Project Report of Machine Learning (Assignment 5)

Author: 赵文浩 23020201153860 (计算机科学系)

I. Multinomial Logistic Regression on MNIST Dataset

1.1 MNIST Dataset

本次课程设计使用 MNIST 手写图片数据集[1]，数据集中包含 10 类手写数字图片（如图 1-1 所示），同时数据集提供了每张图片对应的标签 0~9。本文将采用 Logistic Regression 模型对 MNIST 数据集中的图片分类并通过 Accuracy、Loss 和 AUC 等指标对模型训练的过程和分类效果进行分析。

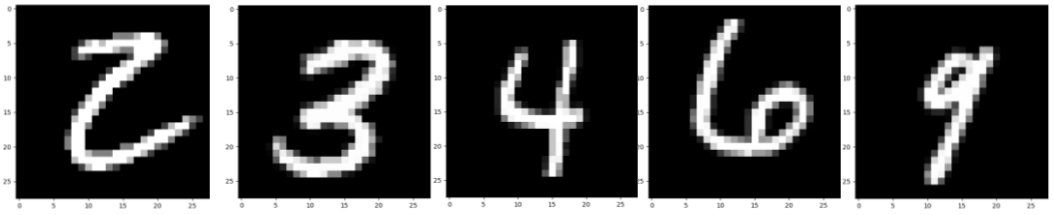


图 1-1 Samples of MNIST Dataset

1.2 Logistic Regression

Logistic Regression 是一种基于概率的线性回归分类模型，为了将模型输出与概率相关联，这里引入压缩函数从而实现原始数据到区间 $(0,1)$ 上的映射。压缩函数是 Logistic Regression 模型的重要组成部分，二分类问题使用 Sigmoid 作为压缩函数；多标签分类问题使用 Softmax 作为压缩函数，二者的本质相同，且 Softmax 是 Sigmoid 的推广。下面将从 ML=L+A+M+BD+A 的角度分别介绍 Logistic Regression 在二分类与多分类场景下的问题定义和求解过程。

1.3 Model

与 Linear Regression 类似，Logistic Regression 模型可表示为如下形式：

$$y(x,w) = w_1x_1 + w_2x_2 + \cdots + w_dx^d + b = \sum_{i=0}^{d} w_ix^i + b = Wx + b$$

二者的不同点在于，Logistic Regression 以概率为基础，根据不同分类场景，分别使用 Sigmoid 和 Softmax 压缩函数将模型输出映射到 $(0,1)$ 的概率区间内，二者的数学表达式如下：

$$sigmoid(y) = \frac{1}{1+e^{-y}} \quad , \quad softmax(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

因此二分类和多分类的 Logistic Regression 模型可进一步定义为：

(1) Binary $P\left(y \mid x,W,b\right)=sigmoid\left(Wx+b\right)=\left(\frac{1}{1+e^{-(Wx+b)}}\right)^{y}\times\left(1-\frac{1}{1+e^{-(Wx+b)}}\right)^{1-y}$

(2) Multinomial $P\left(Y=i \mid x,W,b\right)=softmax_{i}\left(Wx+b\right)=\frac{e^{W_{i}x+b_{i}}}{\sum_{j}e^{W_{j}x+b_{j}}}$

Softmax 函数实际上是 Sigmoid 函数的推广，因此本文主要对多分类问题进行详细分析。采用 Softmax 压缩函数将模型输出映射到概率区间后，可根据模型属于各个类别的概率大小将其划分为对应的标签，该过程的数学表达形式如下：

$$y_{pred}=argmax_{i}P\left(Y=i \mid x,W,b\right)$$

1.4 Loss Function

Section 1.3 介绍了 Logistic Regression 的概率表达形式，因此可以借助最大似然估计（MLE）求解在何参数下，所给定带标签的训练集出现的概率最大。似然函数 Likelihood 如下：

$$L\left(\theta=\{W,b\},D\right)=\sum_{i=0}^{|D|}\log\left(P\left(Y=y_{i} \mid x_{i},W,b\right)\right)$$

对应的损失函数 Loss 可表示为如下形式：

$$\ell\left(\theta=\{W,b\},D\right)=-L\left(\theta=\{W,b\},D\right)=-\sum_{i=0}^{|D|}\log\left(P\left(Y=y_{i} \mid x_{i},W,b\right)\right)$$

1.5 Algorithm

1.5.1 Gradient Descent

在定义了 Loss Function 后，需要求解损失函数最小的情况下对应的模型参数，此时应用求解得到的参数即可获得最终的预测模型。这里采用梯度下降 Gradient Descent 算法最小化 $\ell\left(\theta=\{W,b\},D\right)$，单样本损失函数梯度如下：

$$J\left(W\right)=\frac{\partial\ell_{i}}{\partial W_{ij}}=\begin{cases}\left(S_{i}-1\right)x_{j}, y=i\\ S_{i}x_{j}\quad\quad, y\neq i\end{cases}$$

$$J\left(b\right)=\frac{\partial\ell_{i}}{\partial b_{i}}=\begin{cases}S_{i}-1, y=i\\ S_{i}\quad\quad, y\neq i\end{cases}$$

其中 $S_{i}=softmax_{i}=softmax\left(Wx_{i}+b\right)$。

1.5.2 Different Update Policies for Gradient

求得损失函数ℓ的梯度后，可采用 MSGD 或其他方式逐步更新模型参数$\theta=\{W,b\}$，这一过程称为 Model Training。在 Gradient Descent 的基础上，本小节给出三种不同的模型参数更新策略——Normal GD、Noise GD 以及 Regularization GD，分别代表普通梯度下降、带有随机噪音项的梯度下降和引入正则项的梯度下降，三种策略下参数（权重和偏置项）的更新方式如下：

(1) Normal GD

$$w_k^{(t+1)} \leftarrow w_k^{(t)} - \eta \times \nabla_{w_k} J\left(w\right),\ k=1,2,\cdots,K-1$$

(2) Noise GD

$$w_k^{(t+1)} \leftarrow w_k^{(t)} - \eta \times \nabla_{w_k} J\left(w\right) + \varepsilon_k,\ \varepsilon_k \sim N\left(0,\sigma^2 I\right),\ k=1,2,\cdots,K-1$$

(3) Regularization GD

$$w_k^{(t+1)} \leftarrow w_k^{(t)} - \eta \times \left[\nabla_{w_k} J\left(w\right) + \lambda w_k^{(t)}\right],\ k=1,2,\cdots,K-1$$

1.6 Application and Experiment

上文从 ML=L+A+M+BD+A 的角度分别介绍了 Logistic Regression 的模型定义、损失函数和参数求解算法，本节将分别采用 Section 1.5.2 引入的三种模型参数更新策略对 MNIST 手写图片数据集进行分类。需要特别说明的是，考虑到算力不足的问题，本次课程设计未采用完整的 MNIST 数据集，而是分别从训练集和测试集中分别随机选取 7000 和 3000 个样本进行分类模型的训练和测试。

1.6.1 Adjustable Parameters

在采用 Gradient Descent 对分类模型进行训练的过程中，存在一些可调整的参数，这些参数将对最终的分类结果和训练性能产生一定的影响，表 1-1 展示了 5 个可调节的参数（P.S. ITERATION 由 EPOCHS 和 BATCH 共同决定）。

表 1-1 Adjustable Parameters in Logistic Regression Model Training on MNIST Dataset

ID	Parameter	Description	Our Selection
1	EPOCHS	训练集遍历次数	Adjustable
2	BATCH_SIZE	参数更新一次采用的训练样本数量	Adjustable
·	ITERATION	训练迭代次数	EPOCHS×BATCH_NUM
3	LEARNING_RATE	学习率·参数更新速率/步长	Adjustable
4	GRADIENT_TYPE	梯度下降时参数的更新策略	{normal, noise, regularization}
5	LAMBDA	Regularization GD 下的正则化因子	Adjustable

1.6.2 Result and Analysis

本文分别采用表 1-1 所示的训练参数，记录完整训练过程和最终分类结果，并利用 Accuracy、Loss 以及 ROC 曲线进行可视化展示和分析。

(1) Training with Different Learning Rate

本小节采用普通梯度下降法（Normal GD），并固定 epochs=1、batch-size=10，探究训练过程和分类结果随学习率 lr 的变化情况。

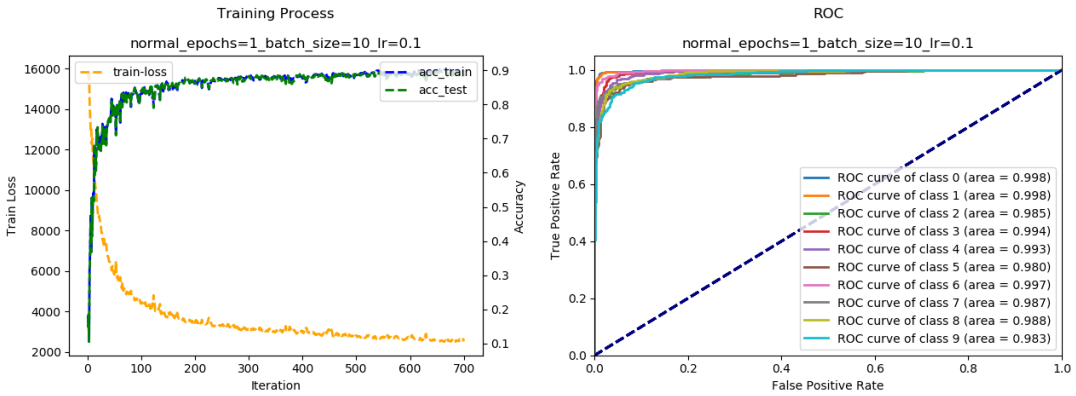


图 1-2 Result of Training with normal-GD, epochs=1, batch-size=10, **lr=0.1**

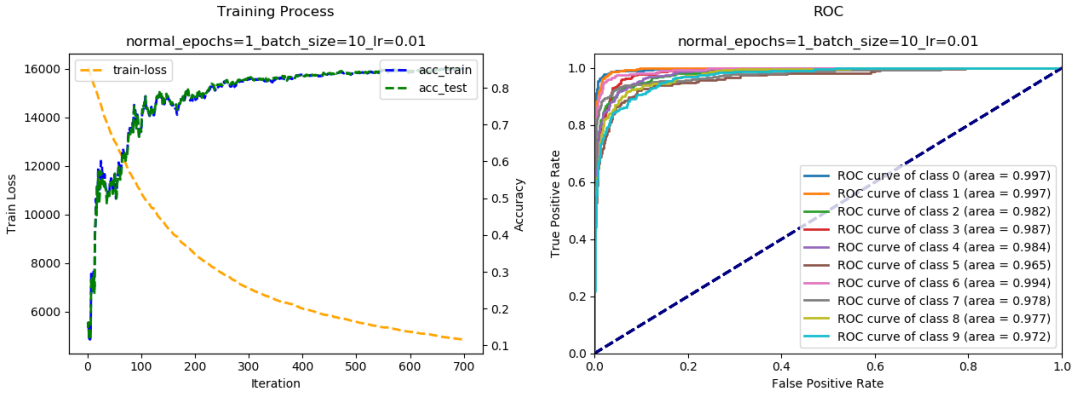


图 1-3 Result of Training with normal-GD, epochs=1, batch-size=10, **lr=0.01**

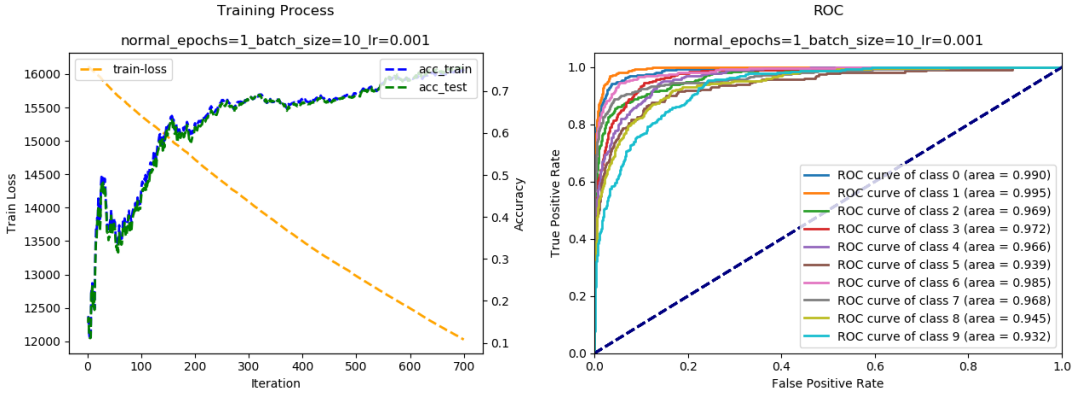


图 1-4 Result of Training with normal-GD, epochs=1, batch-size=10, **lr=0.001**

从上图可以看出，当其他训练参数相同时，模型准确率随学习率 lr 的降低而降低；同时模型在训练集上 loss 的降低速度也变得缓慢。这是由于较低的学习率降低了参数更新速度，应增大迭代次数使模型训练更充分。上文提到，训练迭代次数 iteration 与 epochs 和 batch-size 密切相关，且 iteration = epoch× N/batch_size，下面将进一步探究 epochs 和 batch-size 对训练过程和分类结果的影响。

(2) Training with Different Epochs

本小节仍然采用普通梯度下降法（Normal GD），并固定 batch-size=10，lr=0.001，探究训练过程和分类结果随 epochs 的变化情况。

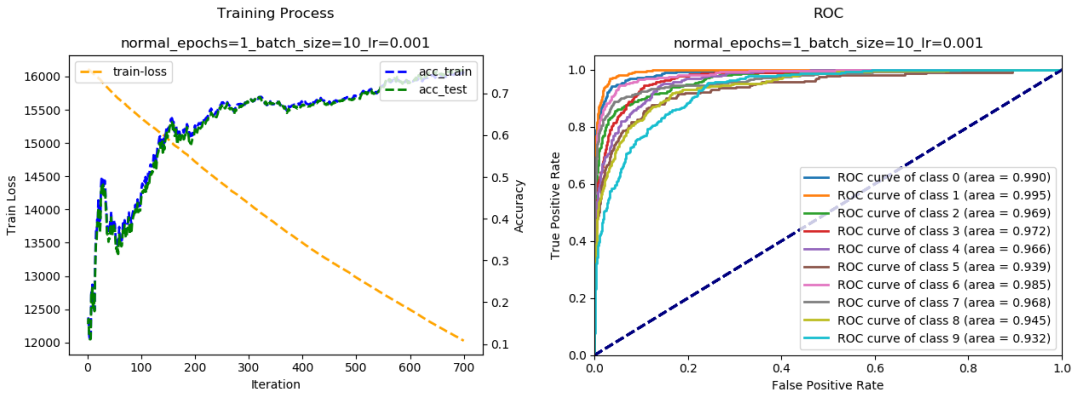


图 1-5 Result of Training with normal-GD, **epochs=1**, batch-size=10, lr=0.001

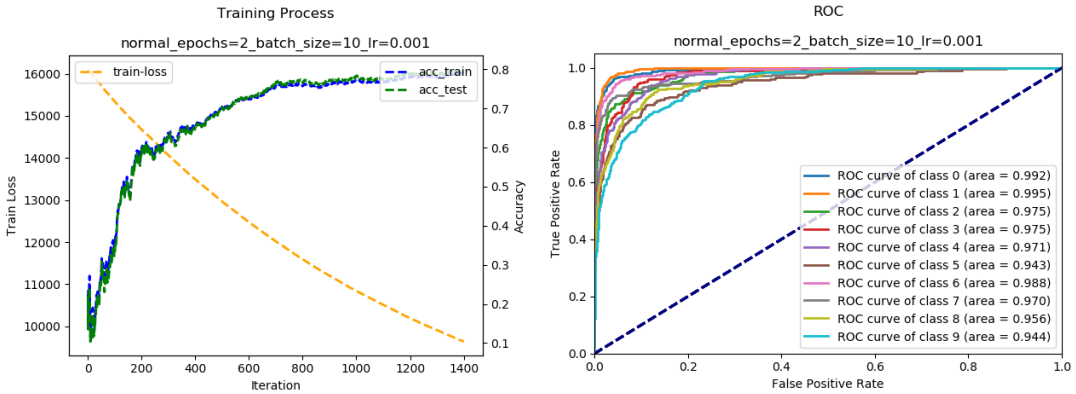


图 1-6 Result of Training with normal-GD, **epochs=2**, batch-size=10, lr=0.001

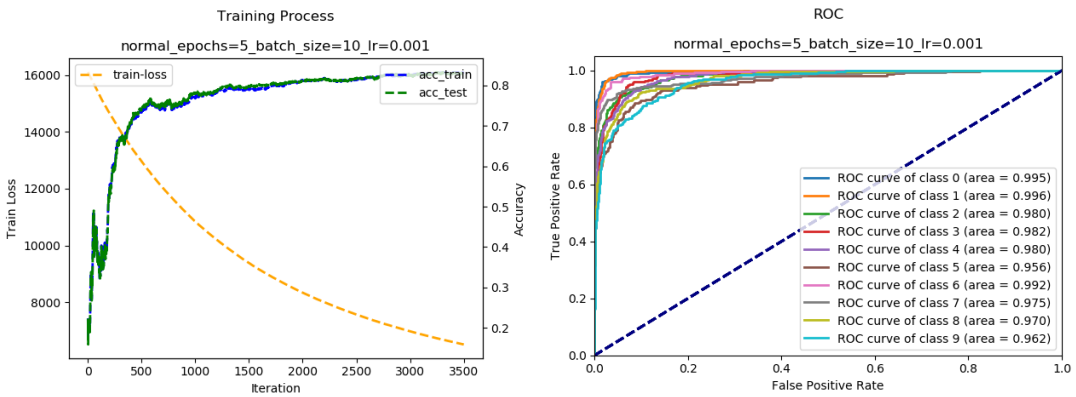


图 1-7 Result of Training with normal-GD, **epochs=5**, batch-size=10, lr=0.001

从不同 epochs 下模型的表现来看，随着 epochs 的提高，训练过程中的迭代次数越高，从而使模型训练更加充分，loss 能够有效降低，准确率也会得到提升。

(3) Training with Different Batch Size

这里仍然采用普通梯度下降法（Normal GD），并固定 epochs=1，lr=0.1，探究训练过程和分类结果随 batch-size 的变化情况。

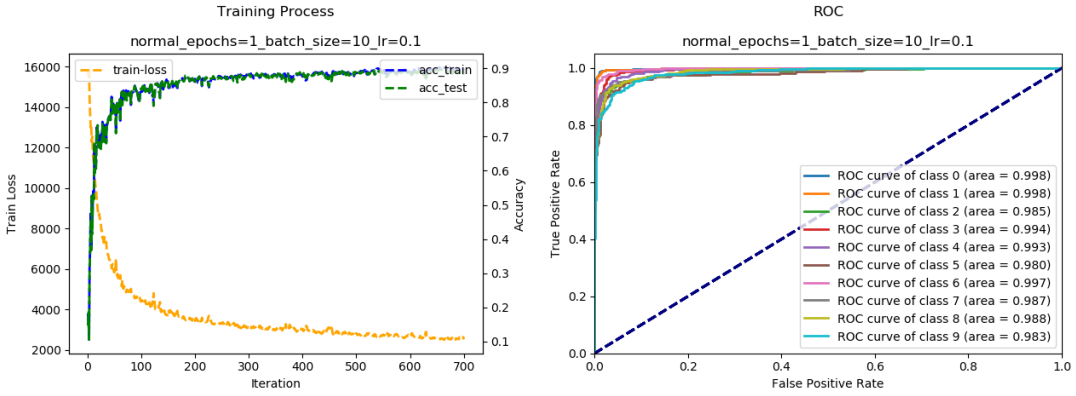


图 1-8 Result of Training with normal-GD, epochs=1, **batch-size=10**, lr=0.1

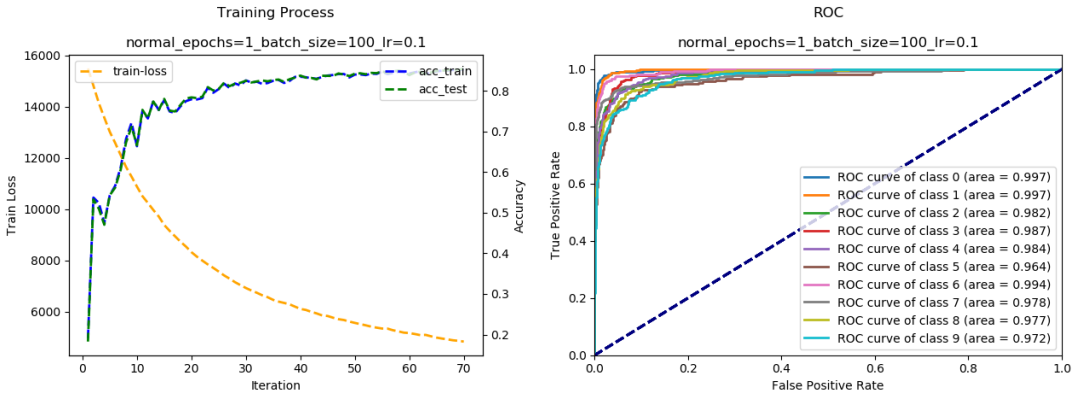


图 1-9 Result of Training with normal-GD, epochs=1, **batch-size=100**, lr=0.1

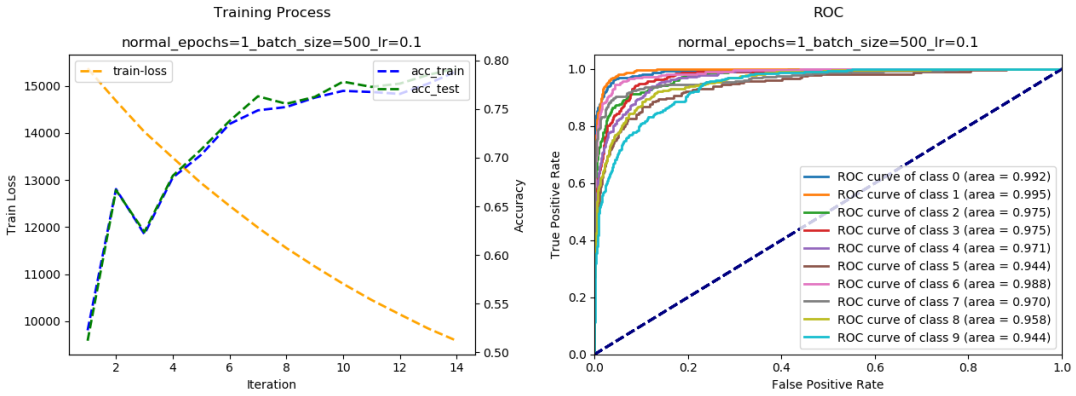


图 1-10 Result of Training with normal-GD, epochs=1, **batch-size=500**, lr=0.1

从上图可以看出，随着 batch-size 的增大，模型在训练集和测试集上的准确率逐步降低。从 loss 变化来看，该现象同样是训练不充分（Underfitting）导致的。

这是因为 batch-size 越大，训练集被划分的 batch 越少，从而在相同的 epochs 下训练迭代次数越少。通过对 batch-size 和 epochs 的分析可知，二者共同影响训练的迭代次数，在模型处于 Underfitting 的情况下，迭代次数 iteration 越多，模型越能得到充分训练，准确率越高。

(4) Training with Different Update Policies

Section 1.5.2 介绍了三种参数更新策略，分别为 Normal、Noise 和 Regularization，下面固定 epochs=1，batch-size=10，lr=0.1，探究不同参数更新策略对训练过程和分类结果的影响（设置 Regularization GD 的超参数 lambda=0.1）。

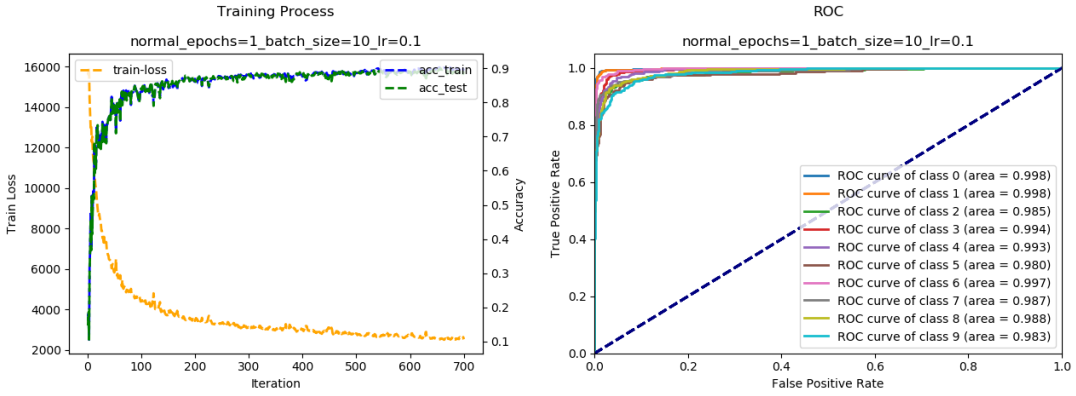


图 1-11 Result of Training with **normal-GD**, epochs=1, batch-size=10, lr=0.1

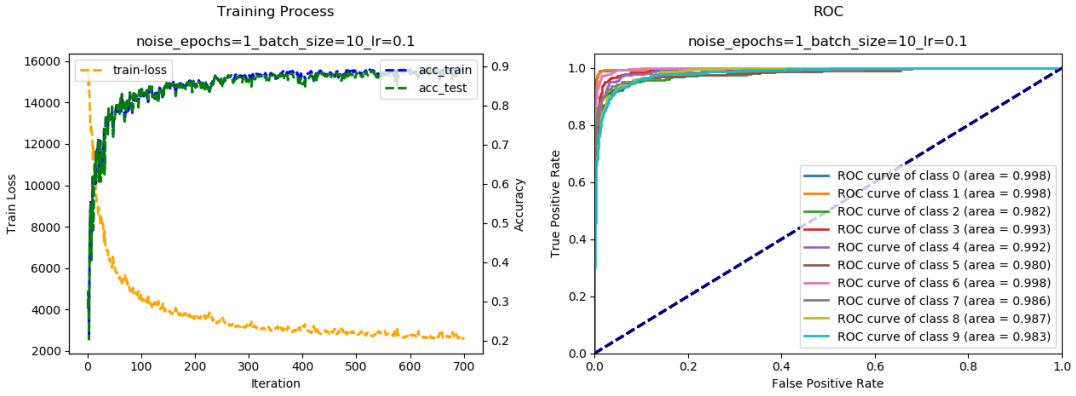


图 1-12 Result of Training with **noise-GD**, epochs=1, batch-size=10, lr=0.1

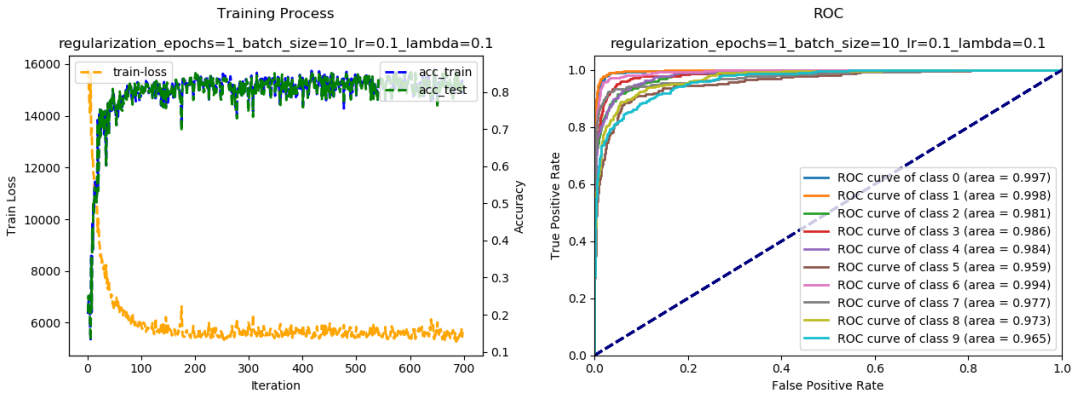


图 1-13 Result of Training with **regularization-GD**, epochs=1, batch-size=10, lr=0.1, lambda=0.1

从图 1-11 和图 1-12 中可以看出，训练表现和最终的分类结果与是否添加噪音的关系不大，loss 的变化趋势和波动幅度也基本相同（这里的原因也有可能是实验设定的噪音方差较小导致的）。从图 1-13 中可以看出，在添加正则项后，无论是模型在数据集上的准确率 Accuracy 还是训练集 Loss 都产生了较大幅度的震荡，且在相同的迭代次数下，未能达到普通更新策略下模型的准确率。

通过观察 Normal GD 和 Regularization GD 的参数更新方式可以发现，前者实际上是后者的特例，即当正则化因子 lambda=0 时，Regularization GD 将退化为 Normal GD。基于此，下面将探究采用带有正则项的梯度下降算法的条件下，lambda 的设定对分类结果的影响。

(5) Training with Different Lambda

本小节采用带有正则项的梯度下降法（Regularization GD），并固定 epochs=1，batch-size=10，lr=0.1，探究正则化因子 lambda 对训练过程和分类结果的影响。

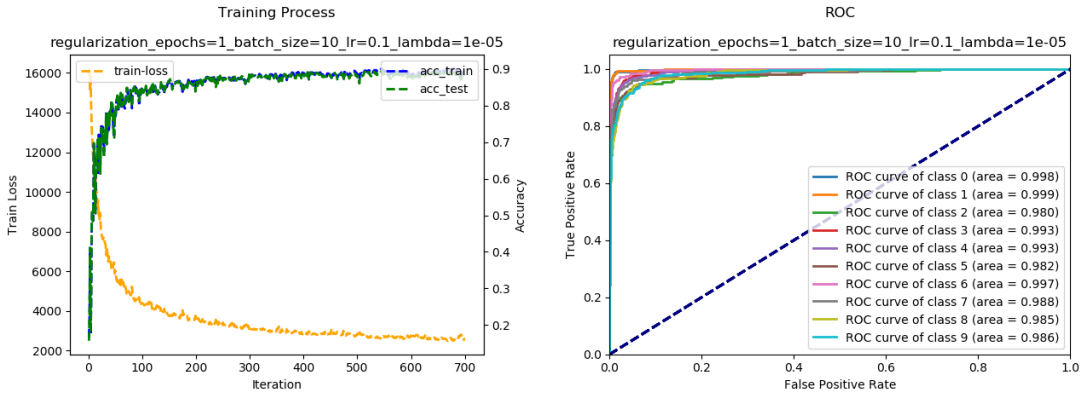


图 1-14 Result of Training with regularization-GD, epochs=1, batch-size=10, lr=0.1, **lambda=1e-5**

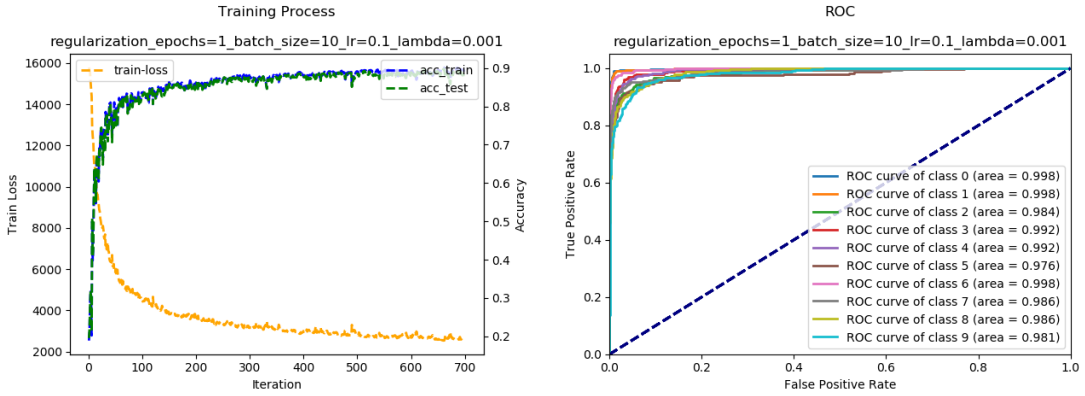


图 1-15 Result of Training with regularization-GD, epochs=1, batch-size=10, lr=0.1, **lambda=0.001**

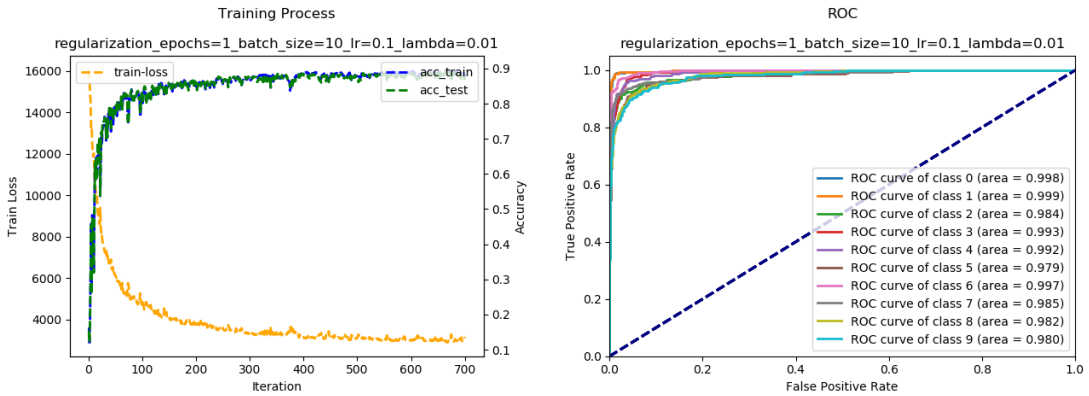


图 1-16 Result of Training with regularization-GD, epochs=1, batch-size=10, lr=0.1, **lambda=0.01**

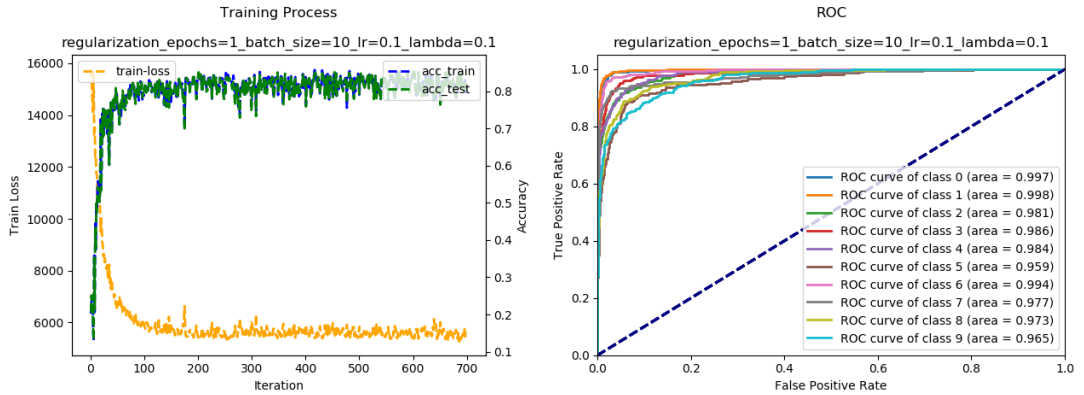


图 1-17 Result of Training with regularization-GD, epochs=1, batch-size=10, lr=0.1, **lambda=0.1**

从上图展示的训练过程和分类结果可以看出，正则化因子 lambda 将在一定程度上影响模型的训练过程，具体表现为较大的 lambda 会使得 Accuracy 和 Loss 曲线产生一定幅度的震荡。且当正则化因子为 0 时的震荡最小，此时退化为 Normal Gradient Descent。

1.7 Conclusion

本章节介绍了 Logistic Regression 的模型定义、损失函数以及参数求解方法，同时提出了 Normal GD、Noise GD 和 Regularization GD 三种不同参数更新策略。在实验部分，本文采用逻辑回归模型，结合以上参数更新策略对 MNIST 手写图片数据集进行了分类，并通过 Accuracy、Loss、ROC 曲线展示并分析了不同训练参数下的训练表现和分类结果。通过实验结果发现，带有正则项的梯度下降将会给 Accuracy 和 Loss 曲线带来一定程度上的震荡；噪音项的添加对最终的分类结果未产生明显降低或提高；同时较低且适当的学习率配合较大且合理的迭代次数将会使模型得到更加充分的训练，从而得到准确率更高的分类结果。

II. Softmax Loss with Center Loss

2.1 Comparison between Softmax and Center Loss

Logistic Regression 模型常采用 Softmax Loss 作为损失函数，其反映出样本被划分为对应类别的准确程度，刻画的是不同类别之间的差异性（Inter-Class）；为了衡量样本在类内的紧凑性，Wen Yandong[2]等人提出了 Center Loss 用于刻画类内的紧凑程度（Intra-Class）。两类损失函数的数学表达形式如下：

$$\ell_S = -\sum_{i=1}^{m} \log \frac{e^{W_{y_i}^T x_i + b_{y_i}}}{\sum_{j=1}^{n} e^{W_j^T x_i + b_j}}, \quad \ell_C = \frac{1}{2} \sum_{i=1}^{m} \left\| x_i - c_{y_i} \right\|_2^2$$

在人脸识别等实际应用中，样本的类内紧凑程度也会对分类结果产生较大影响，需要同时考虑类间差异性和类内紧凑性，因此损失函数通常为如下形式，其中 λ 为超参数，用于控制 Inter-Class 和 Intra-Class 的相对重要程度。

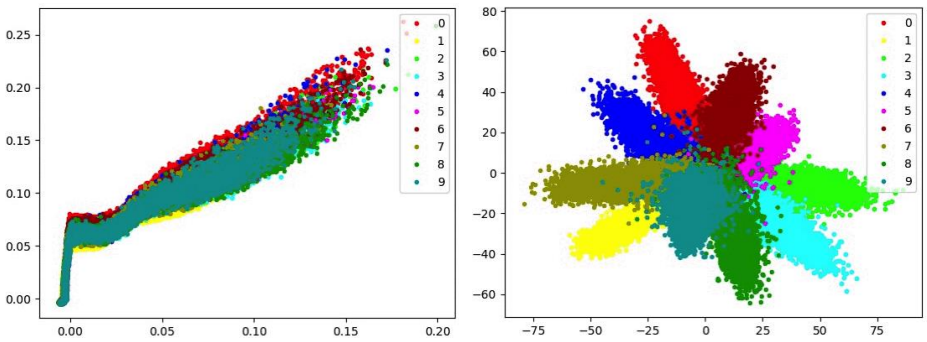
$$\ell = \ell_S + \lambda \ell_C = -\sum_{i=1}^{m} \log \frac{e^{W_{y_i}^T x_i + b_{y_i}}}{\sum_{j=1}^{n} e^{W_j^T x_i + b_j}} + \frac{\lambda}{2} \sum_{i=1}^{m} \left\| x_i - c_{y_i} \right\|_2^2$$

2.2 Feature Map on Different Loss Functions

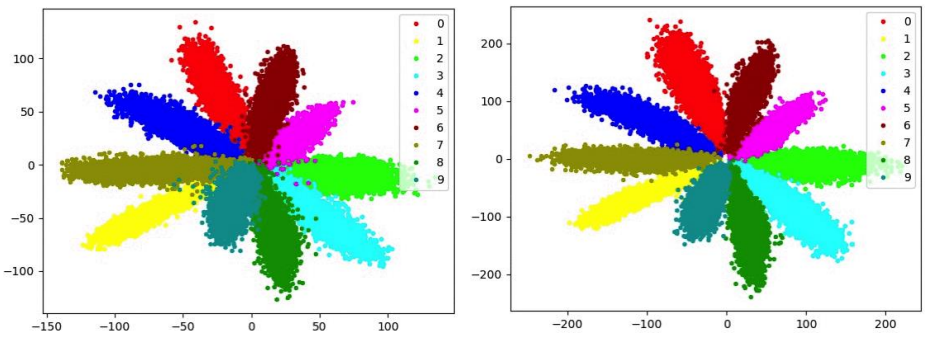
这里使用 CNN 卷积神经网络（Model），并以随机梯度下降 GD 为参数更新方式（Algorithm），分别采用 ℓ_S 和 $\ell = \ell_S + \lambda \ell_C$ 作为损失函数（Loss），同样对 MNIST 手写图片数据集进行分类。

2.2.1 Only Softmax Loss

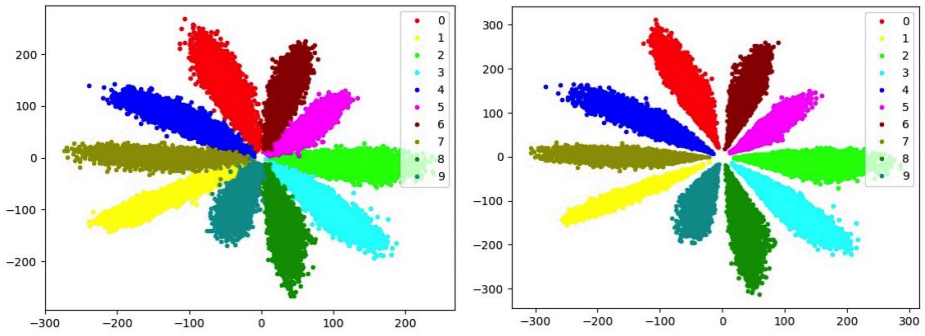
图 2-1 展示了损失函数为 ℓ_S 的条件下，每次 epochs 得到的最后一层 Feature Map 在二维空间上的映射图。



(a) epochs=1　　(b) epochs=5



(c) epochs=15　　(d) epochs=30

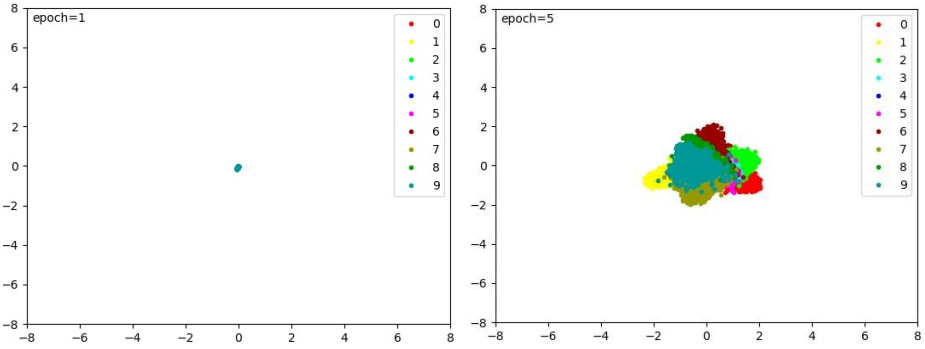


(e) epochs=40　　(f) epochs=50

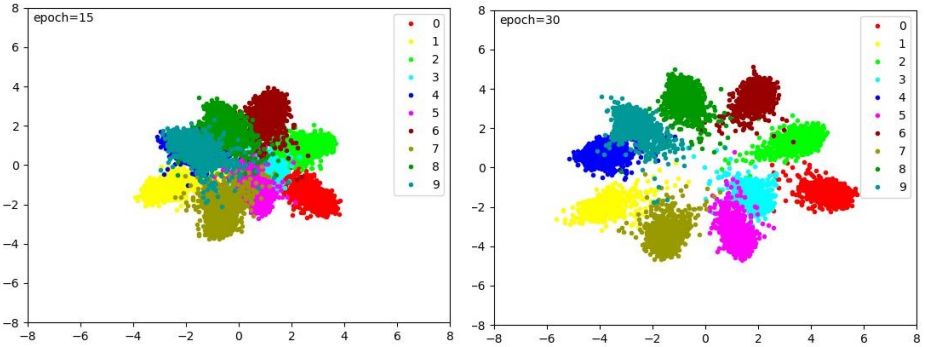
图 2-1 Feature Map of Two-Dimension with ℓ_S

2.2.2 Softmax with Center Loss

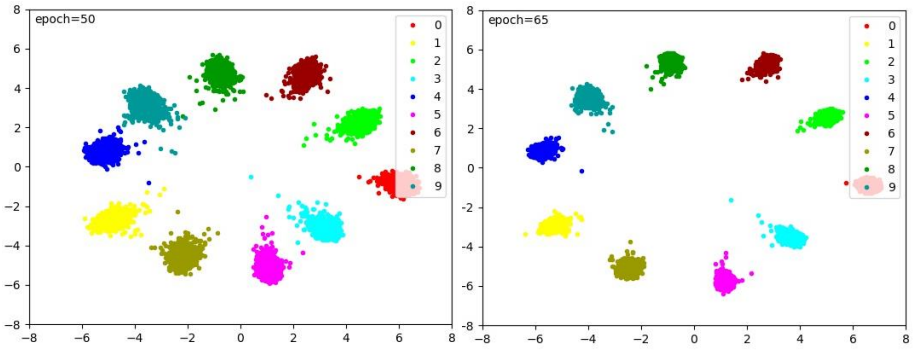
图 2-2 展示了损失函数为$\ell = \ell_S + \lambda \ell_C$的条件下，每次 epochs 得到的最后一层 Feature Map 在二维空间上的映射图（这里选取$\lambda = 1$）。



(a) epochs=1　　(b) epochs=5



(c) epochs=15　　(d) epochs=30



(e) epochs=50　　(f) epochs=65

图 2-2 Feature Map of Two-Dimension with $\ell = \ell_S + \lambda \ell_C$

2.3 Analysis

从图 2-1 中可以看出，将 Softmax Loss 作为损失函数时，随着训练迭代的进行，训练集的各个类别之间的差异越来越明显，反映在图像上即为类与类之间的重叠区域不断缩小；从图 2-2 中可以看出，将 Center Loss 与 Softmax Loss 结合共同作为损失函数时，各个类别之间重叠区域不断缩小的同时，类内的紧凑程度也不断提高，使得各个类别之间更易于划分。因此对于需要同时考虑类内紧凑型和类间差异性的场景下，将 $\ell = \ell_S + \lambda \ell_C$ 作为损失函数将会是一个较好的选择。

III. Code Analysis

3.1 Multinomial Logistic Regression

本节主要展示模型训练相关代码，MNIST 数据集读取这里不再介绍。

P.S. 完整代码见与本报告一同提交的工程文件 P5-Assignment- logistic_regression

表 3-1 主函数整体流程

● 3-1 MainFunction

```
if __name__ == '__main__':
    # 加载训练和测试数据
    print('-> loading dataset...')
    train_data, test_data, label_train, label_test = DataSetUtils.load_data_set()
    # 使用预生成的随机序列选取 7000 个训练样本和 3000 个测试样本
    train_data, test_data, label_train, label_test = random_select(train_data, test_data,
label_train, label_test)
    print(collections.Counter(label_train))
    print(collections.Counter(label_test))
    # 创建 Logistic 回归实例
    print('-> creating model...')
logistic_instance = LogisticUtils.LogisticRegressionML(train_data, test_data, label_train,
```

- 3-1 MainFunction

```
label_test, epochs=1, batch_size=10, lr=0.1, gd_type=normal)
    # logistic_instance = LogisticUtils.LogisticRegressionML(train_data, test_data, label_train,
label_test, epochs=1, batch_size=10, lr=0.1, gd_type='regularization')
    # logistic_instance = LogisticUtils.LogisticRegressionML(train_data, test_data, label_train,
label_test, epochs=1, batch_size=10, lr=0.1, gd_type=noise)
        if logistic_instance.model_exists():
            print('local model already exists')
            print('-> loading models...')
            logistic_instance = logistic_instance.load_model()
            print(logistic_instance.accuracy_on_train())
            print(logistic_instance.accuracy_on_test())
        else:
            print('-> training model...')
            logistic_instance.train()
            print('-> saving model...')
            logistic_instance.save_model()
            print(logistic_instance.accuracy_on_train())
            print(logistic_instance.accuracy_on_test())
        # 绘制各个类别的 ROC 曲线
        PlotUtils.plot_roc_curve(logistic_instance)
        # 绘制训练过程记录
        PlotUtils.plot_train_record(logistic_instance)
```

表 3-2 LogisticRegressionML 构造函数

- 3-2 Initializing of LogisticRegressionML

```
    def __init__(self, train_set, test_set,train_label, test_label, epochs, batch_size, lr,_lambda=1e-5,
gd_type='normal'):
        # =========@BigData=========
        # 导入训练和测试数据
        self.train_set = np.reshape(train_set, (len(train_set), np.size(train_set[0])))
        self.test_set = np.reshape(test_set, (len(test_set), np.size(test_set[0])))
        self.train_label = train_label
        self.test_label = test_label
        self.gd_type = gd_type
        # =========@Model=========
        # 模型参数
        self.weight = np.zeros((CLASS_NUM, len(self.train_set[0])))
        self.bias = np.zeros(CLASS_NUM)
        # 训练参数
        self.epochs = epochs
        self.batch_size = batch_size
        self.lr = lr
```

● 3-2 Initializing of LogisticRegressionML

```
        self._lambda = _lambda
        # 训练过程记录
        self.train_record = []
        # 模型保存相关参数
        if self.gd_type == 'regularization':
            self.save_path =
'./model/{0}_epochs={1}_batch_size={2}_lr={3}_lambda={4}.pkl'.format(self.gd_type,self.epochs,
self.batch_size, self.lr, self._lambda)
        else:
            self.save_path =
'./model/{0}_epochs={1}_batch_size={2}_lr={3}.pkl'.format(self.gd_type,self.epochs,self.batch_siz
e, self.lr)
```

表 3-3 LogisticRegressionML 损失函数

● 3-3 Loss Function

```
    def loss(self):
        # 计算损失 loss
        train_num = len(self.train_set)
        loss = 0
        for i in range(train_num):
            loss += -np.log(soft_max(np.dot(self.weight, self.train_set[i]) +
self.bias)[self.train_label[i]])
        return loss
```

表 3-4 LogisticRegressionML 训练过程（参数更新）

● 3-4 Training Process

```
    def train(self):
        # 循环所有样本为一个 epoch
        for k in range(self.epochs):
            # 每次 epoch 执行一次 shuffle
            print('---> shuffling train_set for epoch {}...'.format(k))
            shuffle_index = random.sample(range(0, len(self.train_set)), len(self.train_set))
            self.train_set = [self.train_set[i] for i in shuffle_index]
            self.train_label = [self.train_label[i] for i in shuffle_index]
            # 循环一个 batch 内的样本为一个 iteration
            batch_num = int(len(self.train_set) / self.batch_size)
            print('batch_num: ', batch_num)
            x_batches = np.zeros((batch_num, self.batch_size, len(self.train_set[0])))
            y_batches = np.zeros((batch_num, self.batch_size))
            # 分批
            for i in range(0, len(self.train_set), self.batch_size):
                x_batches[int(i / self.batch_size)] = self.train_set[i:i + self.batch_size]
```

- 表 3-4 LogisticRegressionML 训练过程（参数更新）

```
                y_batches[int(i / self.batch_size)] = self.train_label[i:i + self.batch_size]
            for i in range(batch_num):
                # 每次 iter 都要进行梯度初始化
                gd_w = np.zeros((CLASS_NUM, len(self.train_set[0])))
                gd_b = np.zeros(CLASS_NUM)
                # 从 trainSet 中随机选取 batch_size 个样本（MBGD）
                for j in range(self.batch_size):
                    gd_weight, gd_bias = gradient(self.weight, self.bias, x_batches[i][j],
y_batches[i][j], self._lambda, self.gd_type)
                    gd_w += gd_weight
                    gd_b += gd_bias
                # 取整个 batch 的梯度平均为本次梯度更新值
                self.weight -= self.lr * (gd_w / self.batch_size)
                self.bias -= self.lr * (gd_b / self.batch_size)
                acc_train = self.accuracy_on_train()
                acc_test = self.accuracy_on_test()
                loss = self.loss()
                # 保存每一次训练迭代记录
                self.train_record.append([k, i, acc_train, acc_test, loss])
                if i % 1 == 0:
                    print('--->> iter= {0}, acc_train= {1}%, acc_test= {2}%, loss=
{3}'.format(i, acc_train * 100, acc_test * 100, loss))
```

表 3-5 LogisticRegressionML 计算梯度

- 3-5 Calculate Gradient

```
def gradient(w, b, x, y, _lambda=0.0, gd_type='normal'):
    gd_weight = np.zeros(np.shape(w))
    gd_bias = np.zeros(np.shape(b))
    s = soft_max(np.dot(w, x) + b)
    r_w = np.shape(w)[0]
    c_w = np.shape(w)[1]
    b_column = b.shape[0]
    # 对 Wij 和 bi 分别求梯度
    for i in range(r_w):
        for j in range(c_w):
            gd_weight[i][j] = (s[i] - 1) * x[j] if y == i else s[i] * x[j]
    for i in range(b_column):
        gd_bias[i] = s[i] - 1 if y == i else s[i]
    # 带有正则项的梯度下降
    if gd_type == 'regularization':
        gd_weight += _lambda * w
    # 带有随机噪音项的梯度下降
```

- 3-5 Calculate Gradient

```
    elif gd_type == 'noise':
        gd_weight += np.random.normal(0, 0.1, np.shape(w))
    # 普通随机下降
    return gd_weight, gd_bias
```

表 3-6 绘制 ROC 曲线

- 3-6 PlotRocCurve

```
def plot_roc_curve(logistic_instance):
    # 对测试样本所属类别进行 one-hot 编码
    test_labels = label_binarize(logistic_instance.test_label, classes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    # 获取每个测试样本所属类别得分
    test_scores = logistic_instance.score_on_test()
    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    plt.figure()
    for i in range(CLASS_NUM):
        fpr[i], tpr[i], _ = roc_curve(test_labels[:, i], test_scores[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])
        lw = 2
        plt.plot(fpr[i], tpr[i], lw=lw, label='ROC curve of class %d (area = %0.3f)' % (i,
roc_auc[i]))
        plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc="lower right")
    plt.title('ROC\n\n{0}'.format(logistic_instance.save_path.split('/')[2][:-4]))
    plt.show()
```

表 3-7 绘制训练过程 Accuracy 和 Loss 曲线

- 3-7 PlotTrainingProcess

```
def plot_train_record(logistic_instance):
    # 获取训练记录
    train_record = np.array(logistic_instance.train_record)
    fig, ax1 = plt.subplots()
    # 创建第二个坐标轴
    iter_index = np.arange(1, len(train_record) + 1)
    acc_train = np.array(train_record)[:, 2]
    acc_test = train_record[:, 3]
```

● 3-7 PlotTrainingProcess

```
    train_loss = train_record[:, 4]
    # 绘图
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Train Loss')
    ax1.plot(iter_index, train_loss, '--', c='orange', label='train-loss', linewidth=2)
    plt.legend(loc=2)
    ax2 = ax1.twinx()
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Accuracy')
    ax2.plot(iter_index, acc_train, '--', c='blue', label='acc_train', linewidth=2)
    ax2.plot(iter_index, acc_test, '--', c='green', label='acc_test', linewidth=2)
    plt.legend(loc=1)
    plt.title('Training Process\n\n{0}'.format(logistic_instance.save_path.split('/')[2][:-4]))
    plt.show()
```

3.2 Feature Map on Different Loss Functions

本文采用 CNN 作为模型，网络搭建、训练以及可视化相关代码如下表所示。

表 3-8 主函数流程

● 3-8 MainFunction

```
use_cuda = torch.cuda.is_available()
# device = torch.device("cuda" if use_cuda else "cpu")
device = torch.device("cuda")
# MNIST 数据集
trainset = datasets.MNIST('../', download=False, train=True, transform=transforms.Compose([
    transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]))
train_loader = DataLoader(trainset, batch_size=128, shuffle=True, num_workers=0)
# Model
model = Net().to(device)
# NLLLoss
nllloss = nn.NLLLoss().to(device)    # CrossEntropyLoss = log_softmax + NLLLoss
# [2]提出的 CenterLoss
loss_weight = 1
centerloss = CenterLoss(10, 2).to(device)
# optimzer4nn
optimizer4nn = optim.SGD(model.parameters(), lr=0.001, momentum=0.9, weight_decay=0.0005)
sheduler = lr_scheduler.StepLR(optimizer4nn, 20, gamma=0.8)
# optimzer4center
optimzer4center = optim.SGD(centerloss.parameters(), lr=0.5)
for epoch in range(100):
    sheduler.step()
    train(epoch + 1)
```

表 3-9 CNN 网络初始化

- 3-9 Net-Initializing

```
def __init__(self):
    super(Net, self).__init__()
    self.conv1_1 = nn.Conv2d(1, 32, kernel_size=5, padding=2)
    self.prelu1_1 = nn.PReLU()
    self.conv1_2 = nn.Conv2d(32, 32, kernel_size=5, padding=2)
    self.prelu1_2 = nn.PReLU()
    self.conv2_1 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
    self.prelu2_1 = nn.PReLU()
    self.conv2_2 = nn.Conv2d(64, 64, kernel_size=5, padding=2)
    self.prelu2_2 = nn.PReLU()
    self.conv3_1 = nn.Conv2d(64, 128, kernel_size=5, padding=2)
    self.prelu3_1 = nn.PReLU()
    self.conv3_2 = nn.Conv2d(128, 128, kernel_size=5, padding=2)
    self.prelu3_2 = nn.PReLU()
    self.preluip1 = nn.PReLU()
    self.ip1 = nn.Linear(128 * 3 * 3, 2)
    self.ip2 = nn.Linear(2, 10, bias=False)
```

表 3-10 网络训练

- 3-10 Training

```
def train(epoch):
    print("Training... Epoch = %d" % epoch)
    ip1_loader = []
    idx_loader = []
    for i, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        ip1, pred = model(data)
        loss = nllloss(pred, target) + loss_weight * centerloss(target, ip1)
        optimizer4nn.zero_grad()
        optimzer4center.zero_grad()
        loss.backward()
        optimizer4nn.step()
        optimzer4center.step()
        ip1_loader.append(ip1)
        idx_loader.append((target))
    feat = torch.cat(ip1_loader, 0)
    labels = torch.cat(idx_loader, 0)
    visualize(feat.data.cpu().numpy(), labels.data.cpu().numpy(), epoch)
```

参考文献

[1] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," in IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141-142, Nov. 2012, doi: 10.1109/MSP.2012.2211477.

[2] Wen, Yandong & Zhang, Kaipeng & Li, Zhifeng & Qiao, Yu. (2016). A Discriminative Feature Learning Approach for Deep Face Recognition. LNCS. 9911. 499-515. 10.1007/978-3-319-46478-7_31.