模式识别与机器学习 -- 实验2

本实验包含以下部分：

- softmax （50%）
- svm （50%）

softmax

1 手动实现 Softmax 函数 （15%）

```
def my_softmax(logits):
    max_logits, _ = logits.max(dim=-1, keepdim=True)
    exp_logits = torch.exp(logits - max_logits)
    sum_exp_logits = exp_logits.sum(dim=-1, keepdim=True)
    probs = exp_logits / sum_exp_logits
    return probs
```

说明 使用 logits.max(dim=-1, keepdim=True) 找到每行的最大值 减去最大值后再计算 exp 使用 exp_logits.sum(dim=-1, keepdim=True)求行和 除以行和得到softmax归一结果

2 创建自定义 Softmax 层 （15%）

```
class MySoftmax(nn.Module):
    def __init__(self, dim=-1):
        super().__init__()
        self.dim = dim

    def forward(self, x):
        return my_softmax(x)
```

说明 继承 nn.Module 类 在 **init** 方法中初始化维度参数 在 forward 方法中调用自定义的 my_softmax 函数

3 参数调优实验（无需给出代码）

4 提交实验结果，只需截图最后的实验结果汇总和最佳模型的配置即可（20%）

```
实验结果汇总:
       Experiment Parameter Best Val Acc Final Val Acc Final Val Loss
0   Learning Rate    0.0001       0.2216        0.2216         2.2243
1   Learning Rate     0.001       0.6732        0.6732         0.8504
2   Learning Rate      0.01       0.8365        0.8353         0.4593
3   Learning Rate       0.1       0.8577        0.8554         0.3946
4      Batch Size        16       0.8423        0.8423         0.4444
5      Batch Size        32       0.8407        0.8407         0.4516
6      Batch Size        64       0.8323        0.8323         0.4723
7      Batch Size       128       0.8236        0.8236         0.5061
8    Architecture       小网络       0.8360        0.8360         0.4694
9    Architecture       中网络       0.8371        0.8371         0.4621
10   Architecture       大网络       0.8342        0.8342         0.4654
11      Optimizer       SGD       0.8298        0.8275         0.4812
12      Optimizer      Adam       0.8830        0.8813         0.3419
```

```
======================================================================
训练最佳模型
======================================================================

最佳配置:
  学习率: 0.01
  批次大小: 64
  网络结构: (256, 128)
  优化器: Adam
Training:  34% 9500/28140 [01:19<02:14, 138.42it/s, epoch=10, val_acc=0.857]

早停于 epoch 10, step 9500

最终验证准确率: 0.8626
模型已保存到 best_model.pth
```

svm

一、损失和梯度的计算

1, 循环实现中的梯度计算（10%）

补全 fduml.linear_svm import中的svm_loss_naive函数

在这里写代码，并简单说明（注意，没有说明会适当扣分）

```
def svm_loss_naive(W, X, y, reg):
    """
    结构化SVM损失函数，朴素实现（使用循环）。
```

```
    输入数据的维度为D，共有C个类别，我们在包含N个样本的小批量数据上操作。

    输入参数：
    - W：形状为(D，C)的numpy数组，包含权重。
    - X：形状为(N，D)的numpy数组，包含一个小批量数据。
    - y：形状为(N,)的numpy数组，包含训练标签；y[i] = c 表示
      X[i]的标签为c，其中 0 <= c < C。
    - reg：(float) 正则化强度

    返回一个元组：
    - loss：单个浮点数，表示损失值
    - gradient：关于权重W的梯度；与W形状相同的数组
    """
    dW = np.zeros(W.shape)

    num_classes = W.shape[1]
    num_train = X.shape[0]
    loss = 0.0
    for i in range(num_train):
        scores = X[i].dot(W)
        correct_class_score = scores[y[i]]
        for j in range(num_classes):
            if j == y[i]:
                continue
            margin = scores[j] - correct_class_score + 1  # note delta = 1
            if margin > 0:
                loss += margin

    loss /= num_train

    loss += reg * np.sum(W * W)

    #############################################################################
    # TODO:                                                                     #
    # 计算损失函数的梯度并将其存储在dW中。                                      #
    # 在计算损失的同时计算导数可能更简单                                         #
    # 可以修改上面的一些代码来计算梯度                                          #
    #############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    for i in range(num_train):
        scores = X[i].dot(W)
        correct_class_score = scores[y[i]]
        for j in range(num_classes):
            if j == y[i]:
                continue
            margin = scores[j] - correct_class_score + 1  # note delta = 1
            if margin > 0:
                dW[:, j] += X[i]
                dW[:, y[i]] -= X[i]
    dW /= num_train
    dW += 2 * reg * W
```

```
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return loss, dW
```

说明 在计算损失的同时，更新梯度dW 对于每个样本i和每个类别j，如果margin>0，则对dW进行相应的增减 最后对dW进行平均并加上正则化项的梯度

2, 向量实现中的损失计算和梯度计算（15%）

补全 fduml.linear_svm import中的svm_loss_vectorized函数

在这里写代码，并简单说明

```
def svm_loss_vectorized(W, X, y, reg):
    """
    结构化SVM损失函数，向量化实现。

    输入和输出与svm_loss_naive相同。
    """
    loss = 0.0
    dW = np.zeros(W.shape)

    ###########################################################################
    # TODO:                                                                   #
    # 实现结构化SVM损失的向量化版本，将结果存储在loss中。                      #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    num_train = X.shape[0]
    scores = X.dot(W)  # (N, C)
    correct_class_scores = scores[np.arange(num_train), y].reshape(-1, 1)
    margins = np.maximum(0, scores - correct_class_scores + 1)
    margins[np.arange(num_train), y] = 0  # 不计算正确类别的margin
    loss = np.sum(margins) / num_train
    loss += reg * np.sum(W * W)


    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    ###########################################################################
    # TODO:                                                                   #
    # 实现SVM损失梯度的向量化版本，将结果存储在dW中。                          #
    # 提示：与其从头计算梯度，重用一些损失计算时的中间值可能更容易             #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    binary = margins
    binary[margins > 0] = 1
    row_sum = np.sum(binary, axis=1)
    binary[np.arange(num_train), y] = -row_sum
```

```
    dW = X.T.dot(binary) / num_train
    dW += 2 * reg * W

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return loss, dW
```

说明 计算所有样本的分数矩阵scores 提取正确类别的分数correct_class_scores并进行广播 计算margin矩阵，并将正确类别的margin设为0 计算总损失并加上正则化项 使用binary矩阵标记哪些margin>0，并计算每行的和 更新dW并加上正则化项的梯度

3, 在这里提交ipynb中的相关检查结果（不占额外分数，但这是判断上面的实现是否正确的重要依据

```python
# 实现梯度计算后，用下面的代码计算，并使用提供的函数进行梯度检查
# 误差应该会在1e-2以内，大多数在1e-5以内

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

from fduml.utils import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

# 在有正则化的情况下，重新检查
loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad)
```

[9] ✓ 14.5s

```
numerical: -5.038809 analytic: -5.044244, relative error: 5.389873e-04
numerical: -20.286494 analytic: -20.279482, relative error: 1.728468e-04
numerical: -20.900922 analytic: -20.910024, relative error: 2.176958e-04
numerical: 10.907980 analytic: 10.909502, relative error: 6.976632e-05
numerical: -26.677396 analytic: -26.672477, relative error: 9.220535e-05
numerical: -6.571528 analytic: -6.571528, relative error: 8.180762e-12
numerical: 9.493192 analytic: 9.493192, relative error: 1.243785e-10
numerical: -19.177079 analytic: -19.179870, relative error: 7.277433e-05
numerical: 1.196430 analytic: 1.199273, relative error: 1.186744e-03
numerical: 13.156268 analytic: 13.155351, relative error: 3.484371e-05
numerical: 8.228801 analytic: 8.221748, relative error: 4.287615e-04
numerical: -6.340842 analytic: -6.350201, relative error: 7.374796e-04
numerical: 9.588452 analytic: 9.588452, relative error: 3.908671e-11
numerical: 20.853010 analytic: 20.863433, relative error: 2.498554e-04
numerical: 8.461353 analytic: 8.461353, relative error: 3.037693e-11
numerical: 25.790870 analytic: 25.791412, relative error: 1.050639e-05
numerical: 10.151353 analytic: 10.151353, relative error: 5.540217e-11
numerical: -12.346893 analytic: -12.341305, relative error: 2.263587e-04
numerical: -12.322345 analytic: -12.322345, relative error: 1.035569e-10
numerical: -10.572321 analytic: -10.572321, relative error: 4.330814e-11
```

```python
# 接下来实现向量化的 svm 损失计算，验证正确性并比较用时
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from fduml.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# The losses should match but your vectorized implementation should be much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

[10] ✓ 0.5s

```
Naive loss: 8.882890e+00 computed in 0.352760s
Vectorized loss: 8.882890e+00 computed in 0.200502s
difference: 0.000000
```

```
    # 完成向量化的梯度计算，并与 naive 实现进行比较，验证正确性
    tic = time.time()
    _, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
    toc = time.time()
    print('Naive loss and gradient: computed in %fs' % (toc - tic))

    tic = time.time()
    _, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
    toc = time.time()
    print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

    difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
    print('difference: %f' % difference)
11] ✓ 0.5s

Naive loss and gradient: computed in 0.357700s
Vectorized loss and gradient: computed in 0.201579s
difference: 0.000000
```

二、实现SGD

代码+简单说明（10%）

```
class LinearClassifier(object):
    def __init__(self):
        self.W = None

    def train(
        self,
        X,
        y,
        learning_rate=1e-3,
        reg=1e-5,
        num_iters=100,
        batch_size=200,
        verbose=False,
    ):
        """
        使用随机梯度下降训练该线性分类器。

        输入：
        - X：形状为（N，D）的 numpy 数组，包含训练数据；有 N 个训练样本，每个样本的维度为 D。
        - y：形状为（N,）的 numpy 数组，包含训练标签；y[i] = c 表示 X[i] 的标签为 0 <= c < C，其中 C 是类别数。
        - learning_rate：（float）优化的学习率。
        - reg：（float）正则化强度。
        - num_iters：（integer）优化时的迭代步数
        - batch_size：（integer）每一步使用的训练样本数量。
        - verbose：（boolean）如果为 true，在优化过程中打印进度。

        输出：
        包含每次训练迭代中损失函数值的列表。
```

```
        """
        num_train, dim = X.shape
        num_classes = (
            np.max(y) + 1
        )  # 假设 y 的取值为 0...K-1, 其中 K 是类别数
        if self.W is None:
            # 延迟初始化 W
            self.W = 0.001 * np.random.randn(dim, num_classes)

        # 运行随机梯度下降以优化 W
        loss_history = []
        for it in range(num_iters):
            X_batch = None
            y_batch = None


###########################################################################
            # TODO:
#
            # 从训练数据中采样 batch_size 个元素及其对应的标签，用于本轮梯度下降。
#
            # 将数据存储在 X_batch 中，对应的标签存储在 y_batch 中；采样后 X_batch 的
#
            # 形状应为 (batch_size, dim)，y_batch 的形状应为 (batch_size,)
#
            #
#
            # 提示：使用 np.random.choice 生成索引。有放回采样比无放回采样更快。
#

###########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            indices = np.random.choice(num_train, batch_size, replace=True)
            X_batch = X[indices]
            y_batch = y[indices]

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            # 评估损失和梯度
            loss, grad = self.loss(X_batch, y_batch, reg)
            loss_history.append(loss)

            # 执行参数更新

###########################################################################
            # TODO:
#
            # 使用梯度和学习率更新权重。
#

###########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
            self.W -= learning_rate * grad

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            if verbose and it % 100 == 0:
                print("iteration %d / %d: loss %f" % (it, num_iters, loss))

        return loss_history

    def predict(self, X):
        """
        使用该线性分类器的训练权重预测数据点的标签。

        输入:
        - X: 形状为 (N, D) 的 numpy 数组，包含训练数据；有 N 个训练样本，每个样本的维度为 D。

        返回:
        - y_pred: X 中数据的预测标签。y_pred 是一个长度为 N 的一维数组，每个元素是一个整数，表示预测的类别。
        """
        y_pred = np.zeros(X.shape[0])

###########################################################################
        # TODO:
#
        # 实现此方法。将预测标签存储在 y_pred 中。
#

###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        scores = X.dot(self.W)
        y_pred = np.argmax(scores, axis=1)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return y_pred

    def loss(self, X_batch, y_batch, reg):
        """
        计算损失函数及其导数。
        子类将重写此方法。

        输入:
        - X_batch: 形状为 (N, D) 的 numpy 数组，包含一个包含 N 个数据点的小批量；每个点的维度为 D。
        - y_batch: 形状为 (N,) 的 numpy 数组，包含小批量的标签。
        - reg: (float) 正则化强度。

        返回: 包含以下内容的元组:
        - loss 作为单个浮点数
        - 关于 self.W 的梯度；与 W 形状相同的数组
        """
        pass
```

说明 定义 LinearClassifier 类，包含初始化、训练、预测和损失计算方法 在 train 方法中实现随机梯度下降，包括随机批量采样、损失和梯度计算以及权重更新 在 predict 方法中使用当前权重进行预测，预测类别为得分最高的类别 loss 方法作为占位符，子类将实现具体的损失函数

三、利用验证集做超参数调优

表格记录（5*5），可以自己尝试不同的组合 （10%）

学习率 \ 正则化强度	1e4	2.5e4	5e4	1e5	2.5e5
1e-8	0.196	0.176	0.190	0.200	0.269
5e-8	0.237	0.281	0.327	0.337	0.345
1e-7	0.307	0.339	0.343	0.324	0.322
5e-7	0.335	0.326	0.317	0.315	0.273
1e-6	0.342	0.295	0.241	0.228	0.209

最优的结果的loss曲线截图和正确率截图（5%）

```
best_params
learning_rate: 5e-08
regularization_strength: 250000.0
iteration 0 / 1500: loss 7691.141350
iteration 100 / 1500: loss 54.266758
iteration 200 / 1500: loss 6.653885
iteration 300 / 1500: loss 6.569456
iteration 400 / 1500: loss 6.851346
iteration 500 / 1500: loss 6.729710
iteration 600 / 1500: loss 6.668028
iteration 700 / 1500: loss 6.338237
iteration 800 / 1500: loss 6.619338
iteration 900 / 1500: loss 6.124870
iteration 1000 / 1500: loss 6.574314
iteration 1100 / 1500: loss 6.784824
iteration 1200 / 1500: loss 6.504430
iteration 1300 / 1500: loss 7.114522
iteration 1400 / 1500: loss 6.565658

 accuracy on best hyperparams: 0.340000
```