一. KNN 分类器对 CIFAR10 数据集进行图像分类

1. 实现说明

①CIFAR10 数据集：

包含 50000 张训练集图片（data_batch_1~data_batch_5 各 10000 张），10000 张测试集图片（test_batch 中），每张图片分别属于 10 个类别中的 1 个。要求在训练集上对分类器进行训练，在测试集上检验分类器的正确率。

②KNN 算法的原理：

对于测试集中的每个图像：扫描训练集中每张图像，计算二者之间的距离。之后找出与其最相近的 K 张图像，选取这 K 张图像中出现频率最高的类别作为对其预测的类别。本次作业中选用 K=5，并且使用 L2 距离计算图像之间的距离。

③实现细节：

可以参见代码中的注释，如下图所示（同时另外给出了.py 源文件）：

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

    def predict(self, X1, X2, Y, K = 5, batch_size = 200):
        y = []
        for i in range(0,X1.shape[0]):  #   对测试集中的各条数据进行分类测试
            if ((i % 10) ==0):  #   每测试10条显示一次测试进度
                print("测试进度为: %f " % (i/X1.shape[0]))
            res = []
            for j in range(0, X2.shape[0]): #   遍历训练集中的各条数据，使用L2距离对比相似程度
                dist=0
                for k in range(0,X1.shape[1]):  #   计算两张图像（一张测试集，一张训练集）的L2距离
                    dist += ((X1[i][k] - X2[j][k]) ** 2) ** 0.5
                res.append([dist,Y[j]])
            score = []
            for j in range (0, np.max(Y) + 1):  #   初始化各个类别在最近K邻中的出现次数
                score.append(0)
            for k in range(0,K):    #找到最近K邻的数据
                MIN = 10000000
                temp = 0
                for l in range(0,X2.shape[0]):
                    if (res[l][0] < MIN):
                        MIN = res[l][0]
                        temp = l
                score[res[temp][1]] += 1
                res[temp][0] = 10000000
            y.append(find(score, np.max(score)))    #   根据最近K邻的数据获取出现次数最多的类别
        return y
```

2.运行说明

本次作业通过在 Visual Studio Code 使用 Anaconda 环境对编写的代码进行解释执行，其中使用到的 numpy 和 pickle 库均已在 Anaconda 环境中安装。通过控制台命令 python -u “文件路径\KNN.py” 可以对源程序进行解释执行。对测试集上的每张图像，使用 KNN 算法对其作出分类预测。运行过程中每隔 10 张图像将显示出测试进度。

由于 KNN 算法在对测试集上的每张图片进行分类预测时需要将训练集中的图片全部遍历一遍，因此效率十分低下。如果对训练集和测试集中的数据全部予以使用，将花费大量的时间。因此分别将训练集、测试集的数据缩小至原规模的 1/ratio1、1/ratio2 以节省运行时间。

3.运行结果

将测试集缩小至原本规模的 1%，训练集缩小至原本规模的 2%，在运行约 15 分钟后将运行结束，最终准确率为 23%。如下图所示：

```
(base) PS F:\VScodeWorkplace\Py\LearnPytorch\MyLearnPytorch> python -u "f:\VScodeWorkplace\Py\LearnPytorch\MyLearnPytorch\KNN.py"
开始测试KNN...
测试进度为：0.000000
测试进度为：0.100000
测试进度为：0.200000
测试进度为：0.300000
测试进度为：0.400000
测试进度为：0.500000
测试进度为：0.600000
测试进度为：0.700000
测试进度为：0.800000
测试进度为：0.900000
测试完成！
在测试集上的准确率为：0.230000
(base) PS F:\VScodeWorkplace\Py\LearnPytorch\MyLearnPytorch>
```

二. SVM 分类器对 CIFAR10 数据集进行图像分类

1.实现说明

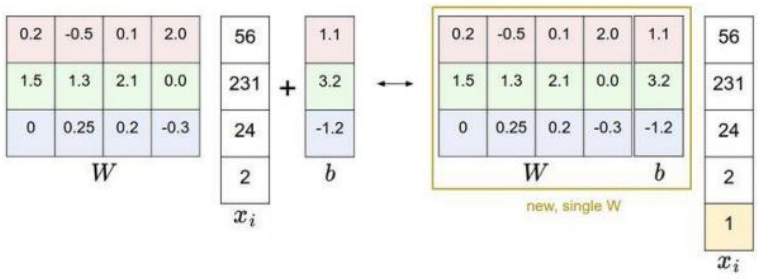
①CIFAR10 数据集

包含 50000 张训练集图片（data_batch_1~data_batch_5 各 10000 张），10000 张测试集图片（test_batch 中），每张图片分别属于 10 个类别中的 1 个。每张图片为 32*32 的 RGB 通道图片。要求在训练集上对分类器进行训练，在测试集上检验分类器的正确率。

②SVM 算法的原理：

SVM 是一种线性分类器。其构造出一个矩阵，计算图像中 3 个颜色通道中所有像素的值与权重的矩阵乘，从而得到分类分值。根据我们对权重设置的值，对于图像中的某些位置的某些颜色，函数表现出喜好或者厌恶（根据每个权重的符号而定）。

其中矩阵的规模为 10*(32*32*3)，即 10 行 3072 列，其中每一行代表了对某个类别的识别模板（或者认为是将该类与其他类分隔开的位于高维空间中的超平面），每一列构成了对检测图像特定位置和特定颜色通道的权值（32*32*3 代表了将 32*32 大小的 RGB 通道图片展平成 1 位列向量）。调整矩阵中的权值将使得高维空间中的超平面旋转，但始终经过原点，因此还需要在矩阵中加入平移的因素。对矩阵增加一列，同时对每张图像的列向量增加一行常数值 1，从而使得新矩阵和图像向量进行点积运算时相当于在原矩阵和图像向量进行点积运算之后又将所得分数进行了偏置（即高维空间中超平面的平移），如下图所示：



通过矩阵与图像向量的点积，可以计算出图像在每一类上的得分。想要使得每张图像在正确类别上的得分比其他类别上的得分均高出一个特定阈值，若不能高出该阈值，则可视为产生了损失，损失函数定义为：

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta)$$

其中 s_{yi} 为在正确类上的得分，s_j 为在其他类上的得分，△为阈值。训练矩阵的最终目的是将所有图像的该损失函数地平均值最小化，即尽可能满足每张图像在正确类别上的得分均比其他类别上的得分高出一个特定阈值的目标。

③正则化：

为了防止构造出的矩阵中权重过于复杂（即矩阵的特殊性非常高）而使得其在测试集上出现过拟合的现象，需要将矩阵的复杂程度纳入到损失函数中，对过于复杂的矩阵进行惩罚，以求得到尽可能简单、通用的分类矩阵。因此在损失函数中加入正则项，本次作业中采用 L2 正则化：

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

最终得到的损失函数为：

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

其中 λ 为正则系数，属于超参数。

⑤实现细节

通过随机梯度下降作为优化方法，使得最终损失函数值尽可能地小。具体细节可以参见代码中的注释，如下图所示（同时另外给出了.py 源文件）：

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

    def loss(self, X, y, reg, delta):
        #   将损失值初始化为0
        loss = 0.0
        dW = np.zeros(self.W.shape)  #  将W的梯度初始化为0
        num = X.shape[0]  # 统计数据集中数据条数
        scores = X.dot(self.W)  #    用训练出来的矩阵计算各个类比的得分值
        correct = scores[range(num), list(y)].reshape(-1, 1)  # 对每条数据根据正确标签作为下标找出正确分类的评分值，再通过reshape（-1，1）转换成1列
        hinge = np.maximum(0, scores - correct + delta)  # 合页函数计算每条数据的损失值
        hinge[range(num), list(y)] = 0  #  正确分类的分数值在上面的运算之后都变成了delta，现在把他们都变成0
        loss = np.sum(hinge) / num + reg * np.sum(self.W * self.W)  #  计算总损失函数值，并且加入了L2正则化防止过拟合
        num_classes = self.W.shape[1]   #   统计分类的类别数目
        #   计算W的梯度
        temp = np.zeros((num, num_classes))
        temp[hinge > 0] = 1
        temp[range(num), list(y)] = 0
        temp[range(num), list(y)] = -np.sum(temp, axis=1)
        dW = (X.T).dot(temp)
        dW = dW / num + 0.5 * reg * self.W
        return loss, dW

    def train(self, X, y, learning_rate=1e-3, reg=1, delta=1, epoch=2000, batch_size=200):
        #   使用随机梯度下降法对SVM进行训练
        num_train, dim = X.shape
        num_classes = np.max(y) + 1  #  y的值从0到分类的类别数目-1，通过此来统计分类的类别数目
        if self.W is None:
            self.W = 0.001 * np.random.randn(dim, num_classes)  #   使用随机数对权重进行初始化
        for it in range(epoch):
            X_batch = None
            y_batch = None
            #   从训练集中随机选择batch_size个数据出来，replace参数表示抽样之后是否放回
            idx_batch = np.random.choice(num_train, batch_size, replace=False)
            X_batch = X[idx_batch]
            y_batch = y[idx_batch]
            loss, grad = self.loss(X_batch, y_batch, reg, delta)    #   计算损失函数值
            self.W -= learning_rate * grad  #   沿梯度方向下降
            if (it + 1) % 100 == 0:   #   每100次迭代显示一次迭代进度
                print('迭代进度 %d / %d: 损失函数值为 %f' % (it + 1, epoch, loss))
        return

    def predict(self, X):
        y = np.zeros(X.shape[0])    #   各个类别的初始得分
        scores = X.dot(self.W)  #   用训练好的W乘以图像数据X即可得到各个类别的得分
        y = np.argmax(scores, axis=1)   #   得分最高的类视为最终识别结果
        return y
```

2.运行说明

本次作业通过在 Visual Studio Code 使用 Anaconda 环境对编写的代码进行解释执行，

其中使用到的 numpy 和 pickle 库均已在 Anaconda 环境中安装。通过控制台命令 python -u “文件路径\SVM.py” 可以对源程序进行解释执行。对测试集上的每张图像，使用 SVM 算法对其作出分类预测。其中 SVM 分类器将在测试开始之前通过在训练集上的多次迭代，训练出用以判定图像类别的矩阵。运行过程中每迭代 100 次将显示出迭代进度。

相比于 KNN 分类器，SVM 分类器花费更多的时间在训练集上训练模型，而对于每次在测试集上进行的分类预测，则只需要进行简单的矩阵乘法即可得到结果。因此 SVM 比 KNN 更具有实用价值。且从最终结果来看，SVM 分类器的分类正确率要高于 KNN 分类器。

3.运行结果

保持训练集和测试集为原规模，运行约 20s 后将运行结束，最终准确率为 37.78%。如下图所示：

```
(base) PS F:\VScodeWorkplace\Py\LearnPytorch\MyLearnPytorch> python -u "f:\VScodeWorkplace\Py\LearnPytorch\MyLearnPytorch\SVM.py"
开始训练SVM...
迭代进度 100 / 2000: 损失函数值为 950.418321
迭代进度 200 / 2000: 损失函数值为 574.374059
迭代进度 300 / 2000: 损失函数值为 348.867444
迭代进度 400 / 2000: 损失函数值为 212.691015
迭代进度 500 / 2000: 损失函数值为 130.255928
迭代进度 600 / 2000: 损失函数值为 81.170701
迭代进度 700 / 2000: 损失函数值为 51.033041
迭代进度 800 / 2000: 损失函数值为 33.224756
迭代进度 900 / 2000: 损失函数值为 22.988195
迭代进度 1000 / 2000: 损失函数值为 15.920094
迭代进度 1100 / 2000: 损失函数值为 12.077379
迭代进度 1200 / 2000: 损失函数值为 9.603882
迭代进度 1300 / 2000: 损失函数值为 8.088829
迭代进度 1400 / 2000: 损失函数值为 7.438872
迭代进度 1500 / 2000: 损失函数值为 6.821322
迭代进度 1600 / 2000: 损失函数值为 6.402916
迭代进度 1700 / 2000: 损失函数值为 6.510678
迭代进度 1800 / 2000: 损失函数值为 6.754002
迭代进度 1900 / 2000: 损失函数值为 6.128543
迭代进度 2000 / 2000: 损失函数值为 6.099090
训练完成！
在测试集上的准确率为：0.377800
```

后续改进

对作业二中使用 SVM 对 CIFAR10 数据集进行分类的部分进行了改进。通过引入方向梯度直方图（HOG）特征提取，将原模型的测试集预测正确率由 38%提升到了 52%

一. HOG 特征提取

1. 作业二中使用数据图片各个像素颜色的 RGB 值作为 SVM 的输入，以此作为区分不同类别物体图像的依据来训练权值矩阵 W（含截距信息）。这种做法传统而简便，对于 32*32 的 RGB 通道图像而言，每个图像对应 3072 个分量的特征向量，同时由于训练集有 50000 张图片，能够为 SVM 提供足够的训练数据。

而仅仅通过颜色作为判定图像类别的依据过于薄弱，对于那些有着大面积相似颜色而形状大相径庭的图片，分类效果显然不够好。因此考虑使用图像的其他特征来作为判定图像类别的依据。

2. 基本思想：图像的内容应该通过图像的形状来得到更好的反应，而图像局部目标的表象

和形状能够被梯度或边缘的方向密度分布很好地描述。因此希望能够利用 CIFAR10 数据集中各个图像的边缘信息来估计反映事物的轮廓，从而作为判断图像类别的主要依据。本质是统计图像中各个位置的梯度信息。

3. 实现方法：将图像划分成一个个的小连通域，在每个小连通域中统计各个像素位置的梯度方向，按照梯度的幅值作为频率，统计出小连通域中的方向梯度直方图。之后由小连通域的方向梯度直方图信息逐渐合并出较大范围内的连通区域的方向梯度直方图，从而得到整张图像的方向梯度直方图信息，反映出图像的总体边缘分布。

4. 具体步骤：

①以 CIFAR10 数据集中 32*32 的 RGB 图像为例，首先将彩色图像转化为灰度图像，便于之后求解各个像素位置的梯度信息。

②对于每个像素位置，通过如下方法计算水平和方向上的（灰度值）梯度：

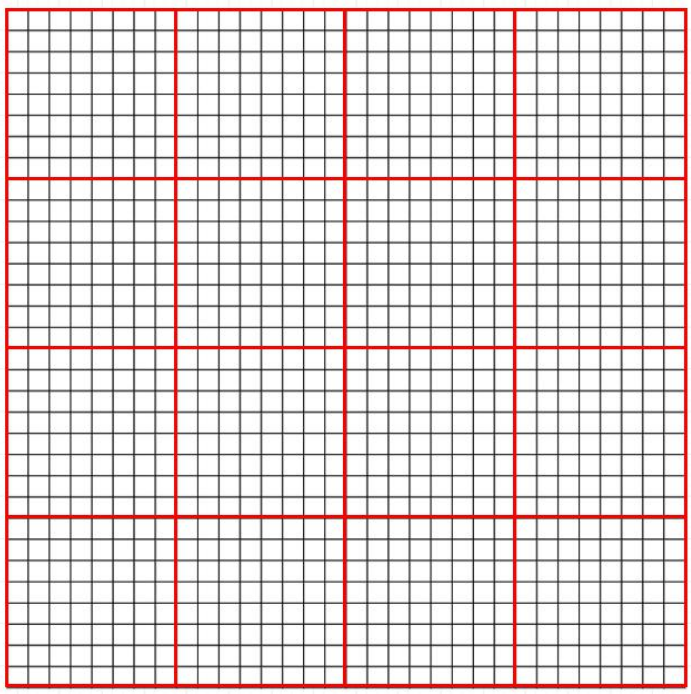
$$G_x(x,y) = H(x+1,y) - H(x-1.y)$$
$$G_y(x,y) = H(x,y+1) - H(x,y-1)$$

其中 $G_x(x,y)$ 代表水平方向上的梯度，$G_y(x,y)$ 代表垂直方向上的梯度，$H(x,y)$ 代表对应像素的灰度值。

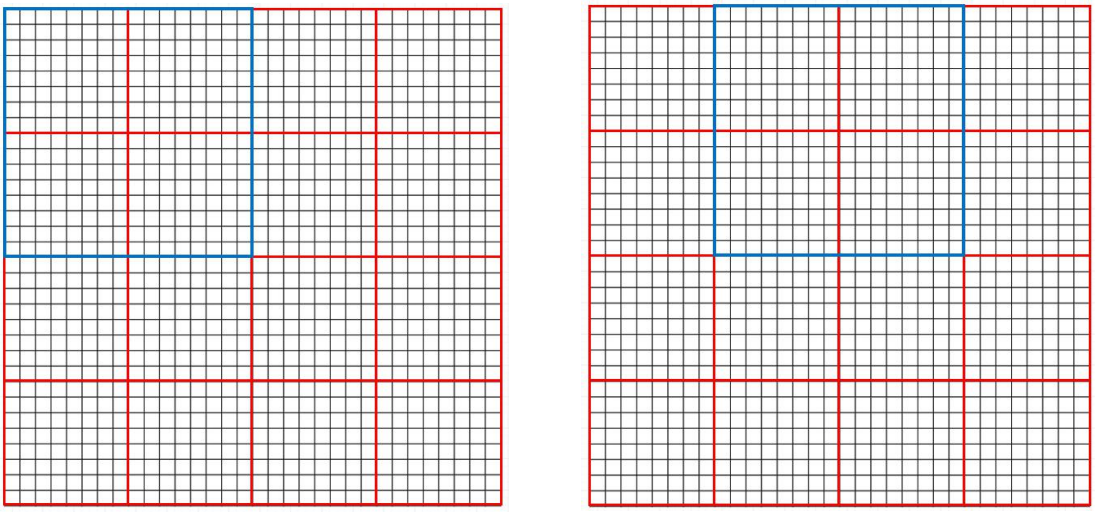
由此可以计算出对应像素位置处的梯度幅值和梯度方向分别为：

$$G(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2}$$
$$\alpha(x,y) = \tan^{-1}(\frac{G_y(x,y)}{G_x(x,y)})$$

③将图像进行划分，每 8*8 个像素为一个单元（cell），且单元为无重叠平铺；每 2*2 个单元为一个块（block），且块为步长为 1 的有重叠平铺，如下图所示：

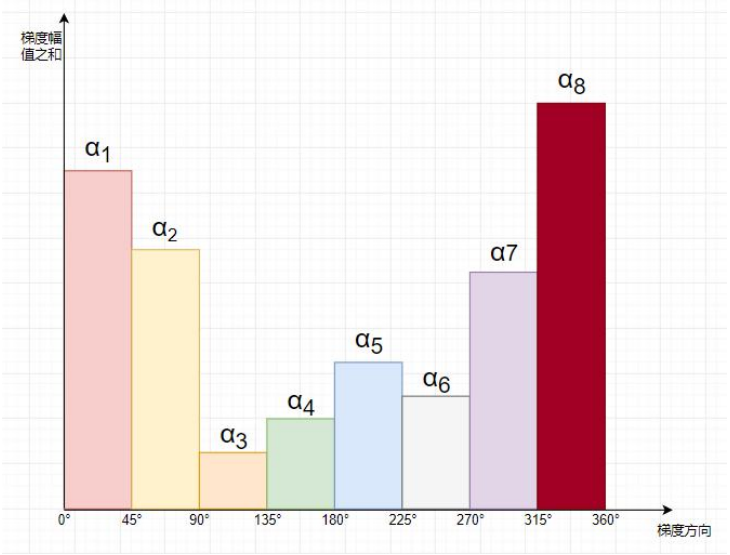


每个红框代表一个 cell



每个蓝框蓝标一个 block，以上示意步长为 1 的可重叠蓝框可能分布情况
（对于 32*32 的图像共 9 种，以上展示了其中的 2 种）。

④对于②中得到的梯度幅值和梯度方向数据，首先按照 cell 为单位统计方向梯度直方图。将 0~360° 划分为 8 个 45° 的区间并编号为 1~8，对于 cell 中每个像素，当其梯度方向位于第 i 个区间且其梯度幅值为 a 时，就在其所属 cell 的方向梯度直方图中，将频数加上 a，得到的方向梯度直方图示意如下所示：



其中 α_1~α_8 构成该 cell 的特征向量($\alpha_1, \alpha_2, \ldots, \alpha_8$)。

对于每个 block，将其包含的所有 cell（此处为 4 个）的特征向量串联起来，即可得到该 block 的特征向量($\alpha_1, \alpha_2, \ldots, \alpha_{32}$)。注意到由于 block 是可重叠的，因此每个 cell 的特征向量可以被多个 block 使用，作为其特征向量的一部分。

由于实际图像中梯度幅值的变化和差异可能很大（受局部光照等因素影响），因此需要对每个 block 的特征向量进行归一化处理，即：

$$\alpha'_i = \frac{\alpha_i}{\sqrt{\sum_{i=1}^{32} \alpha_i^2}}$$

所有 block 的特征向量再次串联，即可得到整张图像的 HOG 特征提取结果。

5.小结：相比于 HOG 特征提取前每张图像的的 3072 维特征向量，HOG 特征提取后每张图片只有 32*9=288 维特征向量。表面上看可供训练的数据减少了，但由于提取后的特征包含边缘信息，更能反映出图像中物体的轮廓，所以实际分类效果会变好。

6.所编写的 HOG 特征提取函数代码如下所示：

```
def get_hog_feat(self, image, stride=8, orientations=8, pixels_per_cell=(8, 8), cells_per_blo
ck=(2, 2)):
        # HOG 特征提取过程中每个 cell 的规模
        cx, cy = pixels_per_cell
        # HOG 特征提取过程中每个 block 的规模
        bx, by = cells_per_block
        sx, sy = image.shape
        # 分别计算水平和垂直方向上 cell 的数量
        n_cellsx = int(np.floor(sx // cx))
        n_cellsy = int(np.floor(sy // cy))
        # 分别计算水平和垂直方向上 block 的数量
        n_blocksx = (n_cellsx - bx) + 1
        n_blocksy = (n_cellsy - by) + 1
        # 对于图像的每个位置，将该处的水平和垂直梯度初始值置为 0
        gx = np.zeros((sx, sy), dtype=np.float32)
        gy = np.zeros((sx, sy), dtype=np.float32)
        eps = 1e-5
        # 对于图像的每个位置，需要存储梯度对应的方向和幅值，因此需要长度为 2 的第三维
        grad = np.zeros((sx, sy, 2), dtype=np.float32)
        for i in range(1, sx-1):
            for j in range(1, sy-1):
                # 计算水平梯度值
                gx[i, j] = image[i, j-1] - image[i, j+1]
                # 计算垂直梯度值
                gy[i, j] = image[i+1, j] - image[i-1, j]
                # 计算梯度方向（使用弧度制表示）
                grad[i, j, 0] = np.arctan(gy[i, j] / (gx[i, j] + eps)) * 180 / math.pi
                if gx[i, j] < 0:
                    grad[i, j, 0] += 180
                grad[i, j, 0] = (grad[i, j, 0] + 360) % 360
                # 计算梯度幅值
                grad[i, j, 1] = np.sqrt(gy[i, j] ** 2 + gx[i, j] ** 2)
```

```
        # 准备求各个 block 的方向梯度直方图并进行归一化
        normalised_blocks = np.zeros((n_blocksy, n_blocksx, by * bx * orientations))
        for y in range(n_blocksy):
            for x in range(n_blocksx):
                # 求每一个 block 中的方向梯度直方图
                block = grad[y*stride:y*stride+16, x*stride:x*stride+16]    # 分离出一个 block
                hist_block = np.zeros(32, dtype=np.float32) # 初始化 block 内方向梯度直方图为 0
                eps = 1e-5
                for k in range(by):
                    for m in range(bx):
                        cell = block[k*8:(k+1)*8, m*8:(m+1)*8]  # 分离出一个 cell
                        hist_cell = np.zeros(8, dtype=np.float32)   # 初始化 cell 内方向梯度直方
图为 0

                        for i in range(cy):
                            for j in range(cx):
                                n = int(cell[i, j, 0] / 45) # 计算梯度方向是 8 个方向中的哪一个
                                hist_cell[n] += cell[i, j, 1] # 对应方向加上梯度的幅值，即计算
cell 内方向梯度直方图项
                        # 将 cell 内的方向梯度直方图填入数组中的对应位置，即将 cell 的梯度直方图串联起
来得到 block 内的方向梯度直方图
                        hist_block[(k * bx + m) * orientations:(k * bx + m + 1) * orientations
 = hist_cell[:]
                # 对每个 block 内的方向梯度直方图进行归一化
                normalised_blocks[y, x, :] = hist_block / np.sqrt(hist_block.sum() ** 2 + eps)
        return normalised_blocks.ravel() # 将所有 block 的梯度方向直方图压缩成一维数组


    def get_feat(self, TrainData, TestData):
        train_feat = []
        test_feat = []
        for data in tqdm.tqdm(TestData):
            image = np.reshape(data[0].T, (32, 32, 3))
            #   将 RGB 通道图像转化为灰度图，便于 HOG 特征提取时计算梯度
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)/255.
            #   对测试集的每张图片进行 HOG 特征提取
            fd = self.get_hog_feat(gray)
            #   将转化为灰度图后的图像和图像标签一起存储为 test_feat 的一个元素，舍弃图像名称
            fd = np.concatenate((fd, data[1]))
            test_feat.append(fd)
        #   将测试集 HOG 特征转为 float 型数组
        test_feat = np.array(test_feat).astype("float")
        #   将测试集 HOG 特征信息存储到文件中，便于后续使用
        np.save("test_feat.npy", test_feat)
        print("Test features are extracted and saved.")
        for data in tqdm.tqdm(TrainData):
```

```
            image = np.reshape(data[0].T, (32, 32, 3))
            #   将 RGB 通道图像转化为灰度图，便于 HOG 特征提取时计算梯度
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) / 255.
            #   对训练集的每张图片进行 HOG 特征提取
            fd = self.get_hog_feat(gray)
            #   将转化为灰度图后的图像和图像标签一起存储为 test_feat 的一个元素，舍弃图像名称
            fd = np.concatenate((fd, data[1]))
            train_feat.append(fd)
        #   将训练集 HOG 特征转为 float 型数组
        train_feat = np.array(train_feat).astype("float")
        #   将训练集 HOG 特征信息存储到文件中，便于后续使用
        np.save("train_feat.npy", train_feat)
        print("Train features are extracted and saved.")
        return train_feat, test_feat
```

二.SVM 分类器对 CIFAR10 数据集进行图像分类

1.实现说明

除了将 SVM 分类器的输入由作业二中每张图像代表颜色信息的 3072 维特征向量更改为本次的代表轮廓信息的 288 维特征向量外，其他部分相同。

2.运行说明

本次作业通过在 Visual Studio Code 使用 Anaconda 环境对编写的代码进行解释执行，其中使用到的 numpy 和 pickle 库均已在 Anaconda 环境中安装。通过控制台命令 python -u“文件路径\SVM_t.py” 可以对源程序进行解释执行。对测试集上的每张图像，使用 SVM 算法对其作出分类预测。其中 SVM 分类器将在测试开始之前通过在训练集上的多次迭代，训练出用以判定图像类别的矩阵。运行过程中每迭代 100 次将显示出迭代进度。

3.数据集说明

本次作业改进，在作业二的和 dataset 目录同级的位置新增了包含有 CIFAR10 数据集对应所有图像的特征提取结果的文件。由于每次对 CIFAR10 数据集完整提取 HOG 特征耗时较长，因此可以直接读取 test_feat.npy 和 train_feat.npy 文件获取已经提取完的 HOG 特征。也可以将这两个文件删除进行重新提取。

4.运行结果

保持训练集和测试集为原规模，运行约 5s 后将运行结束，最终准确率为 51.93%。如下图所示：


终端 SQL CONSOLE: MESSAGES 问题 输出 调试控制台
迭代进度 1100 / 2000: 损失函数值为 1.117515
迭代进度 1200 / 2000: 损失函数值为 1.071227
迭代进度 1300 / 2000: 损失函数值为 1.043168
迭代进度 1400 / 2000: 损失函数值为 1.026173
迭代进度 1500 / 2000: 损失函数值为 1.015864
迭代进度 1600 / 2000: 损失函数值为 1.009615
迭代进度 1700 / 2000: 损失函数值为 1.005830
迭代进度 1800 / 2000: 损失函数值为 1.003532
迭代进度 1900 / 2000: 损失函数值为 1.002142
迭代进度 2000 / 2000: 损失函数值为 1.001298
The testing classification accuracy is 0.519300
The testing cost of time is :4.817539
The Training classification accuracy is 0.518200