计算机视觉大作业 1：图像识别

21121319 刘彦辰

目录

1 数据集分析以及预处理

本次实验的数据集是 1 – 9 的数字图像以及空白图像, 拆分了 1409 张作为训练集, 997 张作为测试集. 训练集中和测试集中 1 – 9 以及空白图像的数量相同. 数据集图像的质量较好, 数字均处于图像中央 (不用对空间变换进行处理) 且比较清晰. 图像大小不统一. 图像没有按照标签放在不同文件夹中, 标签信息可由每张图像的文件名得到.

读取图像时, 直接读取为灰度图 (单一通道), 并统一缩放为 64*64 pixels 的大小. 图像的标签需要对图像文件名进行一些解析才能提取到, 具体代码与实验内容无关不作展示.

2 PCA + KNN

2.1 算法介绍

在上一次实验中, 我根据原理手写 Principal Component Analysis (PCA) + K-Nearest Neighbors (KNN), 实现 Eigenface 算法, 最终在测试集中达到 90% 以上的准确率. 由于上次已经详细介绍了 Eigenface 算法的原理 (并附上了 PDF 文件), 这次就简单介绍一下.

Eigenface 算法本质上是把每个人脸展开当成一个向量, 寻找训练集人脸中的一组特征向量 (当成特征脸空间的基向量). 然后将训练集中的每个人脸的向量投影至基向量构成的空间, 形成一些点, 并打上标签. 预测时, 将新的人脸用同样的方法投影至基向量构成的空间, 寻找与先前训练集投影中欧式距离最短的点, 该点的标签 (这个点原先属于哪个人的脸) 就应该是预测人脸的标签.

算法中的后半部分 “计算测试点在空间中的欧氏距离而寻找最近的训练点” 其实就用到了 KNN 这一算法 (由此可见 KNN 的想法非常简单, 但是如何构建合适的空间, 让训练集投影进去后能够实现最好的分类和分类效果是真正需要处理的事情). 为了构建合适的空间, 我们才使用 PCA 求特征向量作为空间的基向量.

PCA + KNN 自然不局限于检测人脸. 因此本次实验, 在任务一中我尝试用这套方法对 1-9 这些数字图像进行分类.

2.2 算法流程

如上文所述, 我们需要利用 PCA 计算得到的特征向量, 将它们的一部分作为特征空间的基向量. 假设训练集中一共有 M 张图像, 每张图像有 N 个像素点.

首先平展所有图像为一维向量, 求出所有训练集中的平均向量 $\Psi_{(N\times1)}$:

$$\Psi = \frac{1}{M}\sum_{i=1}^{M}\Gamma_i$$

将每个向量减去 Ψ , 得到差向量 $\Phi_{(N\times1)}$:

$$\Phi = \Gamma_i - \Psi$$

计算出协方差矩阵 $C_{(N^2\times N^2)}$, 其中 $A = [\Phi_1, \Phi_2, \dots, \Phi_M]_{(N\times M)}$:

$$C_{(N^2\times N^2)} = \frac{1}{M}\sum_{i=1}^{M}\Phi_i\Phi_i^T = \frac{1}{M}AA^T$$

对用以下公式对协方差矩阵进行特征值分解:

$$AA^T u_i = \lambda_i u_i$$

如果像素点过多，计算 $C_{(N^2\times N^2)}$ 的特征值和特征向量的时间花费是不可接受的．这种情况下，我们可以选择计算 A^TA 的特征值和特征向量，然后转化为 $C_{(N^2\times N^2)}$ 的特征值和特征向量．推导过程如下:

$$A^TAv_i = \mu_i v_i$$
$$\Rightarrow AA^TAv_i = \lambda_i Av_i$$
$$\Rightarrow u_i = Av_i \text{ and } \lambda_i = \mu_i$$

取最大的前 K 个特征值对应的特征向量作为基向量，接下来利用以下公式将之前的所有差向量投影到特征空间:

$$w_i = u_i^T\Phi, \|u_i\| = 1$$

训练集可以重新表示为:

$$\Omega = [w_1, w_2, \dots, w_K]^T$$

对于每个测试图像，用与训练集一样的方式投影到特征空间中，重新表示为 Ω'，然后用以下公式计算训练集中每个图像离测试集中欧式距离最近的图像:

$$e_r = \min_i \|\Omega - \Omega'\| \text{ where } \|\Omega - \Omega'\| = \sum_{i=1}^{K}(w_i - w_i')^2$$

2.3 关键代码

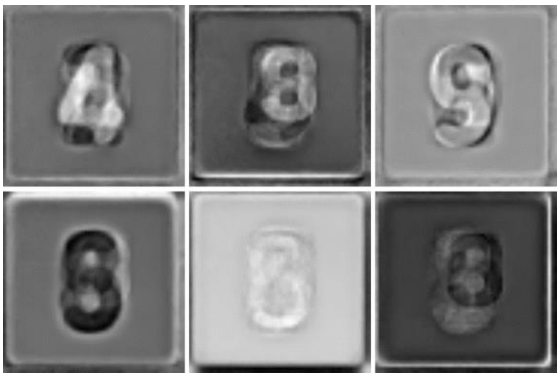
以下代码实现用 PCA 计算出训练集中的基向量和平均向量:

Python Code to Get Base Vectors

```
def getBaseVecs (images2D, K):
    # Calculate the average img
    averageimg = np.mean(images2D, axis=0)
    averageimg = averageimg.astype('float32')
    # Calculate the covariance matrix ([N x N]) of the result mat
    covMat = np.cov(images2D)
    # Calculate the eigenvalues and eigenvectors of the covariance matrix
    eigenValues, eigenVecs = np.linalg.eig(covMat)
    # Calculate the engine imgs of original images
    eigenVecs = np.dot(eigenVecs, images2D - averageimg)
    # Get the indexes of the K largest eigenvalues
    indexes = np.argsort(eigenValues)[::-1][:K]
    # Get the K largest eigenvalues
    eigenValues = eigenValues[indexes].copy()
    # Get the K largest eigenvectors, i.e., the base vectors
    eigenVecs = eigenVecs[indexes].copy()
    # Orthogonalize the engine imgs
    orth_eigenVecs, _ = np.linalg.qr(eigenVecs.T)
    orth_eigenVecs = orth_eigenVecs.T
    # Save the orthogonalized engine vecs and average vec into a .npy file
    np.save('orth_engineVecs.npy', [orth_eigenVecs, averageimg])
```

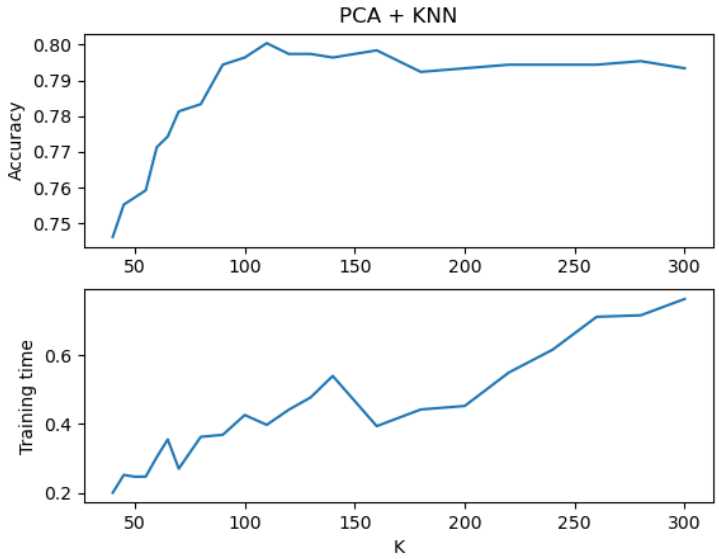
2.4 性能分析

选定基向量数量为 40，下图罗列了其中部分特征向量.



Pic-2-1

Pic-2-1 展示了六张特征向量转化的图片. 这些图片经过了一些特殊处理以增加可见性（原始图像可见性低是因为特征向量经过正交化处理，直接显示亮度和区分度非常低).



Pic-2-2: Training Time and Testing Accuracy of PCA + KNN

Pic-2-2 展示了不同 K 取值下的训练时长和准确度. 可以发现，当 K 取至 100 后，模型的预测准确度基本维持在 80% (并不太高)，但是训练时间一直在增加. 这是由于当基向量数量增多，训练集中的样本点投影到特征空间所耗费的时间会越来越长.

3 HOG + SVM

3.1 算法介绍

当想要检测具有许多一致的内部特性且不受背景影响的对象时，上节所说的 PCA + KNN 算法非常有用，因为例如“人脸”有许多不受图像背景影响的一致的内部特征（眼睛，鼻子和嘴巴). 然而，当试图进行更一般的对象识别时，这些类型的算法并不能很好地工作；原因是物体的内在特征不像人脸的特征那样一致，每个物体形状可能都不同，因此我们需要一些能够更全面地描述一个人的特征.

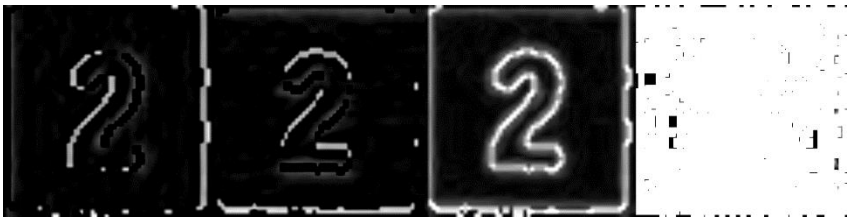
在一副图像中，即使我们不知道局部目标对应的梯度和边缘的位置，它们外表和形状的局部梯度的分布也很好描述（因为梯度主要存在于边缘或角落的地方).

与 PCA 不同，在 HOG 特征描述符中，梯度方向的分布（也就是梯度方向的直方图）被视作特征. 图像的梯度（x 和 y 导数）非常有用，因为边缘和拐角（强度突变的区域）周围的梯度幅度很大，并且边缘和拐角比平坦区域包含更多关于物体形状的信息.

HOG 算法的主要目的是对图像进行梯度计算，统计图像的梯度方向和梯度大小. 算法提取的边缘和梯度特征能够很好的抓住局部形状的特点；并且因为对图像做了 Gamma 校正和采用 cell 方式进行归一化处理，对几何和光学变化都有很好的不变性，变换或旋转对于足够小的区域影响很小.

3.2 算法流程

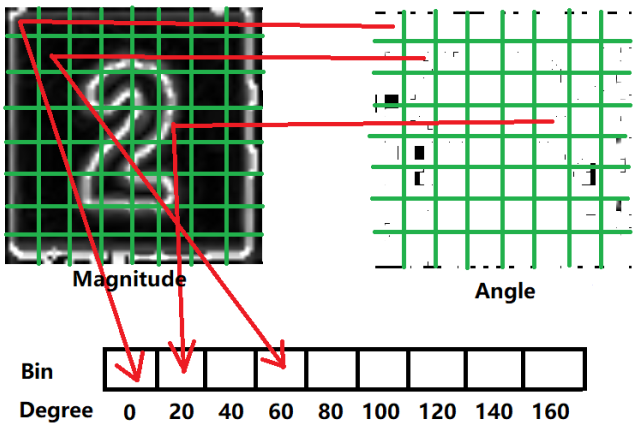
对于每一张输入图像，首先进行 Gamma 校正和颜色归一化，然后进行梯度计算. 以训练集中的某一张“2”为例，梯度图如 Pic-3-1 所示.



Pic-3-1: Grad X, Grad Y, Magnitude and Angle

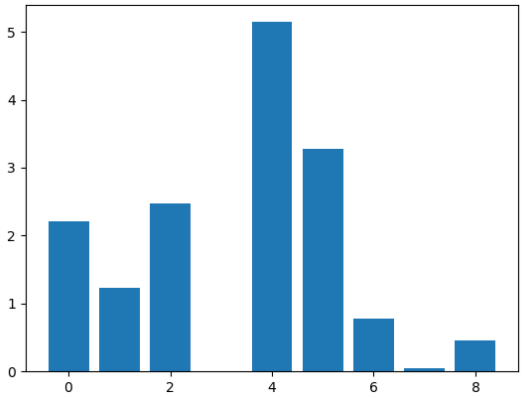
Pic-3-1 从左至右分别是 X-梯度绝对值，Y-梯度绝对值，梯度的幅值和角度. 这里可以发现，数字图像数据集中样本的梯度特征非常明显，这是由于背景和字体颜色都比较单纯，噪声较小.

接着需要计算梯度直方图. 在这一步中，图像被分为若干个 Cell；具体来说，我将我的每个图像（大小为 64*64 pixels）划分为 64 个 8*8 pixels 的 Cells. 将 0-180° 等分为 9 bins，然后对每个 bin 中梯度的贡献进行加权统计.



Pic-3-2

Pic-3-2 表示出了 HOG 的计算过程. 有一个细节是，如果某个像素的梯度角度大于 160°，也就是在 160° - 180° 之间，那么应该把这个像素对应的梯度值按比例分给 0° 和 160° 对应的 bin.



Pic-3-3: Histogram of Gradients

得到一个如 Pic-3-3 的直方图后需要以 2*2 Cell 为一组作为一个 Block，通过滑动窗口对梯度进行归一化处理. 归一化处理能够减少光照变化带来的影响.

处理完后，合并特征，获得多维 HOG 特征，可以放入 SVM 进行分类.

另外，HOG 特征本身不支持旋转不变性与多尺度检测，但是通过构建高斯金字塔实现多尺度的开窗检测就会得到不同分辨率的多尺度检测支持.

3.3 关键代码

由于手搓 HOG + SVM 确实有点困难，我就自己写了对单张图片 HOG 特征的提取代码.

Python Code to Get HOG Features

```
1   img = test_imgs[120]
2   # Normalize
3   img = np.float32(img)/255.0
4   # Calculate grad x and grad y
5   gx = cv2.Sobel(img, cv2.CV_32F, 1, 0, ksize=1)
6   gy = cv2.Sobel(img, cv2.CV_32F, 0, 1, ksize=1)
7   # Calculate magnitude and angle
8   mag, ang = cv2.cartToPolar(gx, gy, angleInDegrees=True)
9   # Calculate bin
10  bin_n = 9
11  bin = np.int32(bin_n*ang/360.0)
12  # Calculate histogram
13  bin_cells = []
14  mag_cells = []
15  cellx = celly = 8
16  for i in range(0, img.shape[0], celly):
17      for j in range(0, img.shape[1], cellx):
18          bin_cells.append(bin[i:i+celly, j:j+cellx])
19          mag_cells.append(mag[i:i+celly, j:j+cellx])
20  hists = [np.bincount(b.ravel(), m.ravel(), bin_n) for b, m in zip(bin_cells,
    mag_cells)]
21  hist = np.hstack(hists)
```

3.4 性能分析

根据 Pic-3-1 展示的 Magnitude 和 Angle 以及 3.1 节所作的介绍，我们可以推断提取数据集 HOG 特征可以很好的区分不同数字的图像. 数据集中的输入图像大小是 64*64 pixels. 我将 Cell 设置为 8*8 pixels, Block 设置为 16*16 pixels, 滑动窗口时 Block 每次移动 8*8 pixels, 并且设置了 9 bins 以表示 0°-180° 的区间. 具体训练时间和参数设置见 Table-3-1 和 Table-3-2.

Table-3-1: Parameters for HOG Descriptor

Input Size	Block Size	Block Stride	Cell Size	Bin Num
64*64 pix	16*16 pix	8*8 pix	8*8 pix	9

Table-3-2: Training Time of SVM and Accuracy

Num for train	Num for test	Training Time	Accuracy
1409	997	0.49s	0.99929

4 CNN

4.1 算法介绍

卷积神经网络 (convolutional neural network, CNN), 是一种专门用来处理具有类似网格结构的数据的神经网络. 卷积神经网络主要包括: 输入层 (Input layer), 卷积层 (Convolution layer), 激活层 (Activation layer), 池化层 (Poling layer), 全连接层 (Full-connected layer), 输出层 (output layer).

如果用全连接神经网络处理大尺寸图像具有三个明显的缺点:

1) 将图像展开为向量会丢失空间信息
2) 参数过多效率低下, 训练困难
3) 大量的参数也很快会导致网络过拟合

与全连接神经网络不同, 卷积具有局部连接, 参数共享这两大特征.

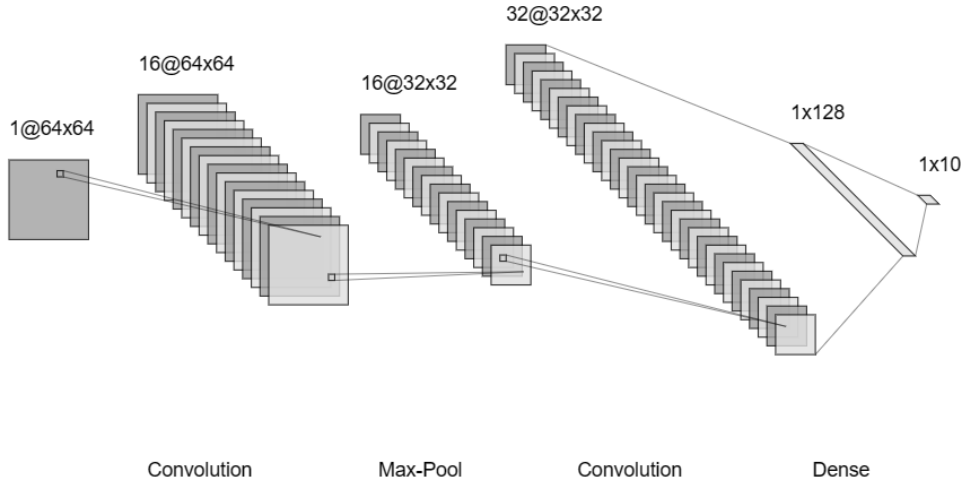
局部连接便是指每个输出通过权值和所有输入相连. 在卷积层中, 每个输出神经元在通道方向保持全连接, 而在空间方向上只和小部分输入神经元相连.

参数共享是指在一个模型的多个函数中使用相同的参数.

如果一组权值可以在图像中某个区域提取出有效的表示, 那么它们也能在图像的另外区域中提取出有效的表示. 也就是说, 如果一个模式出现在图像中的某个区域, 那么它们也可以出现在图像中的其他任何区域. 因此, 卷积层不同空间位置的神经元共享权值, 用于发现图像中不同空间位置的模式. 共享参数是深度学习一个重要的思想, 其在减少网络参数的同时仍然能保持很高的网络容量.

4.2 算法流程

Pic-4-1 展示了我所搭建的神经网络架构. 输入图像是单通道 64*64 pixels 的大小, 经过的第一层卷积层形状为 16*64*64 (激活函数为 ReLU); 然后经过池化层, 减为 16*32*32; 再经过第二层卷积层, 形状为 32*32*32; 最后分别经过 1*12 和 1*10 的全连接层得到输入图像被分为每一类 (共有 10 类) 的概率.



Pic-4-1

为减低内存消耗, 我将 Batch size 设置为 32, 即每次读取 32 张图像放入卷积网络中学习.

在训练时, 选择 pytorch 提供的 Cross Entropy Loss 为损失函数, Adam 为优化器. 在每个 Epoch 中, 利用测试集作为验证集对模型拟合程度进行评估 (验证时不更新模型参数). 更新参数的流程具体为:

1. Clear gradients.
2. Forward pass: Predict outputs of the data using the model.
3. Calculate loss.
4. Backward pass: Compute gradient of the loss with respect to model parameters.
5. Perform a single optimization step (parameter update).

4.3 关键代码

以下为卷积模型的定义代码.

Python Code of Definition of CNN Model

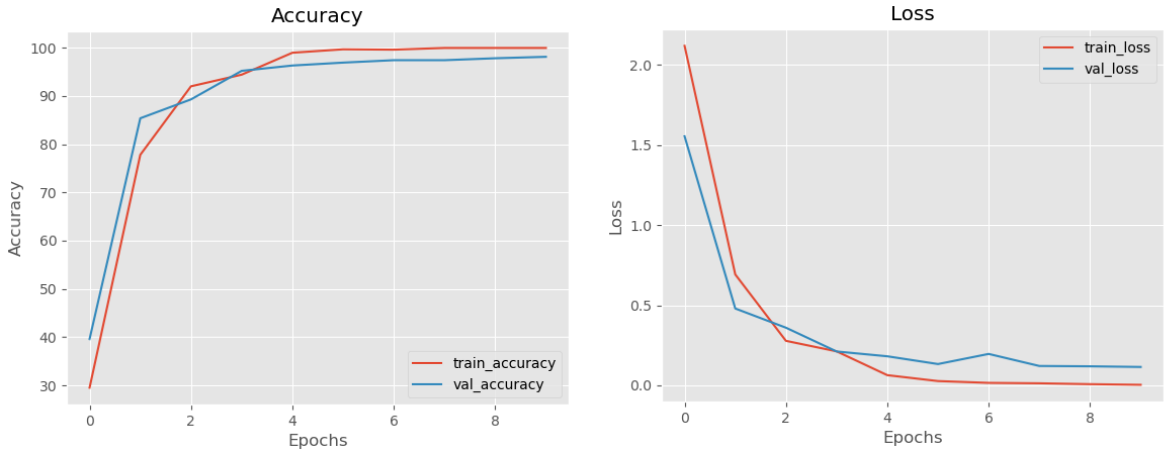
```
class CNN_Model(torch.nn.Module):
    def __init__(self):
        super(CNN_Model, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2)
        # Fully connected layers
        self.fc1 = nn.Linear(32 * 32 * 32, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        # Convolutional layers
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
```

```
16          # Fully connected layers
17          x = x.view(-1, 32 * 32 * 32)
18          x = F.relu(self.fc1(x))
19          x = self.fc2(x)
20          return x
```

4.4 性能分析

由于数据集较少，并且我将模型和数据放在 GPU 中运算，模型的训练时间平均只用了 6.5 秒；而准确率在 4 epochs 之后达到了 98%.



Pic-4-2: Accuracy and Loss in each Epoch

Pic-4-2 展示了模型训练时在每个 Epoch 下的训练准确率，验证准确率，训练损失和验证损失. 可以发现，模型的拟合程度较好，在 6th epoch 之后可能存在略微的过拟合现象. 这是由于数字图像数据集比较简单，特征的提取比较容易.

5 对比分析

Table-4-1: Comparison of Different Methods

	PCA + KNN	**HOG + SVM**	**CNN**
Training Time	0.65s	0.49s	6.5s
Testing Accuracy	0.79	0.99	0.98

Tabel-4-1 展示了三种方法的性能对比. 可以看见, HOG + SVM 训练时间最短，准确度最高, PCA + KNN 准确度最低, CNN 训练时间最长.

这是由于本次实验的数据集 (数字图像) 的边缘特征非常明显，图像质量较高，比较适合 HOG + SVM. 当数据集数量变多，复杂度更高, CNN 可能会有更好的性能. 如上文所说, PCA + KNN 不太适合识别对象内部差异很大的一般识别问题，更好的应用场景在诸如“人脸识别”等的同类且形态差异不大的物体识别.

6 实验总结

本次实验我对 PCA 和 HOG 这两种特征提取算法有了更深的理解, 并且学会了自己写 CNN 并进行模型调参, 有很大收获.