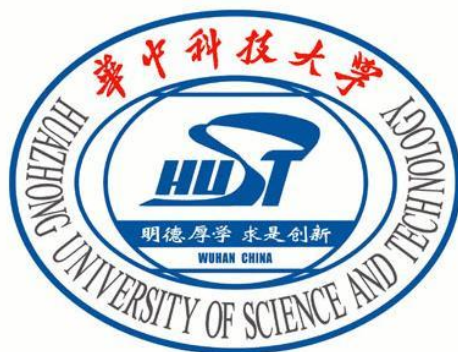


华中科技大学计算机科学与技术学院


机器学习报告



专 业： 计算机科学与技术

班 级： _____

学 号： _____

姓 名： 

成 绩： _____

指导教师： _____

完成日期： 2020 年 06 月 20 日

结课项目-收入预测

一、实验题目：收入预测

给定训练集 income.csv，要求根据每个人的属性值来判断此人年收入是否大于 50k。

二、实验要求

使用 Logistic 回归实现二分类，实现要求如下：

1. 实现语言要求为 python;
2. 使用梯度下降法时，调整学习率的固定值，探究不同学习率的选择对训练误差收敛速度的影响，绘制 misclassification rate 曲线进行比较并分析。
3. 加入正则项，并调整多个正则化参数后探究正则化参数对于训练效果的影响。

三、算法设计

1. 数据预处理

先采用非线性变换，再进行最大最小归一化，最后进行标准归一化。非线性变换见下式

$$x' = (\tan x)^{\frac{2}{\pi}}$$

2. Logistic 回归

sigmoid 函数作为激活函数的线性回归，根据交叉熵损失利用梯度下降法寻得损失最小处的权重值。

3. 动态学习率

大体思想是希望学习率随着迭代过程逐渐减小。

(1) 阶梯指数衰减

$$w_t = w_{t-1} - \alpha \cdot d^{\lfloor \frac{epoch}{\beta} \rfloor} \cdot m_t$$

(2) 连续指数衰减

$$w_t = w_{t-1} - \alpha \cdot d^{epoch} \cdot m_t$$

(3) 梯度模的平方和衰减

$$w_t = w_{t-1} - \frac{\alpha}{\sum_1^t (gradient \cdot gradient^T)} \cdot m_t$$

(4) 根号衰减

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{epoch}} \cdot gradient$$

4. 梯度下降

(1) Vanilla

固定学习率，选取整个样本计算当前梯度，进行“朴素”梯度下降，即

$$w_t = w_{t-1} - \alpha \cdot gradient$$

(2) Momentum

为了抑制固定学习率导致的震荡，可引入一阶动量，当前动量由累积动量和当前梯度按照一定权重求和得到，即

$$m_t = \beta \cdot m_{t-1} + (1 - \beta) \cdot gradient$$

$$w_t = w_{t-1} - \alpha \cdot m_t$$

(3) Nesterov

为了避免得到的是局部最优解，在一阶动量的基础上再引入一步“试探”，即在累积动量下“向前走一步”，计算这个时候的梯度，再利用这个梯度去更新动量。

(4) AdaGrad

对于经常更新的参数，我们已经积累了大量关于它的知识，不希望被单个样本影响太大，希望学习速率慢一些；对于偶尔更新的参数，我们了解的信息太少，希望能从每个偶然出现的样本身上多学一些，即学习速率大一些。为了量度历史更新的频率，可以引入“二阶动量”的概念，即在该维度上，迄今为止所有梯度值的平方和，即

$$v_t = \sum_{\tau=1}^t gradient_{\tau}^2$$

$$w_t = w_{t-1} - \frac{\alpha}{v_t} \cdot gradient$$

(5) AdaDelta

AdaGrad 存在明显的缺陷，即随着迭代次数的增多，二阶动量越来越大导致更新的幅度非常小。为了解决这个问题，AdaDelta 的改进之处在于只取一段窗口期内的梯度的平方和，且用指数移动平均代替过去一段时间内的平均值，即

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot gradient_t^2$$

$$w_t = w_{t-1} - \frac{\alpha}{v_t} \cdot gradient$$

(6) Adam

Adam 方法结合了 Momentum 方法和 AdaDelta 方法，结合一阶动量和二阶动量来更新权重，即

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot gradient$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot gradient_t^2$$

$$w_t = w_{t-1} - \frac{\alpha}{v_t} \cdot m_t$$

(7) Nadam

Nadam 在 Adam 的基础上引入了 Nesterov 的思想，即先行更新一次权重，再根据此时的梯度去更新一阶动量和二阶动量，最后重新计算权重值。

四、实验环境与平台

本实验使用的语言为 python，版本为 python3.7，实现的开发工具是 Pycharm 2020.1.1，系统为 Windows 10，CPU 为 i7-8550U，内存 8G，其后的测试也均在此软硬件环境下。

五、程序实现

```
import math
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
```



```

class LinearLogicRegression:
    def __init__(self, epoches=1000, alpha=0.01002, beta1=0.9, beta2=0.9, mode="SGD",
penalty=None, lbd=0.00001, silent=True, graph=True):
        """
        :param epoches: the number of rounds
        :param alpha: learning rate
        :param beta1: moment parameter(Moment, Nesterov, Adam, Nadam)
        :param beta2: delta parameter(AdaDelta, Adam, Nadam)
        :param mode: "Vanilla", "Moment", "Nesterov", "AdaGrad", "AdaDelta", "Adam",
"Nadam", "StepDiminish", "ContinuousExpDiminish", "DotSumDiminish", "SqrtDiminish"
        :param penalty: "l1", "l2"
        :param lbd: lambda, the parameter for regularization
        :param silent: whether to print the updating process
        :param graph: whether to draw the loss function
        """
        self.epoches = epoches
        self.alpha = alpha
        self.beta1 = beta1
        self.beta2 = beta2
        self.mode = mode
        self.penalty = penalty
        self.lbd = lbd
        self.silent = silent
        self.graph = graph
        self.weights = None

    def sigmoid(self, x):
        return 1.0 / (1 + np.exp(-x))

    def data2matrix(self, X):

        return np.array([[1.0, *d] for d in X])

    def fit(self, X, X_test, y, y_test):
        # to add an column to plus b
        mat_train = self.data2matrix(X)
        mat_test = self.data2matrix(X_test)

        # initialization
        self.weights = np.random.uniform(-0.1, 0.1, (1, len(mat_train[0])))
        V = np.zeros((1, len(mat_train[0])))
        squareWeights = 0
        moment = np.zeros(self.weights.shape)
        if self.graph:

```

```

iterations = []
loss_training = []
loss_testing = []

# start iterations
for epoch in range(self.epochs):
    # Adaptive Learning Rate && update the weights
    if self.mode == "Vanilla":
        gradient = self.computeGradient(mat_train, y)
        self.weights -= self.alpha * gradient
    elif self.mode == "Momentum":
        gradient = self.computeGradient(mat_train, y)
        moment = self.beta1 * moment + (1 - self.beta1) * gradient
        self.weights -= self.alpha * moment
    elif self.mode == "Nesterov":
        weights = self.weights
        self.weights -= self.alpha * moment
        gradient = self.computeGradient(mat_train, y)
        moment = self.beta1 * moment + (1 - self.beta1) * gradient
        self.weights = weights - self.alpha * moment
    elif self.mode == "AdaGrad":
        gradient = self.computeGradient(mat_train, y)
        V += np.square(gradient)
        self.weights -= (self.alpha / np.sqrt(V)) * gradient
    elif self.mode == "AdaDelta":
        gradient = self.computeGradient(mat_train, y)
        V = self.beta2 * V + (1 - self.beta2) * np.square(gradient)
        self.weights -= (self.alpha / np.sqrt(V)) * gradient
    elif self.mode == "Adam":
        gradient = self.computeGradient(mat_train, y)
        moment = self.beta1 * moment + (1 - self.beta1) * gradient
        V = self.beta2 * V + (1 - self.beta2) * np.square(gradient)
        self.weights -= (self.alpha / np.sqrt(V)) * moment
    elif self.mode == "Nadam":
        weights = self.weights
        self.weights -= (self.alpha / np.sqrt(V + 1)) * moment
        gradient = self.computeGradient(mat_train, y)
        moment = self.beta1 * moment + (1 - self.beta1) * gradient
        V = self.beta2 * V + (1 - self.beta2) * np.square(gradient)
        self.weights = weights - (self.alpha / np.sqrt(V)) * moment
    elif self.mode == "SqrtDiminish":
        gradient = self.computeGradient(mat_train, y)
        self.weights -= (self.alpha / math.sqrt(epoch + 1)) * gradient
    elif self.mode == "StepDiminish":

```

```

        if epoch % 100 == 0:
            self.alpha *= 0.8
            gradient = self.computeGradient(mat_train, y)
            self.weights -= (self.alpha / math.sqrt(epoch + 1)) * gradient
        elif self.mode == "ContinuousExpDiminish":
            self.alpha *= 0.99
            gradient = self.computeGradient(mat_train, y)
            self.weights -= (self.alpha / math.sqrt(epoch + 1)) * gradient
        elif self.mode == "DotSumDiminish":
            gradient = self.computeGradient(mat_train, y)
            squareWeights += np.dot(gradient, gradient.T)
            self.weights -= (self.alpha / math.sqrt(squareWeights)) * gradient
    else:
        print("Error: No such mode!")

    # document the data
    if self.graph:
        iterations.append(epoch)
        loss_training.append(self.computeEntropyLoss(mat_train, y))
        loss_testing.append(self.computeEntropyLoss(mat_test, y_test))

    # output the process
    if not self.silent:
        print('Round {}: \n'.format(epoch), self.weights)

    # plot the loss graph
    if self.graph:
        plt.plot(iterations, loss_training, label='training loss')
        plt.plot(iterations, loss_testing, label='test loss')
        plt.xlabel('epoches')
        plt.ylabel('loss')
        plt.legend()
        plt.show()

def computeGradient(self, x, y):
    """
    :param x: data
    :param y: labels
    :return: gradient
    """
    # compute gradient
    z = self.sigmoid(np.dot(x, self.weights.T))
    gradient = -np.sum(np.dot(x.T, (y - z)), axis=1)

```

```

# add different kind of regularization
if self.penalty == "l2":
    gradient += self.lbd * self.weights[0, :]
elif self.penalty == "l1":
    gradient += self.lbd * np.array(list(map(lambda x: 1 if x > 0 else -1,
self.weights[0, :])))

```

```

return gradient

```

```

def computeEntropyLoss(self, mat, y):

```

```

    """

```

```

    :param mat: transformed data

```

```

    :param y: labels

```

```

    :return: Cross Entropy

```

```

    """

```

```

    calculation = self.sigmoid(np.dot(mat, self.weights.T))

```

```

    one_minus_calculation = np.where(calculation==1, 1, 1 - calculation)

```

```

    calculation = np.where(calculation==0, 1, calculation)

```

```

    one_minus_calculation = np.log(one_minus_calculation)

```

```

    calculation = np.log(calculation)

```

```

    loss = np.mean(np.where(y, - calculation, - one_minus_calculation))

```

```

    return loss

```

```

def score(self, X_test, y_test):

```

```

    """

```

```

    :param X_test: test data

```

```

    :param y_test: test labels

```

```

    :return: accuracy

```

```

    """

```

```

    X_test = self.data2matrix(X_test)

```

```

    calculation = np.dot(X_test, self.weights.T)

```

```

    prediction = np.where(calculation < 0.5, 0, 1)

```

```

    correct = np.equal(prediction, y_test)

```

```

    accuracy = np.mean(correct)

```

```

    return accuracy

```

```

def FindBestAlpha(self, X, X_test, y, y_test, start, end, step):

```

```

    """

```

```

    :param X: training data

```

```

    :param X_test: test data

```

```

    :param y: training labels

```

```

    :param y_test: test labels

```

```

    :param start: the start point of alpha

```

```

    :param end: the end point of alpha

```



```
:param step: the change of alpha of each round  
:return: the best alpha with the highest accuracy  
"""
```

```
alphas = []  
mis = []  
self.alpha = start  
while self.alpha <= end:  
    self.fit(X, X_test, y, y_test)  
    alphas.append(self.alpha)  
    mis.append(1 - self.score(X_test, y_test))  
    self.alpha += step  
# output the best lambda  
print(mis.index(min(mis)), 1 - min(mis))  
# plot the data  
plt.plot(alphas, mis)  
plt.xlabel('learning rate')  
plt.ylabel('misclassification rate')  
plt.show()
```

```
def FindBestLambda(self, X, X_test, y, y_test, start, end, step):  
    """
```

```
:param X: training data  
:param X_test: test data  
:param y: training labels  
:param y_test: test labels  
:param start: the start point of lambda  
:param end: the end point of lambda  
:param step: the change of lambda of each round  
:return: the best parameter lambda with the highest accuracy  
"""
```

```
lbds = []  
mis = []  
self.lbd = start  
while self.lbd <= end:  
    self.fit(X, X_test, y, y_test)  
    lbds.append(self.lbd)  
    mis.append(1 - self.score(test_data, test_labels))  
    self.lbd += step  
# output the best lambda  
print(mis.index(min(mis)), 1 - min(mis))  
# plot the data  
plt.plot(lbds, mis)  
plt.xlabel('lambda')
```

```

plt.ylabel('misclassification rate')
plt.show()

if __name__ == '__main__':
    # parameters
    filename =
    r'C:/Users/Jovy/OneDrive/Desktop/MyFile/Desk/Engineer/CS/python_project/income-
    KNN/income.csv'
    training_size = 3000
    total_size = 4000

    # read the .csv file
    with open(filename, encoding='utf-8') as f:
        all_data = np.loadtxt(f, delimiter=',')

    # preprocessing:  $x' = (\tan x)^{(2 / \pi)}$  -> MinMaxScaler -> StandardScaler
    data = all_data[:, 1:58]
    data = np.array(data)
    data = np.arctan(data) ** (2 / np.pi)
    mms = MinMaxScaler()
    data = mms.fit_transform(data)
    stdsc = StandardScaler()
    data = stdsc.fit_transform(data)

    # divide the data: 3000 for training and 1000 for testing
    training_data = data[:training_size, :]
    training_labels = all_data[:training_size, 58:59]
    test_data = data[training_size:total_size, :]
    test_labels = all_data[training_size:total_size, 58:59]

    # set hyper-parameters and start training
    clf = LinearLogicRegression(epochs=1000, alpha=.0036, penalty="l2", mode="Vanilla",
    lbd=3, silent=True, graph=True)
    clf.fit(training_data, test_data, training_labels, test_labels)

    # test the result
    print(clf.score(test_data, test_labels))

```

六、实验结果

1. 正确率测试

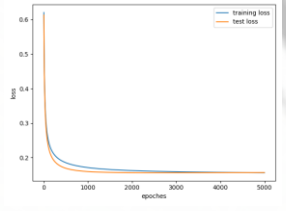
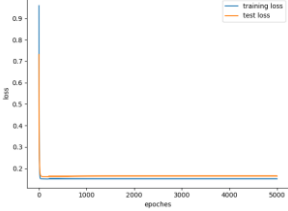
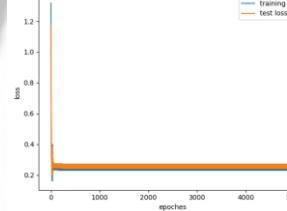
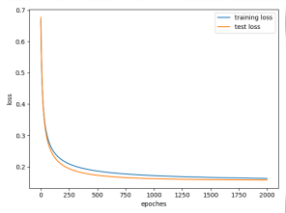
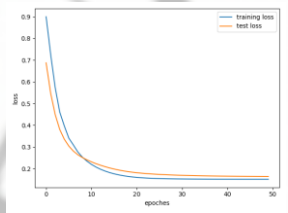
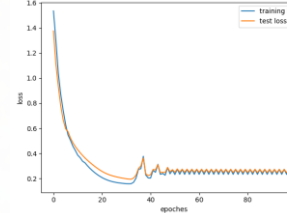
数据经过非线性变换之后，在合适的学习率下，各种方法的正确率在 94.4%

上下波动，最高正确率为 95.1%。

2. 探究不同学习率对于收敛速度和收敛效果的影响（Vanilla 梯度下降）

（1）取不同学习率下的损失函数曲线见表 6-1。

表 6-1 不同学习率下收敛速度和收敛效果的对比

学习率	0.00001	0.004285	0.008
正确率	94.4%	94.6%	91%
损失函数曲线			
损失函数曲线局部放大图			
迭代多少次之后收敛	大约 1500 次	大约 30 次	大约 50 次
是否出现震荡	否	否	是

（1）从 0.000001 至 0.01 每隔 0.000001 取一个学习率，并进行测试（无正则化，5000 次迭代），“错误率-学习率”曲线见图 6-1。

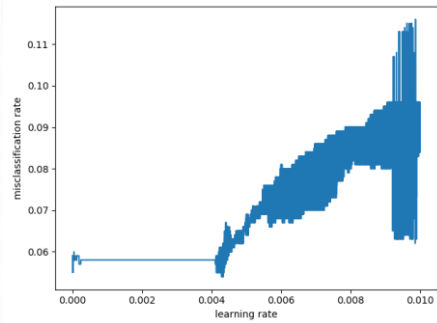


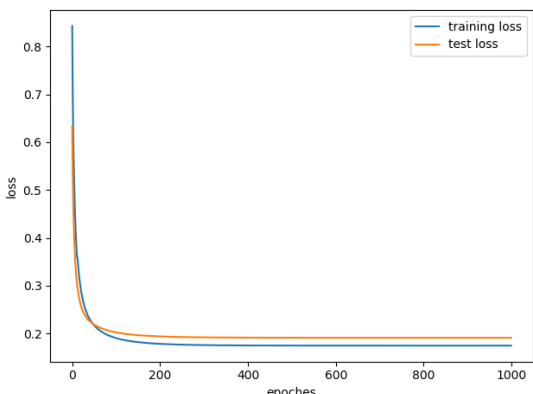
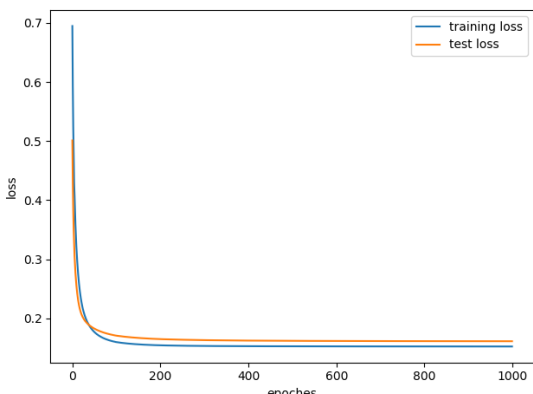
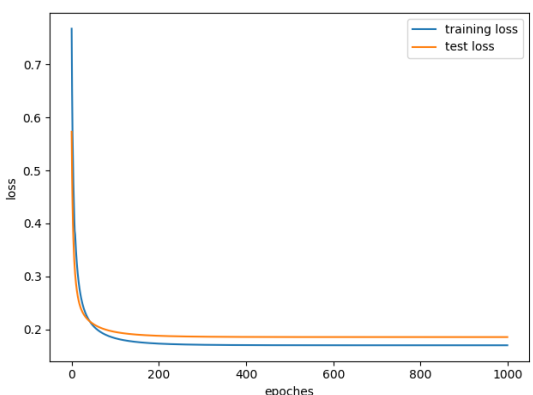
图 6-1 学习率-错误率曲线

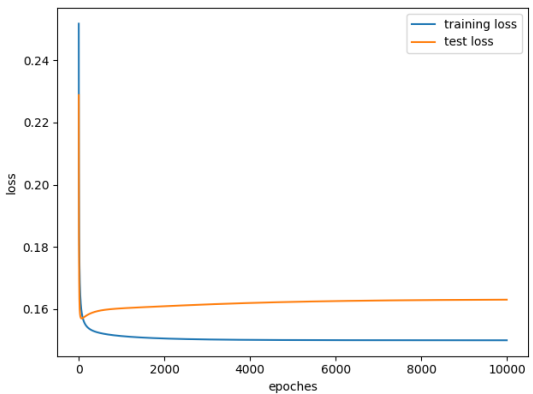
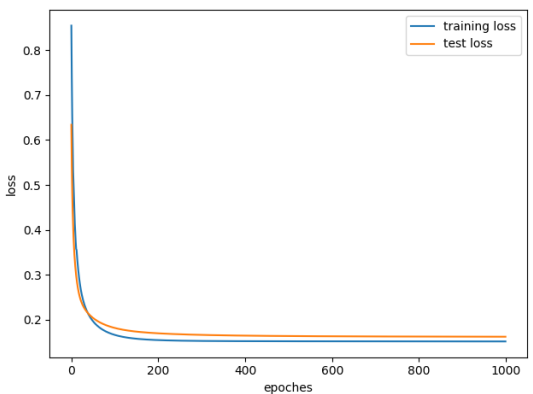
可以看到 0.004 附近是比较合适的学习率，在这个范围内的学习率下，既不会出现下降速度过慢，也不会出现震荡，是收敛速度和收敛效果的一个平衡点。

3. 探究动态调整学习率的不同方法对于收敛速度和收敛效果的影响

表 6-2 动态调整学习率不同方法下的收敛速度和收敛效果的对比

(学习率为 0.0036，迭代次数为 1000，无正则化)

调整方法	正确率	损失函数曲线
不调整	93.8%	
阶梯指数衰减	93.9%	
连续指数衰减	94.2%	

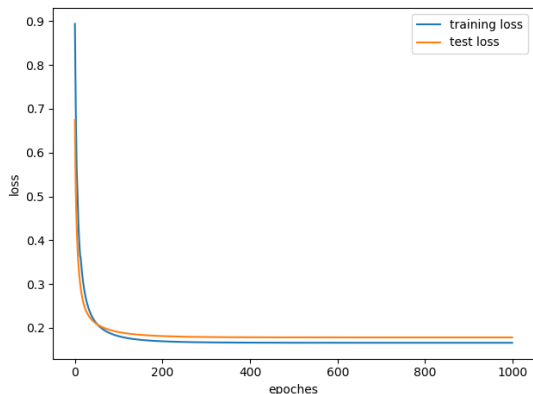
梯度模的平方和衰减	94.2% ($\alpha=1$)	
根号衰减	94.2%	

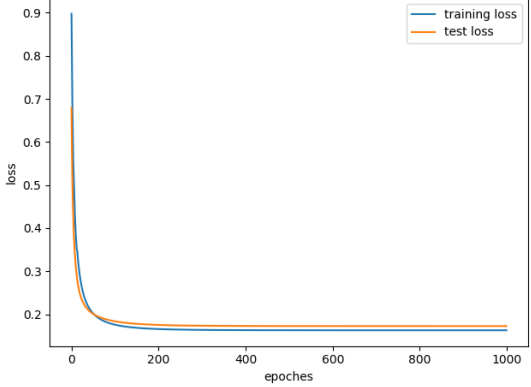
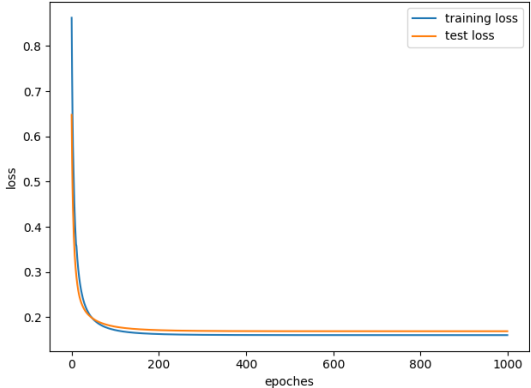
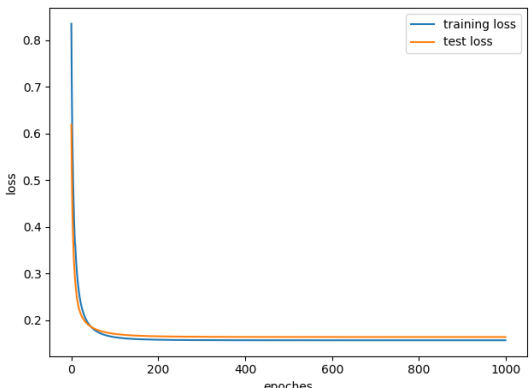
可以看到，不同的动态调整学习率的方法效果略有差异，但在大多数情况下能够使正确率在一定程度上得到提升。其原因是，学习率随着迭代次数的增加而衰减，使得能进行“微调”，从而达到更好的收敛效果。

4. 探究不同正则化参数对于收敛效果的影响

(1) L1 正则 (初始学习率 0.0036 并以连续指数衰减且迭代次数为 1000 次)

表 6-3 不同 L1 正则化参数下收敛效果的对比

正则化参数	正确率	损失函数曲线
$\lambda=0$	93.8%	
$\lambda=3$	93.8%	

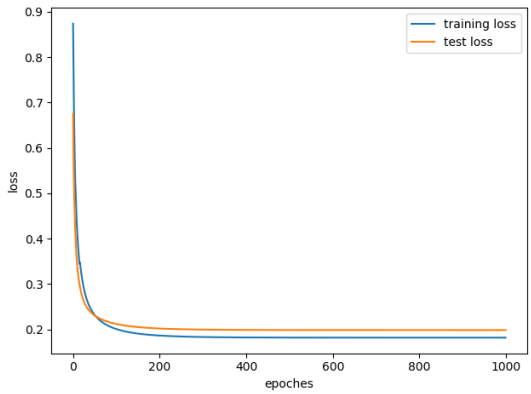
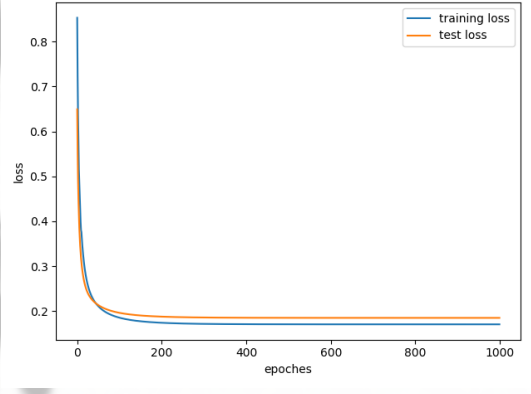
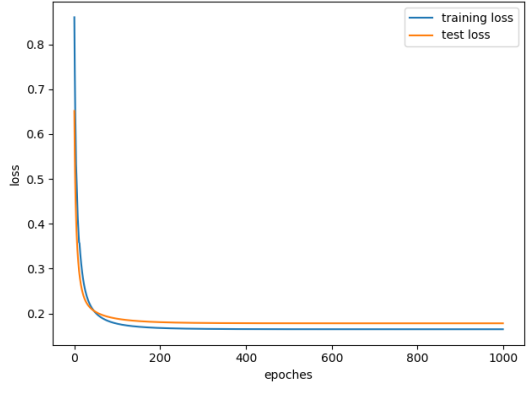
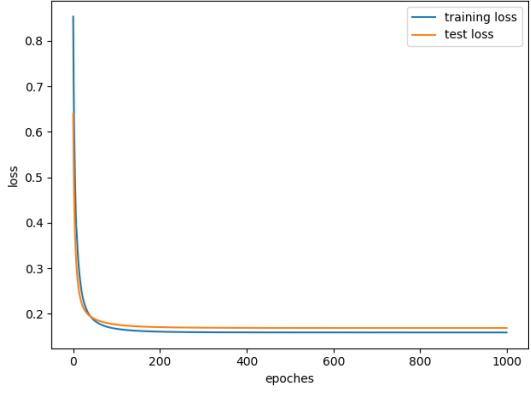
$\lambda=4$	94.2%	
$\lambda=5$	94.0%	
$\lambda=6$	93.9%	

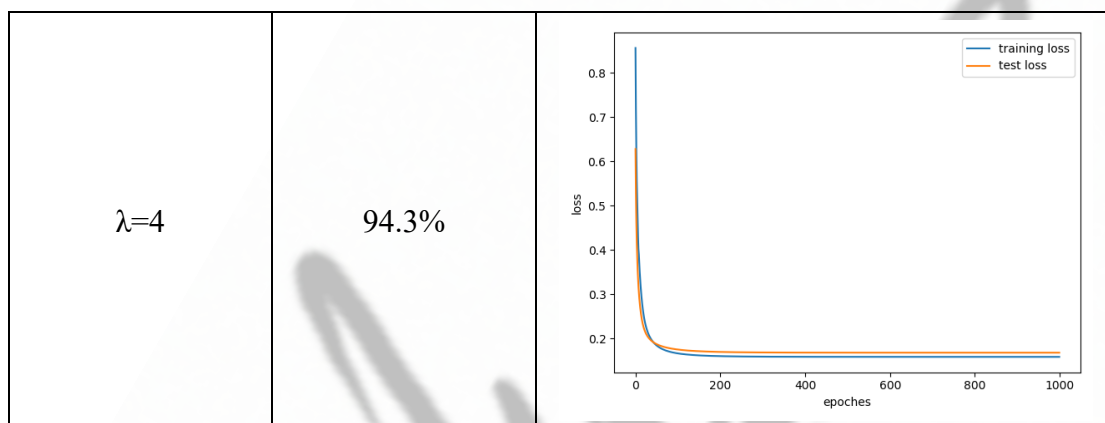
可以看到，随着 λ 正则化参数的增大，训练样本和测试样本的损失函数曲线的收敛值之差逐渐减小，而正确率是先逐渐增大后逐渐减小。正确率出现这样的变化规律的原因是，当正则化参数过小时，仍然存在过拟合的问题；当正则化参数过大，反而会“拟合不足”，因而应该存在一个平衡的 λ 正则化参数使得正确率最大。根据以上实验结果，这个最优的 λ 正则化参数应该在 $\lambda=4$ 附近。

(2) $\lambda=0.0036$ 并以连续指数衰减且迭代次数为 1000 次)

表 6-4 不同 λ 正则化参数下收敛效果的对比

正则化参数	正确率	损失函数曲线
-------	-----	--------

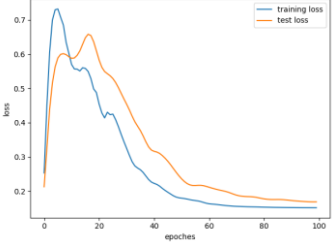
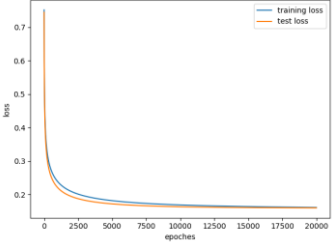
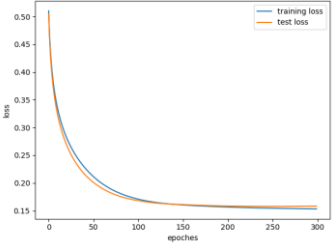
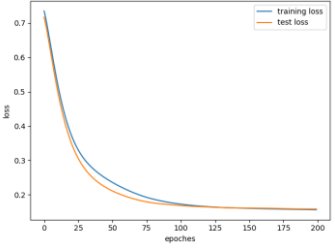
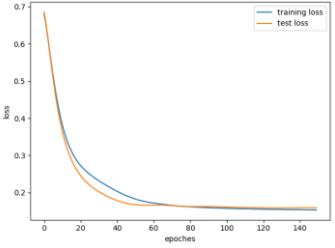
$\lambda=0$	94.2%	 <p>Line graph showing training loss (blue) and test loss (orange) over 1000 epochs for $\lambda=0$. The y-axis is labeled 'loss' and ranges from 0.2 to 0.9. The x-axis is labeled 'epoques' and ranges from 0 to 1000. Both losses decrease rapidly and plateau around 0.2.</p>
$\lambda=1$	94.3%	 <p>Line graph showing training loss (blue) and test loss (orange) over 1000 epochs for $\lambda=1$. The y-axis is labeled 'loss' and ranges from 0.2 to 0.8. The x-axis is labeled 'epoques' and ranges from 0 to 1000. Both losses decrease rapidly and plateau around 0.2.</p>
$\lambda=2$	94.6%	 <p>Line graph showing training loss (blue) and test loss (orange) over 1000 epochs for $\lambda=2$. The y-axis is labeled 'loss' and ranges from 0.2 to 0.8. The x-axis is labeled 'epoques' and ranges from 0 to 1000. Both losses decrease rapidly and plateau around 0.2.</p>
$\lambda=3$	94.5%	 <p>Line graph showing training loss (blue) and test loss (orange) over 1000 epochs for $\lambda=3$. The y-axis is labeled 'loss' and ranges from 0.2 to 0.8. The x-axis is labeled 'epoques' and ranges from 0 to 1000. Both losses decrease rapidly and plateau around 0.2.</p>



可以看到， $\lambda=4$ 正则化参数对于收敛效果的影响与 $\lambda=2$ 正则化参数类似，即随着 $\lambda=4$ 正则化参数的增大，训练样本和测试样本的损失函数曲线的收敛值之差逐渐减小，而正确率也是先逐渐增大后逐渐减小。正确率出现这样的变化规律的原因是，当正则化参数过小时，仍然存在过拟合的问题；当正则化参数过大，反而会“拟合不足”，因而应该存在一个平衡的 $\lambda=4$ 正则化参数使得正确率最大。根据以上实验结果，这个最优的 $\lambda=4$ 正则化参数应该在 $\lambda=2$ 附近。

5. 探究不同梯度下降方法对于收敛速度和收敛效果的影响（无正则化）

方法名称	相 关 参 数	正确率	损失函数曲线	大约迭代多少次收敛	是否出现明显震荡
Vanilla	$\alpha=0.006$	92.1%		50	是
Momentum	$\alpha=0.006$ $\beta_1=0.9$	94.2%		100	否

Nesterov	$\alpha=0.006$ $\beta_1=0.9$	94.3%		100	否
AdaGrad	$\alpha=0.006$	94.2%		15000	否
AdaDelta	$\alpha=0.006$ $\beta_2=0.9$	94.3%		300	否
Adam	$\alpha=0.006$ $\beta_1=0.9$ $\beta_2=0.9$	94.3%		150	否
Nadam	$\alpha=0.006$ $\beta_1=0.9$ $\beta_2=0.9$	94.3%		100	否

七、结果分析

本次实验着重关注了学习率对于收敛速度和收敛效果的影响，也学习了各种梯度下降方法的原理并进行了实现，也比较了各种梯度学习方法。具体结果分析如下：

1. 学习率对于收敛速度和收敛效果的影响

(1) 学习率对于梯度下降是非常重要的一个参数，它对于收敛速度和收敛效果起着决定性的作用；

(2) 当学习率过大时，可能会在最优解处来回晃动，也可能出现剧烈震荡，甚至会出现不收敛，从而导致学习失败；

(3) 当学习率过小时，收敛速度会比较慢；

(4) 渐变学习率希望在以一个较大的学习率开始，而随着迭代过程渐渐减小，使得在趋近收敛时能以“微调”的方式尽可能地趋近最优解。渐变学习率的衰减方法有很多，大多数效果相近。一般而言，渐变学习率最后的正确率相较于固定学习率略有增加，但相差无几，不过其收敛速度更快；

3. 正则化参数对于收敛效果的影响

无论是 $L1$ 正则还是 $L2$ 正则，其参数对于收敛效果的影响是类似的。即随着正则化参数的增大，训练样本和测试样本的损失函数曲线的收敛值之差逐渐减小，而正确率是先逐渐增大后逐渐减小。正确率出现这样的变化规律的原因是，当正则化参数过小时，仍然存在过拟合的问题；当正则化参数过大，反而会“拟合不足”，因而应该存在一个平衡的 $L1$ 正则化参数使得正确率最大。可以根据实验结果找到这个最优正则化参数的范围。

2. 各种梯度学习方法脉络分析与异同比较

(1) 最“朴素”的方法就是 Vanilla 方法，选取整个样本以固定的学习率计算梯度并进行更新，但它的问题是可能下降速度慢，可能出现持续震荡，并停留在局部最优解，很难找到一个完美的学习率；

(2) 为了抑制震荡问题，引入 Momentum 的概念，把梯度平滑一下，它希望在“坡比较陡”时利用惯性下降地快一点，从实验效果而言，它确实可以在一定程度上抑制震荡的问题；

(3) 为了避免出现局部最优解，可以使用 Nesterov 加速，它希望“站得更高，看得更远”，从而在一定程度上避免出现局部最优解；

(4) AdaGrad 方法开启了“自适应学习率”的大门，对于更新越频繁的参数，我们越不希望它被单个样本影响太大，从而引入了二阶动量的概念，以达到“自适应”的效果。但 AdaGrad 方法存在一个问题，那就是随着迭代次数增多，二阶动量越来越大，从而导致迭代几乎没有效果；

(5) 为了解决 AdaGrad 的问题，采用了平滑处理的思想，即计算窗口时间内的二阶动量累积，这就是 AdaDelta 方法；

(6) 在 AdaDelta 中添加 Momentum 就得到了 Adam 方法。Adam 是上述集中方法的集大成者，但 Adam 方法也存在两大缺陷——可能不收敛^[1]以及可能错过全局最优解^[2]。如果想要解决这个问题，一个可行的思路是——先用 Adam 方法，经过一定迭代次数再回到 Vanilla^[3]；

(7) 在 Adam 基础上引入 Nesterov 加速就得到了 Nadam 方法，它在一定程度上可以避免错过全局最优解。

(8) 上述所有梯度下降方法除了在收敛速度和在最后的正确率上有区别以外，对于不同的梯度下降方法，下降方向也不同，这就导致了不同的梯度下降方法可能进入完全不同的局部最优点，且相关实验表明，各种梯度下降的方法所找到的解都有其各自的特征^[4]，也就是说不同的梯度下降方法并不是完全随机的，它有其固有的特征。

参考文献

- [1] Sashank J. Reddi, Satyen Kale, Sanjiv Kumar. On the Convergence of Adam and Beyond. ICLR 2018 Conference Blind Submission, 2018
- [2] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, Benjamin Recht. The Marginal Value of Adaptive Gradient Methods in Machine Learning. arXiv:1705.08292v2, 2018
- [3] Nitish Shirish Keskar, Richard Socher. Improving Generalization Performance by Switching from Adam to SGD. arXiv:1712.07628v1, 2017
- [4] Daniel Jiwoong Im, Michael Tao, Kristin Branson. An empirical analysis of the optimization of deep network loss surfaces. arXiv:1612.04010v4. 2017