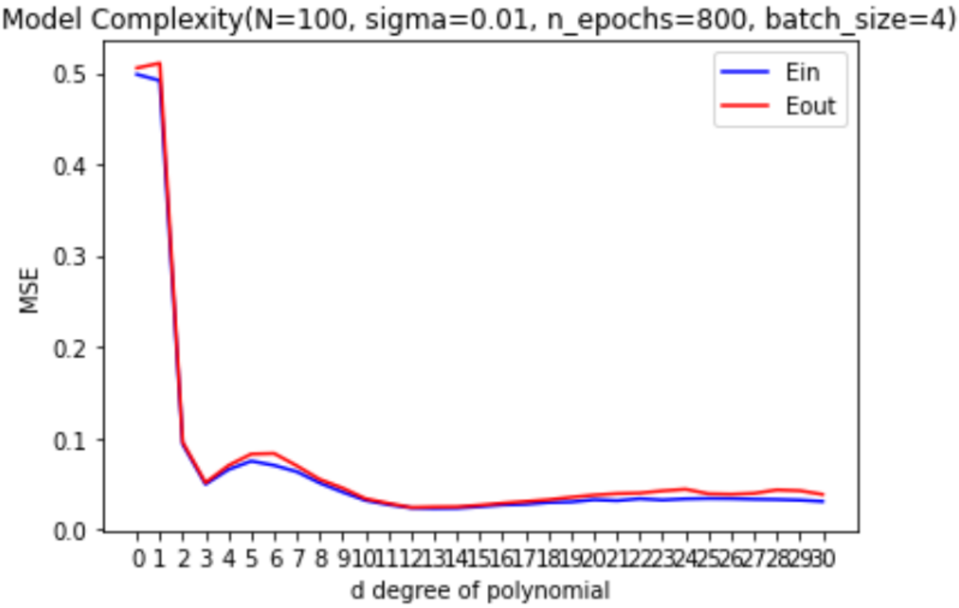

Figure 4: E_{in} and E_{out} with big N and small batch_size and increasing model complexity

- **Different sample size N** also have different affect on the E_{in} and E_{out} in sample model as well as complex model. So I set $N \in \{2, 6, 12, 20, 50, 100, 200\}$, $d \in \{10, 30\}$, $\sigma = 0.01$ and n_epoches=1000, to see the MSE curve of E_{in} and E_{out} in different complexity model.

  From the following two figure 5 and 6 we can see that they have a similar trend, where E_{out} goes down sharply and E_{in} grows gently. It is because with the growing of N, MSE of training set (E_{in}) will grow a bit, but owing to the fact that the performance of the model becomes is getting better and better, E_{out} declines sharply. They all settle out in the end, and converge to a small value, the MSE of complex model is even a bit lower than the simpler one.

  Moreover, the points that two curves converge to a stable low level is different in these two figures, the one in figure 5 appears later than the one in figure 6. It means that, in the simpler model, we need more data to get a accurate model than the complex one.

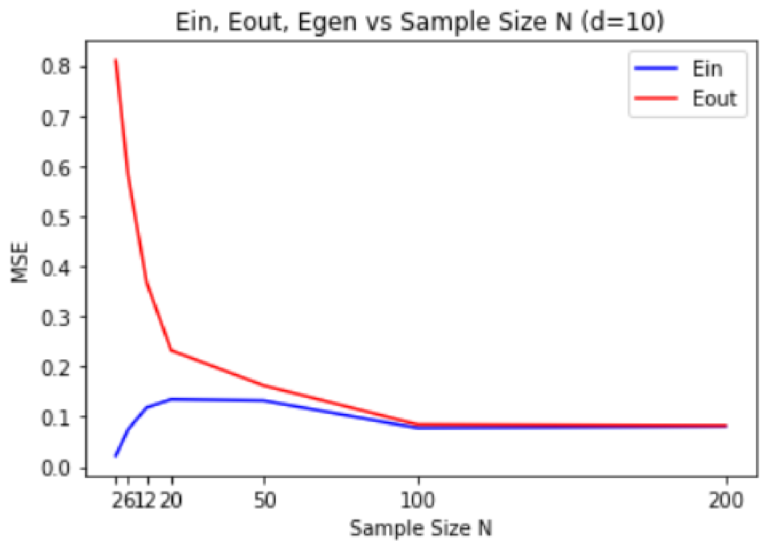

Figure 5: Different sample size N with simple model

- The third experiment is to change **the value of** σ. I set $\sigma \in \{0.01, 0.1, 0.5, 1, 2\}$, N=100, d=10,

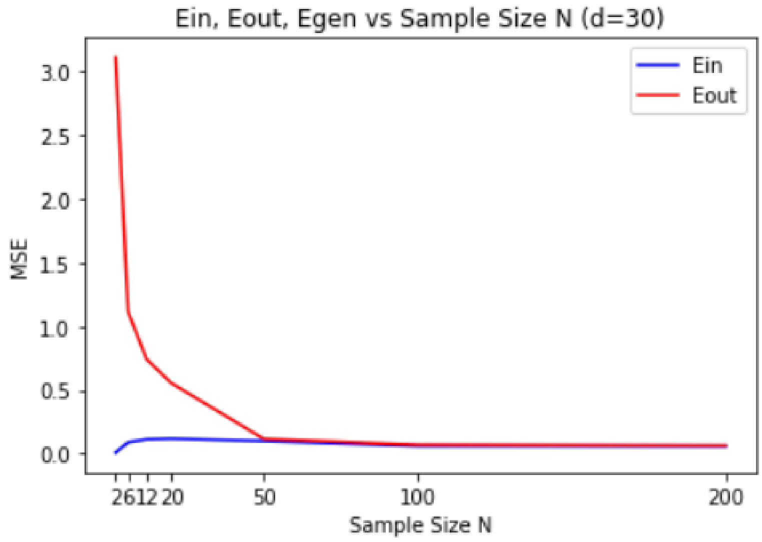

Figure 6: Different sample size N with complex model

batch_size=10 and n_epochs=800. Obviously, with the increase of σ, both E_{in} and E_{out} increase. An interesting phenomenon is that with current combination of parameters, E_{in} is greater than E_{out}.

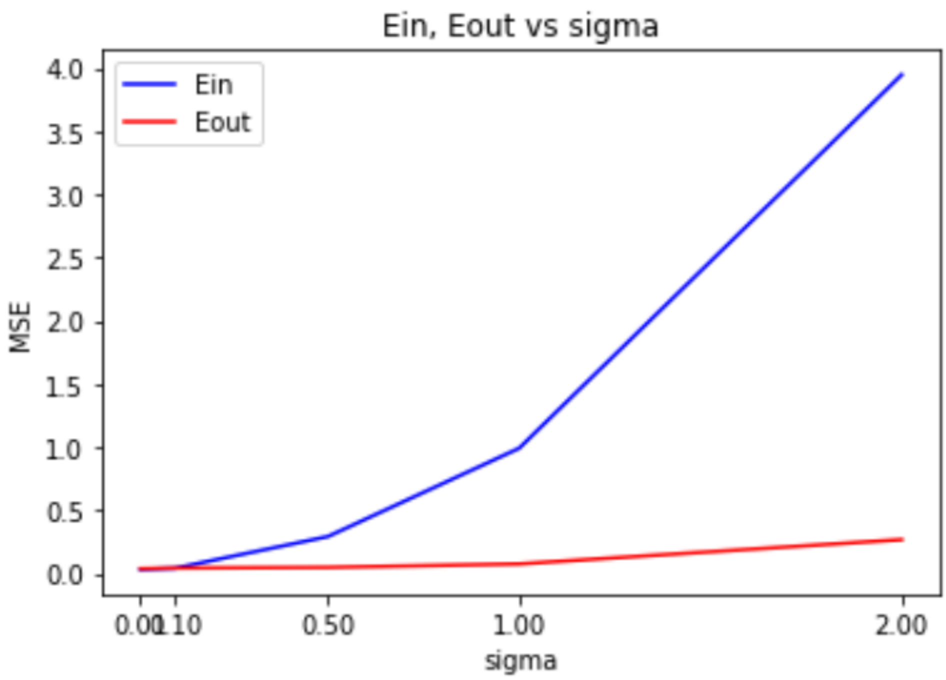

Figure 7: E_{in} and E_{out} VS sigma

7 F

In order to reduce the probability of overfitting, we should apply regularization to our model. The most popular one is L2-Regularizer (a.k.a, "weight decay"). The main purpose of weight decay was to suppress the magnitude of the updated parameters. The way to do it is to adding a penalty term to the loss function to reduce the affect of model complexity.

The loss function is defined as (6):

$$Loss = MSE + \lambda_{wd} \|\theta\|_2^2 \tag{6}$$

Then the derivative of $Loss$ with respect to θ is (7):

$$\frac{\partial Loss}{\partial \theta} = \frac{\partial MSE}{\partial \theta} + 2\lambda_{wd}\theta \tag{7}$$

The update of θ in mini_batch SGD should be ():

$$\theta^{new} = (1 - 2\lambda)\theta^{old} - \eta \frac{1}{|\beta|} \sum_{(x,y)\in\beta} 2(y - \theta^{old^T} x)x \tag{8}$$

For better observing the performance of weight decay, I apply the changed functions to the same parameter combination as the first experiment in section E (N=12, $\sigma = 0.01$, batch_size=4, n_epochs=800, $d \in [0, 30]$). Figure 8 is the result. As we can see, it is not overfitting anymore, but it also has a higher MSE both in training data and testing data, which obviously is the effect of weight decay.

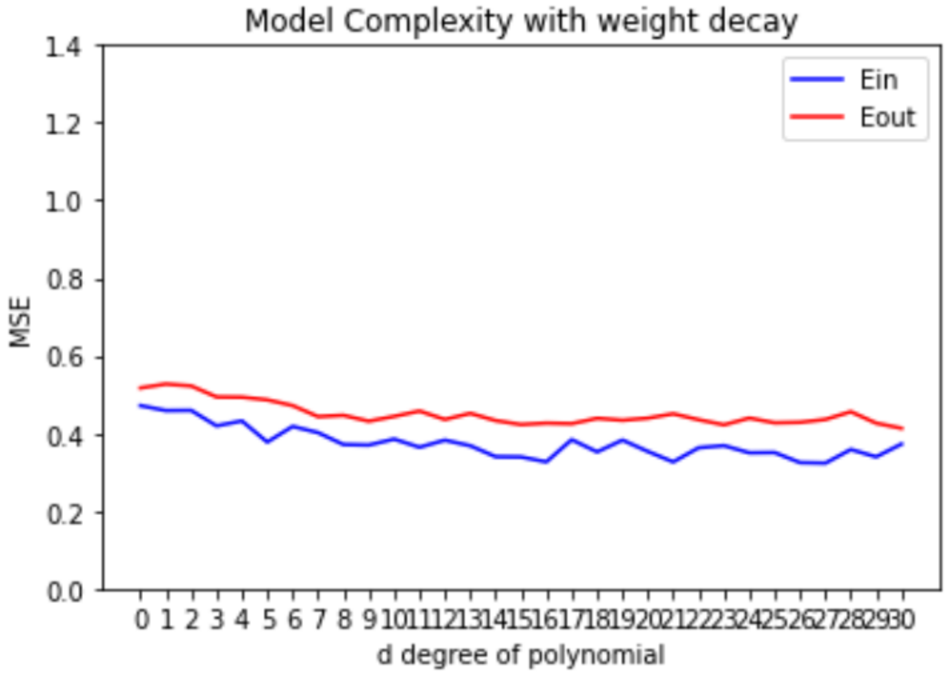

Figure 8: Added weight decay to the model