《模式识别与机器学习 A》实验报告

实验题目：多项式拟合正弦函数实验

学号：_____

姓名：_____

1.实验目的

在本次实验中，我们需要使用梯度下降法，通过高阶多项式函数来拟合正弦函数。通过本次实验，要掌握机器学习训练拟合原理（无惩罚项的损失函数）、掌握加惩罚项（L2范数）的损失函数优化、梯度下降法、理解过拟合、克服过拟合的方法(如加惩罚项、增加样本)。

2．实验内容

生成一组含有噪声的来自某正弦函数的散点。尝试使用梯度下降法，利用多项式函数拟合正弦函数。并且考察所得模型的泛化能力。如果模型出现了过拟合现象，尝试使用增大数据量、增大惩罚项、调整超参数等方式减小过拟合问题，增大模型泛化能力。

3．实验环境

我在本实验中使用了 python 3.10, 利用了 numpy 库进行辅助矩阵运算，使用 matplotlib 库进行图像绘制。

4．实验过程、结果及分析（包括代码截图、运行结果截图及必要的理论支撑等）

4.1 理论推导

本次实验中，我使用的损失函数是 L2 损失函数，即:

$$L = (\hat{y} - y)^2 + \lambda * \|w\|_2^2 \tag{1}$$

其中λ是惩罚系数:

$$\hat{y} = X \cdot W \tag{2}$$

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \tag{3}$$

$$W = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \tag{4}$$

使用 L2 损失函数是合理的，这是由于我们生成的噪声符合高斯分布。若高斯分布取到最大似然，同时 L2 损失函数也会最小。列向量 W 由多项式各项系数组成。相应的，矩阵 X 是对应的多项式的项本身。

我们采用梯度下降法进行拟合。可以将 L 对列向量 W 求导，记为列向量 g,则:

$$g = X^T \cdot (\hat{y} - y) + 2 * \lambda * W \tag{5}$$

那么，W 的更新公式就很容易给出了:

$$W = W - \eta * g \tag{6}$$

4.2 实验过程

4.2.1 生成数据并加入噪声

```
def get_data(a, w, cont, m):
    x = np.linspace(-3, 3, cont)
    y = a * np.sin(w*x) + np.random.normal(0, 10, cont)
    input_x = x[:int(cont*2/3)]
    input_y = y[:int(cont*2/3)]
    # input_x = x
    # input_y = y
    W = np.zeros(m+1)
    return x, y, input_x, input_y, W
```

将区间[-3, 3]划分为 cont 份，生成来自曲线$y = a * sin(w * x)$的点，同时加入符合（0, 10）高斯分布的噪声。取前 2/3 的样本作训练样本，以说明是否存在过拟合问题。

4.2.2 训练

```
# train
for epoch in range(Epoch):
    y_hat = predict(X, W)
    loss = get_loss(input_y, y_hat, W, flag)
    if epoch % 1000 == 0:
        print("loss={}".format(loss))
    if flag == 0:
        g = (X.T.dot(y_hat-input_y))/len(input_y)
        G += g**2
        eta = alpha/(np.sqrt(G)+epsilon)
        W = W - eta * g
    else:
        g = (X.T.dot(y_hat-input_y))/len(input_y) + 2*lamb*W
        G += g**2
        eta = alpha/(np.sqrt(G)+epsilon)
        W = W - eta * g
    if epoch % 1000 == 0:
        print(W)
```

根据前面给出的公式进行迭代。

4.2.3 给出结果

```
print("W1 = {}".format(W1))
print("W2 = {}".format(W2))

loss1 = get_loss(y, y_hat1, W, flag)
loss2 = get_loss(y, y_hat2, W, flag)
print("loss1={}".format(loss1))
print("loss2={}".format(loss2))
plt.plot(x, y, 'bo', label='real-data')
plt.plot(x, y_hat1, 'r', label='predicted-data-unpunished')
plt.plot(x, y_hat2, 'c', label='predicted-data-punished')
plt.legend()
plt.show()
```

给出结果并绘制图像。

4.3 实验过程中遇到的问题

正弦函数和多项式函数在性质上存在很多差异。例如，正弦函数是一个周期函数，而多项式函数不是一个周期函数，更坏的情况是，一旦自变量的绝对值变大，多项式函数的高次项值就会很快变大。

这首先会给训练带来麻烦。看我们前面给出的公式:

$$W = W - \eta * (X^T \cdot (\hat{y} - y) + 2\lambda * W) \tag{7}$$

可以发现，高次项的值会影响高次项系数的拟合。统一的η值肯定是行不通了，否则我们会陷入两难的境地，η值太大，高次项系数就很难收敛，η值太小，低次项系数收敛慢，训练时间长。

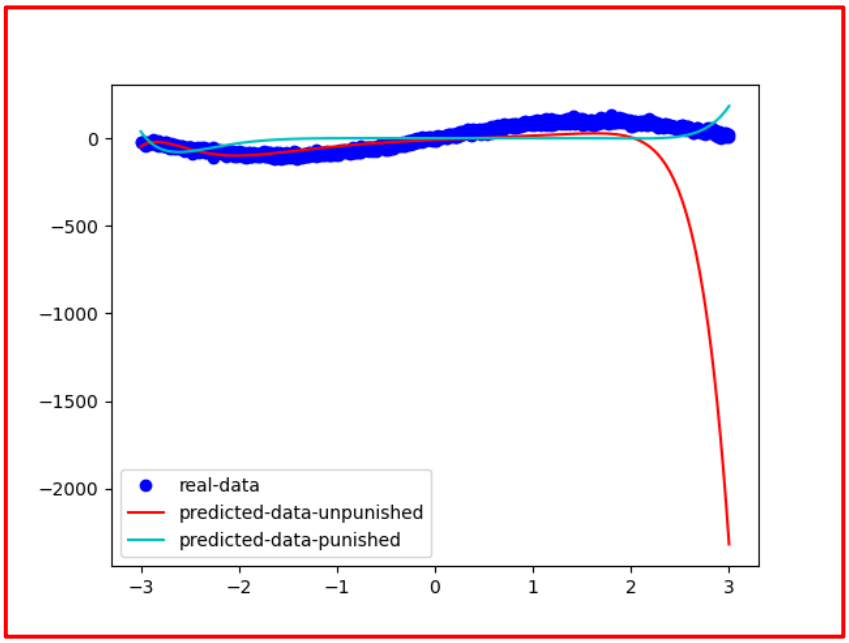
另一点在于，如果我们扩大自变量的取值范围，那么在边界上的值将会很大影响了画图的便利性和有效性。

对于第一个问题，我们可以给每个参数一个独特的学习率，采用自适应梯度法（AdaGrad）对每一个学习率进行控制。 但是自适应梯度法学习率下降过快，于是我们最后采用均方根传播法（RMSProp），来减小学习率下降的速度，加快拟合的速度，下面给出具体的公式:

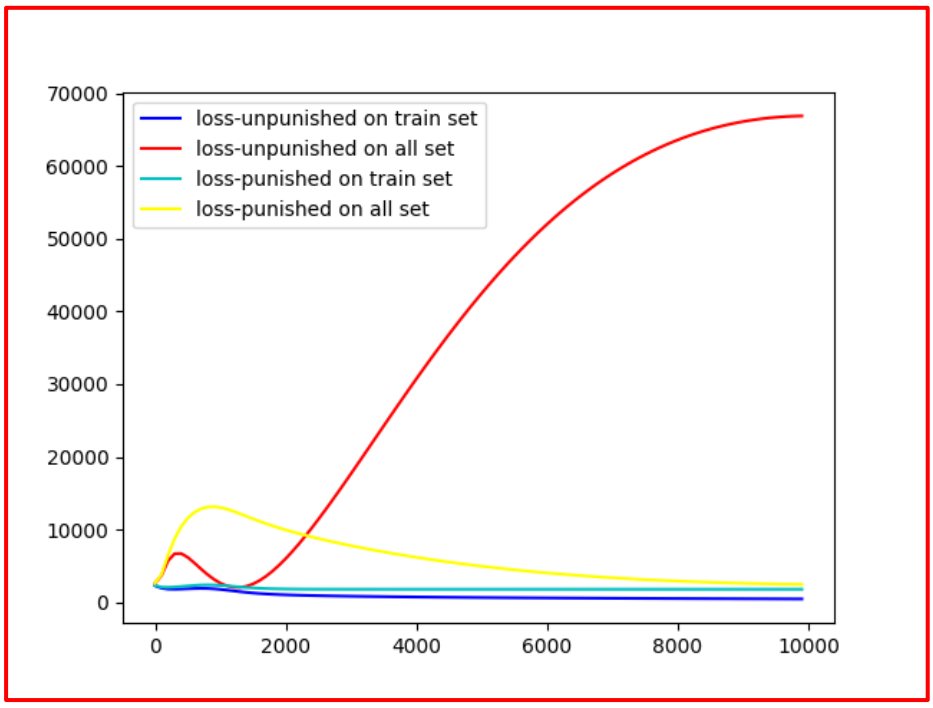
$$\begin{cases} G_t = \rho * \sum_{\tau=1}^{t-1} g_\tau^2 + (1-\rho) * g_t^2 \\ W_t = W_{t-1} - \dfrac{\alpha}{\left(\sqrt{G_t} + \epsilon\right)} * g_t \end{cases} \tag{8}$$

对于第二个问题，我们可以限制自变量的范围，调整正弦函数的参数，也可以达到相似的效果，同时在全局上拟合的表现也会更好。

4.4 实验结果展示



上图是实验中数据点未采用惩罚项获得的模型以及采用惩罚项获得的模型对比示意图。其中我们选取的训练数据集是区间[-3, 1]上的样本点，从上图可以看出，两个模型对这部分的处理都较好，并且没有使用惩罚项的模型表现更加。但是在区间[1, 3]上，没有使用 惩罚项的模型表现就不优秀了。



上图是训练过程中记录的损失。红线和黄线分别是不使用惩罚项的模型在全局上的损失和使用惩罚项的模型在全局上的损失。可以看出，不使用惩罚项的模型在训练的过程中出现了过拟合的现象，导致其损失变大很快。而使用了惩罚项的模型在全局上损失不会随

着迭代次数的增加变大很多。这说明增加惩罚项对于克服过拟合具有很好的效果。

5. 实验总体结论

在局部上，采用多项式函数拟合正弦函数表现出较好的效果，但会存在过拟合等问题。可以通过增加惩罚项来抑制过拟合现象。另外，在用多项式函数拟合时，可以尝试使用自适应梯度法和均方根传播法来控制不同项系数的训练速度。

6. 完整实验代码

```
import numpy as np
import matplotlib.pyplot as plt

cont = 800  # 样本点数量
a = 100  # 欲拟合函数参数 1
w = 1  # 欲拟合函数参数 2
m = 10  # 拟合使用的多项式阶数
Epoch = 10000  # 迭代次数
alpha = 1  # 学习率的初值
epsilon = 1  # 学习率的参数
flag = 0  # 0 表示不使用惩罚项 1 表示使用惩罚项
lamb = 1000  # 惩罚项的系数
rho = 0.2  # 累计梯度的参数


def get_data(a, w, cont, m):
    x = np.linspace(-3, 3, cont)
    y = a * np.sin(w*x) + np.random.normal(0, 10, cont)
    input_x = x[:int(cont*2/3)]
    input_y = y[:int(cont*2/3)]
    # input_x = x
    # input_y = y
    W = np.zeros(m+1)
    return x, y, input_x, input_y, W


def predict(x, W):
    y_hat = x.dot(W)
    return y_hat


def get_loss(y, y_hat, W, flag):
    loss = 0
    for i in range(len(y)):
        loss += (y[i]-y_hat[i])**2
    loss /= len(y)*2
    if flag == 1:
        for i in range(len(W)):
```

```
            loss += lamb * (W[i]**2)
    # print(loss)
    return loss


def train(input_x, input_y, x, y, W):
    # preparation
    X1 = np.ones((len(input_x), 1))
    X2 = np.ones((len(x), 1))
    tmp1 = np.ones(len(input_x))
    tmp2 = np.ones(len(x))
    g = np.zeros(len(W))
    G = np.zeros(len(W))
    eta = np.zeros(len(W))

    for i in range(len(W)-1):
        tmp1 = tmp1 * input_x
        X1 = np.hstack((X1, tmp1.reshape(len(tmp1), 1)))
        tmp2 = tmp2 * x
        X2 = np.hstack((X2, tmp2.reshape(len(tmp2), 1)))
    # print(X)

    train_losses = []
    all_losses = []
    loss_x = []

    # train
    for epoch in range(Epoch):
        y_hat1 = predict(X1, W)
        loss1 = get_loss(input_y, y_hat1, W, flag)
        y_hat2 = predict(X2, W)
        loss2 = get_loss(y, y_hat2, W, flag)
        if flag == 0:
            G = G - (1-rho)*g**2 + rho*g**2
            g = (X1.T.dot(y_hat1-input_y))/len(input_y)
            G = G + (1-rho)*g**2
            eta = alpha/(np.sqrt(G)+epsilon)
            W = W - eta * g
        else:
            G = G - (1 - rho) * g ** 2 + rho * g ** 2
            g = (X1.T.dot(y_hat1-input_y))/len(input_y) + 2*lamb*W
            G = G + (1-rho)*g**2
            eta = alpha/(np.sqrt(G)+epsilon)
            W = W - eta * g
        if epoch % 100 == 0:
            print(W)
            print("loss={}".format(loss1))
            train_losses.append(loss1)
            all_losses.append(loss2)
            loss_x.append(epoch)

    return W, train_losses, all_losses, loss_x


if __name__ == "__main__":
    x, y, input_x, input_y, W = get_data(a, w, cont, m)
```

```
    W1, train_losses1, all_losses1, loss_x1 = train(input_x, input_y,
x, y, W)

    X = np.ones((len(x), 1))
    tmp = np.ones(len(x))

    for i in range(len(W)-1):
        tmp = tmp * x
        X = np.hstack((X, tmp.reshape(len(tmp), 1)))

    y_hat1 = predict(X, W1)

    flag = 1
    W2, train_losses2, all_losses2, loss_x2 = train(input_x, input_y,
x, y, W)
    y_hat2 = predict(X, W2)

    print("W1 = {}".format(W1))
    print("W2 = {}".format(W2))

    loss1 = get_loss(y, y_hat1, W, 0)
    loss2 = get_loss(y, y_hat2, W, 0)
    print("loss1={}".format(loss1))
    print("loss2={}".format(loss2))
    plt.plot(x, y, 'bo', label='real-data')
    plt.plot(x, y_hat1, 'r', label='predicted-data-unpunished')
    plt.plot(x, y_hat2, 'c', label='predicted-data-punished')
    plt.legend()
    plt.show()

    plt.plot(loss_x1, train_losses1, 'b', label='loss-unpunished on
train set')
    # plt.plot(loss_x1, all_losses1, 'r', label='loss-unpunished on
all set')
    plt.plot(loss_x2, train_losses2, 'c', label='loss-punished on
train set')
    # plt.plot(loss_x2, all_losses2, 'yellow', label='loss-punished on
all set')
    plt.legend()
    plt.show()
```

7. 参考文献

[1]《深度学习基础》高等教育出版社 刘远超