手写数字识别实验报告

手写数字识别，作为深度学习的入门内容及在视觉识别领域中研究较为成熟的领域，能够较好使人同时掌握深度学习及视觉识别的前沿概念。本报告依托MNIST数据集(Modified National Institute of Standards and Technology Dataset)，采用两种方法初探图像特征匹配及基于神经网络之方法，以此达到加深理解课程内容之目的，为数字图像处理与机器视觉之学习画上句号。

原理

1. 图像特征匹配

本实验首先采用一种基于**分块特征提取与模板匹配**的简单图像匹配方法。首先，将每张图像划分为 7×7 个小区域（每个小块包含 $4 \times 4 = 16$ 个像素点）。通过统计每个小块中的像素值，当前景像素（如反转后二值图中的白色像素）数量超过某个阈值（譬如 8 个 —— 也即总像素点的一半）时，将该小块标记为 1，否则标记为 0。最终，每张图像被编码为一个包含 49 维 0-1 元素的特征向量。

在训练阶段，处理训练集（train-part）中的所有图像，提取其特征向量并存入模板库，模板库以 $N \times 49$ 的矩阵形式存储所有 N 个训练样本的特征向量。

在测试阶段，将测试集（test-part）中的每张图像同样转化为 49 维特征向量，并与模板库中的所有模板进行匹配，采用**欧氏距离**作为相似度度量标准，其计算过程如下：

> 设有两个 n 维向量：
>
> $$\mathbf{p} = (p_1, p_2, \ldots, p_n), \mathbf{q} = (q_1, q_2, \ldots, q_n)$$
>
> 它们之间的欧式距离 $d(\mathbf{p}, \mathbf{q})$ 计算公式为：
>
> $$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

容易看出，其值越小，图像之间越为匹配。

2. 深度学习/神经网络方法

为体现多样性，对比不同神经网络结构的表现异同，本报告分别采用两种深度学习方法：全连接网络法/卷积神经网络法。

2.1 全连接神经网络（FCNN）

FCNN 的核心思想是通过**多层线性变换和非线性激活**，逐步提取和组合输入数据中的特征。模型通过训练数据学习权重参数，使得在输入一张图像后，输出层能够给出对应类别的预测概率。在训练过程中，FCNN 使用 **反向传播算法(Backpropagation)** 和 **梯度下降优化算法**，最小化预测结果与真实标签之间的损失函数，从而不断调整网络中的权重和偏置参数。

本实验中采用的 FCNN 网络结构如下，主要由**输入层**、**多个隐藏层**和**输出层**组成，并合理应用了 Dropout 技术以增强模型的泛化能力：

层次	输入 → 输出维度	激活函数 / 作用	备注
输入层	$28 \times 28 \to 784$	无	展平成向量，作为 FC 层输入
第一层 Dropout	$784 \to 784$	无	25% 概率置零，防止过拟合
第一隐藏层	$784 \to 512$	ReLU	缓解梯度消失，增强非线性表达
第二隐藏层	$512 \to 256$	ReLU	提取更高层次特征
第二层 Dropout	$256 \to 256$	无	50% 概率置零，进一步增强鲁棒性
第三隐藏层	$256 \to 128$	ReLU	进一步压缩特征，增强判别性
输出层	$128 \to 10$	log_softmax	输出对数概率，适配 NLLLoss

这一结构规模既小，又非常典型，适合中小型的分类任务。

2.2 卷积神经网络（CNN）

CNN 的核心思想是利用**卷积操作**（Convolution）代替传统全连接神经网络中的部分矩阵乘法操作，从而更好地捕获图像中的**局部模式**（如边缘、角点、纹理等）。在 CNN 中，卷积层通过若干可学习的 **卷积核(或称滤波器)** 在输入特征图上滑动，计算局部加权和，从而产生新的特征图（Feature Map）。卷积核的权重在整个输入图像上共享，这大大减少了模型参数数量，提高了训练效率和泛化能力。

层次	输入 → 输出维度	激活函数 / 作用	备注
第一卷积层	$28 \times 28 \to 26 \times 26 \times 32$	ReLU	卷积核 3×3，提取低层次特征
第二卷积层	$26 \times 26 \times 32 \to 24 \times 24 \times 64$	ReLU	深层特征提取
最大池化层	$24 \times 24 \times 64 \to 12 \times 12 \times 64$	无	降采样，减小特征图尺寸，增强平移不变性
第一 Dropout	不改变尺寸	无	25% 概率置零，防止过拟合
扁平化	$12 \times 12 \times 64 = 9216 \to 9216$	无	转换为全连接层输入

层次	输入 → 输出维度	激活函数 / 作用	备注
第一全连接层	$9216 \to 128$	ReLU	高维特征映射到低维空间
第二 Dropout	$128 \to 128$	无	50% 概率置零，进一步增强鲁棒性
输出层	$128 \to 10$	log_softmax	输出对数概率，适配 NLLLoss

CNN 为现代计算机视觉任务中的主流模型架构，广泛应用于图像分类、目标检测、图像分割、人脸识别等领域。

实验

1. 下载MNIST数据集

torchvision自带MNIST数据集的下载函数，可直接利用其进行下载。

```
train_dataset = datasets.MNIST(root='../data', train=True, download=True,
transform=transform)

test_dataset = datasets.MNIST(root='../data', train=False, download=True,
transform=transform)
```

其和实验指导书中所要求的内容区别在于——实验书提供的下载包中将图像特征识别方法的图像数量限制在了六千张。然而，笔者认为，这样做无法形成有效的两种方法的对照。**因为本报告并不是为了达成其最好效果（笔者也志不在此），而是为了初步学习掌握所学内容、了解各种方法优劣而写的，因此，两种方法采取了同样的七万张图像作为训练集。当然，这似乎导致了神奇的后果，后文会进行讨论。**

2. 图像特征匹配法

首先构建模版库：

```
train_features = []
train_labels = []
for i in tqdm(range(len(train_dataset)), desc="Extracting train features"):
    img, label = train_dataset[i]
    vec = extract_feature(img)
    train_features.append(vec)
    train_labels.append(label)
```

```
    train_features = np.array(train_features)
    train_labels = np.array(train_labels)
```

其中，抽取特征的方式如原理所述，非常地简单粗暴。先将 28×28 图像分为 7×7 块，如若其二值灰度图像中的一压缩块中像素点亮度超过0.5（实际上是127），则将本压缩块点亮，否则置暗：

```
def extract_feature(image_tensor):
    image = image_tensor.squeeze().numpy()
    feature = []
    for i in range(0, 28, 4):
        for j in range(0, 28, 4):
            block = image[i:i+4, j:j+4]
            threshold = (block > 0.5).sum()
            feature.append(1 if threshold > 8 else 0)
            return np.array(feature)
```

随后在测试集中检验模型：

```
test_features = []
test_labels = []
predicted_labels = []

for i in tqdm(range(len(test_dataset)), desc="Matching test samples"):
    img, label = test_dataset[i]
    vec = extract_feature(img)
    test_features.append(vec)
    test_labels.append(label)

    distances = np.linalg.norm(train_features - vec, axis=1)
    min_index = np.argmin(distances)
    predicted_labels.append(train_labels[min_index])

accuracy = accuracy_score(test_labels, predicted_labels)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

最后对自己手写的样本（验证集）进行预测，看看该方法表现如何。

```
transform = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))
])

mnist_train = datasets.MNIST(root='../data', train=True, download=True,
```

```
transform=transform)

template_features, template_labels = build_template_library(mnist_train)

print("正在显示0~9的特征图像...")

raw_train = datasets.MNIST(root='../data', train=True, download=True,
transform=transforms.ToTensor())
show_digit_feature_examples(raw_train)

print("正在批量预测 hand_written/0.x -> 9.x 中的自定义图像...")
image_folder = "hand_written"
process_and_predict_images_in_batches(image_folder, template_features,
template_labels)
```

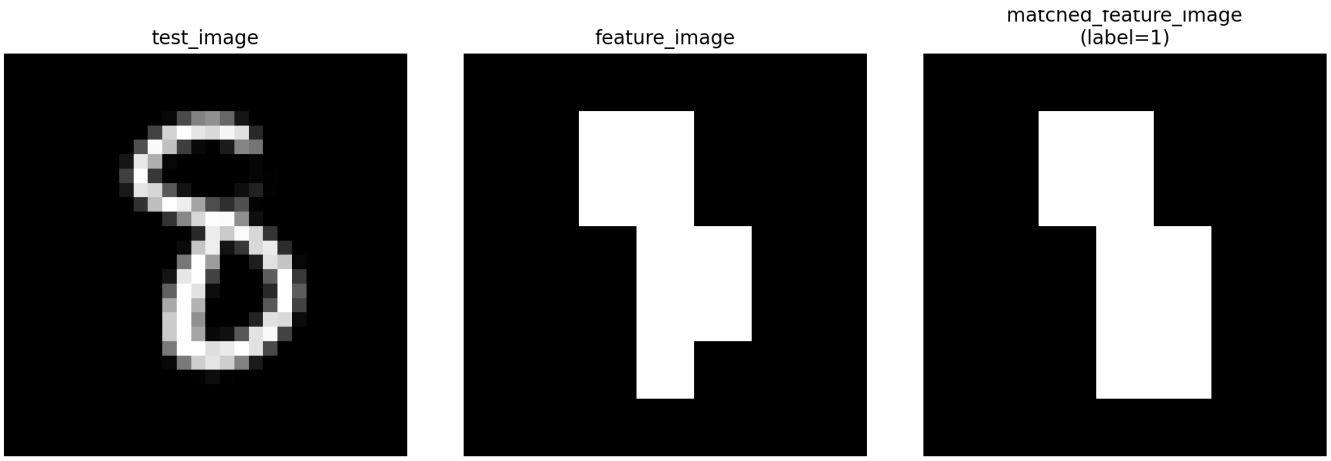
好了。现在就可以来跑结果了。其配准率采用测试集进行检验，由如下方式计算得到：

$$a_{cc} = \frac{Card(True)}{Card(All)}$$

```
Extracting train features: 100%|| 60000/60000 [00:11<00:00, 5279.65it/s]
Matching test samples: 100%|| 10000/10000 [01:45<00:00, 94.97it/s]
匹配准确率: 60.77%
```

……这个可怕的配准率，有种不祥的预感。随便拿一张图像来验证一下。由于手写图像质量的原因，将点亮阈值设置为0.25，只需4个点即可点亮小分区。

```
count = torch.sum(block > 0.25).item() # 阈值调低了一些
bit = 1 if count > 4 else 0
```

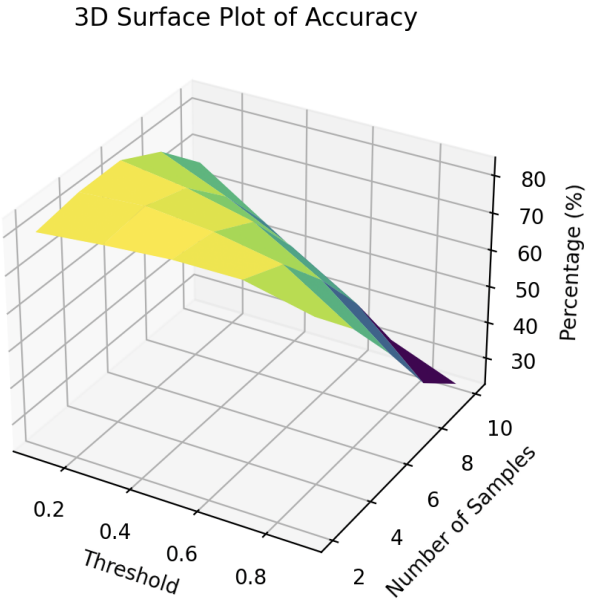


😅 如图可见，虽然特征图像按照规则被很好地提取出来了，然而，由于这样纯粹降采样的划分方法实在是太过粗糙，他意外地匹配上了特征集中一个错误的元素，最终被识别为了1……这便是为什么本方案的识别率仅有 60.7% 。大量的格点信息都被浪费了，以至于发生误匹配的概率居高不下。

那么，改变阈值呢？额……事实证明，有可能变得更糟糕。譬如，将阈值略微调高，让其需要十个格点才能点亮格子（表现出来便是特征图样变得稀疏），训练准确率如下：

```
(base) hch@hchdeMacBook-Pro Final % /usr/local/bin/python3 /Users/hch/Desktop/Dr.Red/ZJU/数字图像/Final/Matching.py
Extracting train features: 100%|██████████| 60000/60000 [00:11<00:00, 5354.65it/s]
Matching test samples: 100%|██████████| 10000/10000 [01:43<00:00, 96.99it/s]
匹配准确率： 42.72%
```

更低了。这自然让人想到，为了调整参数，可不可以将模型表现随着参数的变化画出一张超参数图样。经过粗略而漫长的跑模型过程，得到如下三维图：



其大趋势并不让人意外。从图左下至右上事实上反映的 **都是阈值变化带来的查准率变化** 。阈值越高，匹配得到的点越少，越难以反映数字的基本形状规律，是而百分比下降。而阈值低到一定程度，该点亮的部分业已点亮，特征图不再随着阈值继续降低而变化，因此查准率不再上升。

最优处出现在 $(0.3, 4)$ 左右，大约有 83.5% ，比较令人满意（但可以预见到，仍然可能有非常多的错误，让一些试图在停车场过视觉识别闸的车主非常抓狂）。因此，我们需要有更好、更稳定的方法，来让其应用达到工业级别。

3. 神经网络法

构建FCNN模型如下：

```
class FCNN(nn.Module):
    def __init__(self):
        super(FCNN, self).__init__()
        self.flatten = nn.Flatten()
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)

        self.fc1 = nn.Linear(28 * 28, 512)
```

```
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.dropout1(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return F.log_softmax(x, dim=1)
```

对于CNN，模型改为：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

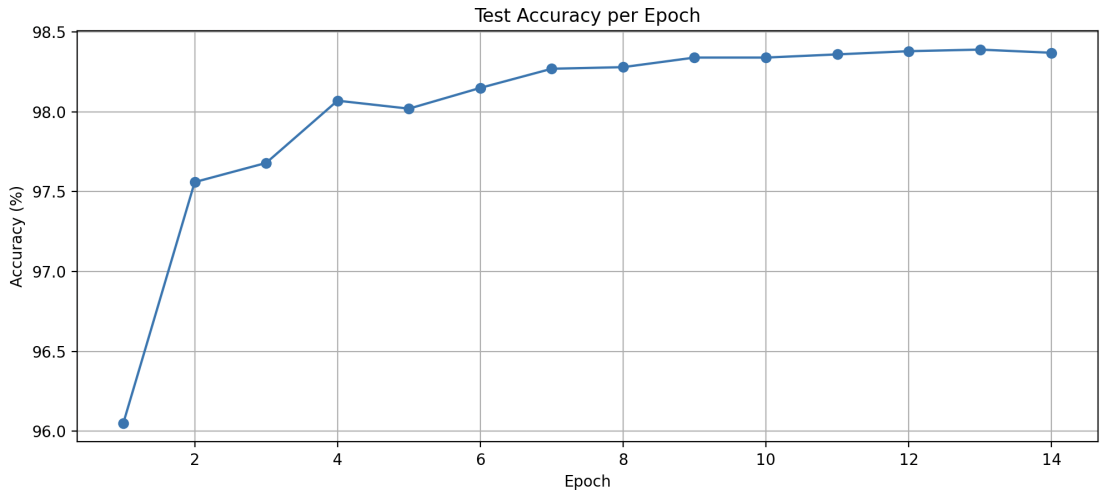
随后只需要分别训练两个模型，观察测试集测准率曲线变化，保存为 `.pt` 文件即可。

```
train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)
```

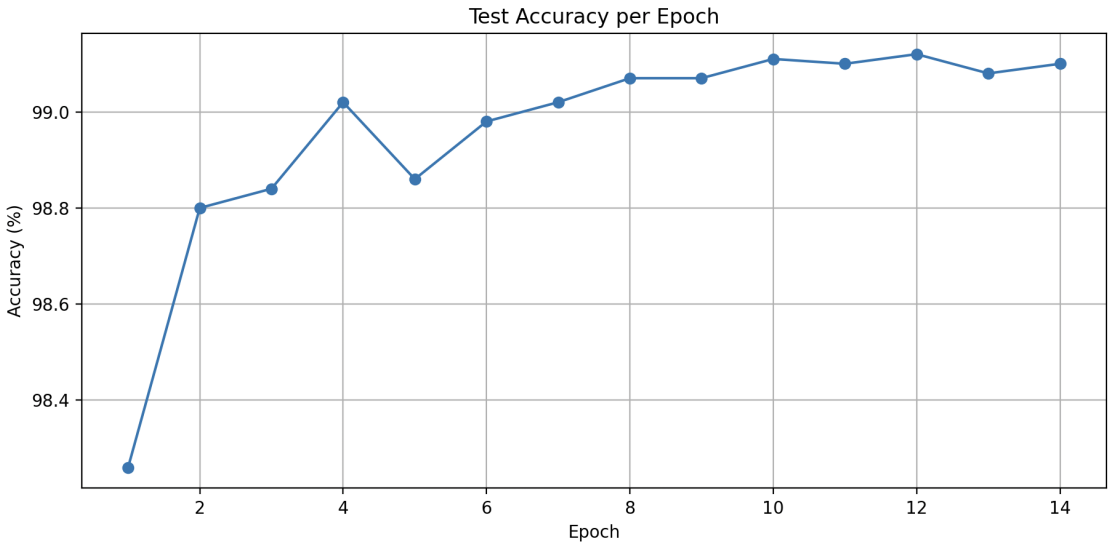
```
model = Net().to(device)
optimizer = optim.Adadelta(model.parameters(), lr=args.lr)
scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)

for epoch in range(1, args.epochs + 1):
    train(args, model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
    scheduler.step()

if args.save_model:
    torch.save(model.state_dict(), "mnist_cnn.pt")
```

训练结果如下：

- FCNN



- CNN

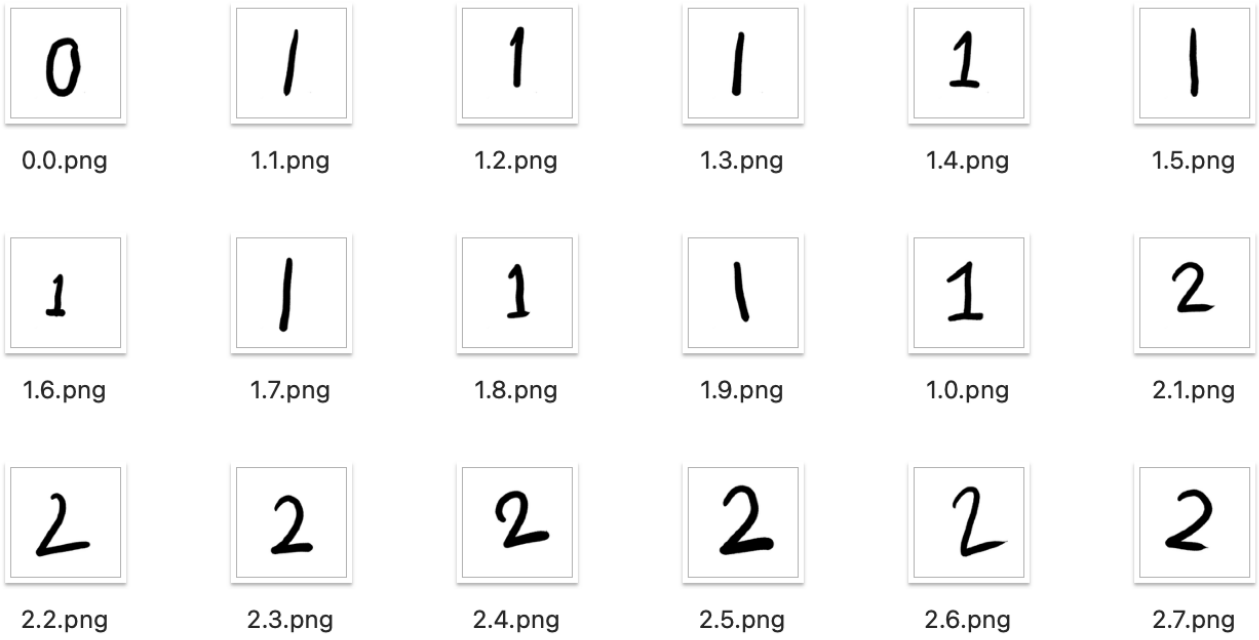


可以看到，两个模型都在十轮训练后准确度收敛，CNN模型的表现（并不意外地）比FCNN好，大约高出 0.8% 。不过，值得注意的是，虽然CNN模型的表现要优于FCNN，但是FCNN的训练速度非常之快，即使对于没有CUDA协力的苹果电脑，单纯使用m1芯片也能在两分钟

左右结束训练，而CNN——即使使用的结构较为简单——却要十分钟之巨。两者的大小也有差别，CNN是FCNN大小的两倍。因此，对于一些空间有限的应用场合（单片机？虽然2.3MB也放不进单片机就是了），选择FCNN不失为一种取舍。

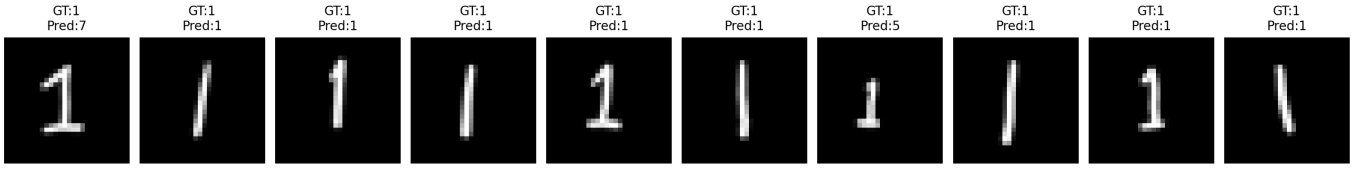
4. 自手写图样的测试

为测试本训练得到的模型的鲁棒性，特写一百张手写数字，按0至9的顺序分为十组，每组十张图片。如图：

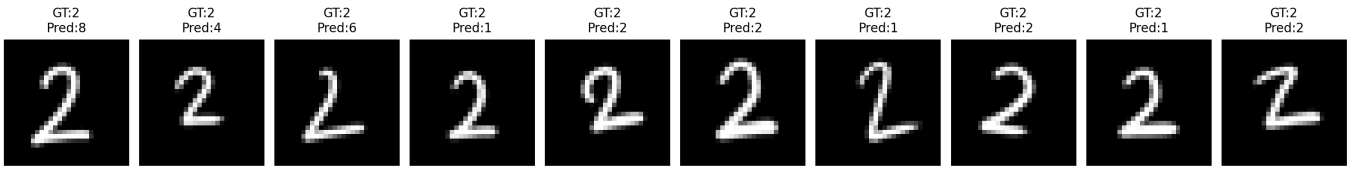


其中，非常多的数字故意写的非常扭曲和歪斜，以达成比较极端的效果，测试模型的边界在何处。

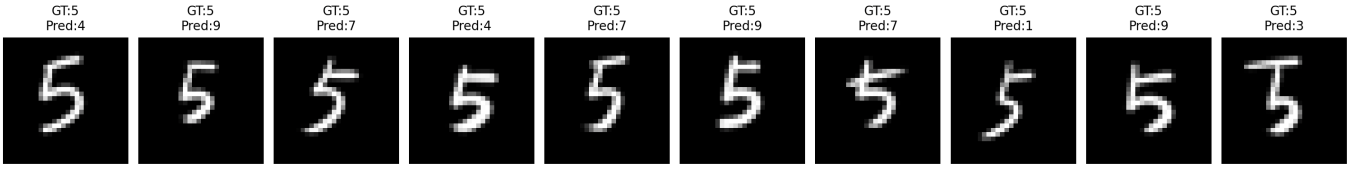
4.1 图像匹配法结果



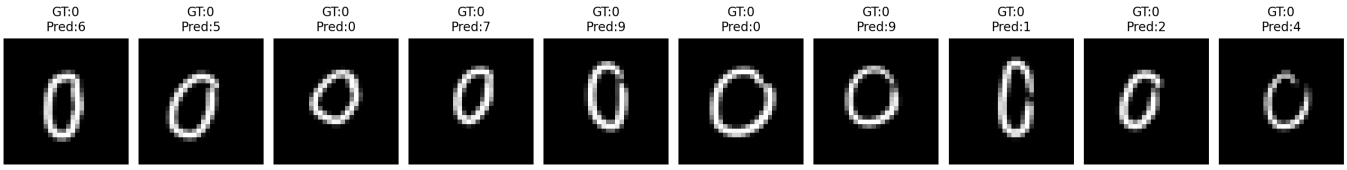
在数字1这样的直筒数字上，图像匹配法表现的很好，虽然偶发抽风，但是总体达到了识别的要求。然而，我们还记得，原本得到的模型训练准确度只有 $60-80\%$ ，因此必然会有抽象的结果诞生，譬如数字二：



这事实上不令人意外，其原因已经在2中分析过。对于这类划分任务，如此粗暴简单的特征划分和匹配方法对于混在一起的特征点表现就应当是极为糟糕的。为了找乐子，我们还可以看看一些群魔乱舞的结果：

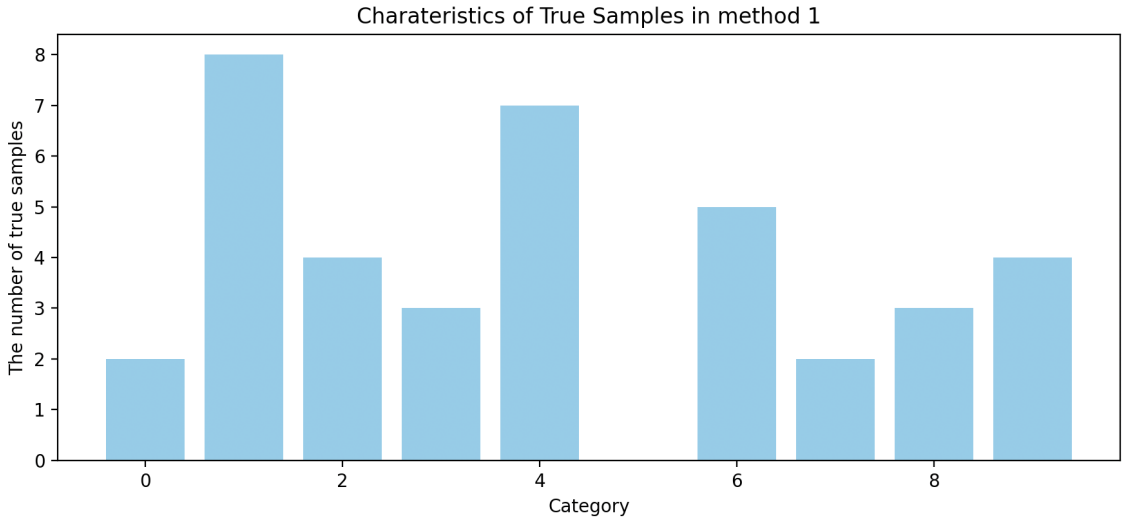


- 5：一个都没识别对。太不令人意外了。



- 0：这相对比较令人意外，但经过对特征图样的观察可以得知，一些别的图样的长相确实与它所生成的特征图很相似。

最终，我们便可以得出一张相对具有统计意义的图像识别柱状图。这充分表达了我们对于如此简单的识别方法生成的在停车场暴怒的车主的歉意。

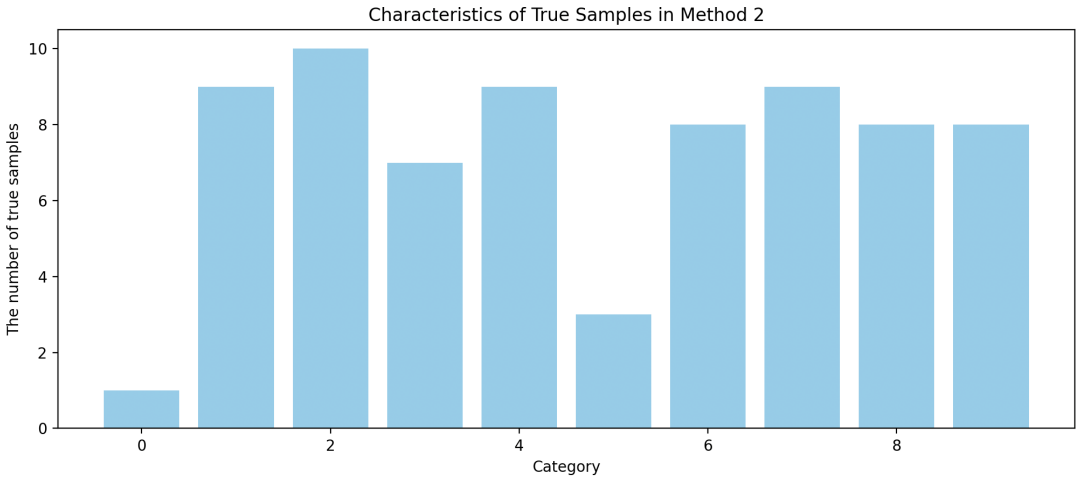


其其比例z检验得到的p值为 4×10^{-9} ，可拒绝 H_0 假设，认为此方法显著优于随机猜测。可以看到，越是乖张的数字，识别的越好。

4.2 神经网络结果

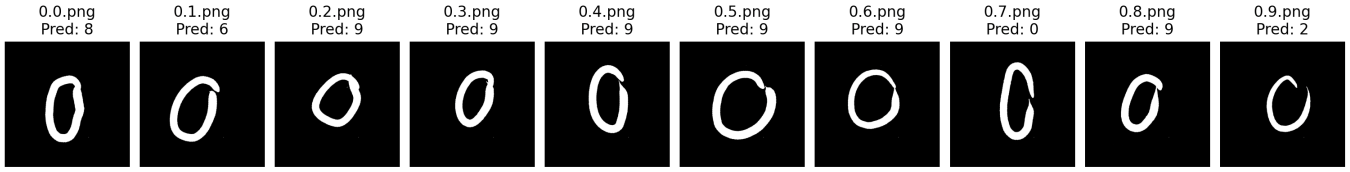
相对地，两个神经网络在此任务上的表现十分出色。其柱状图统计可以证明这一点：

- FCNN

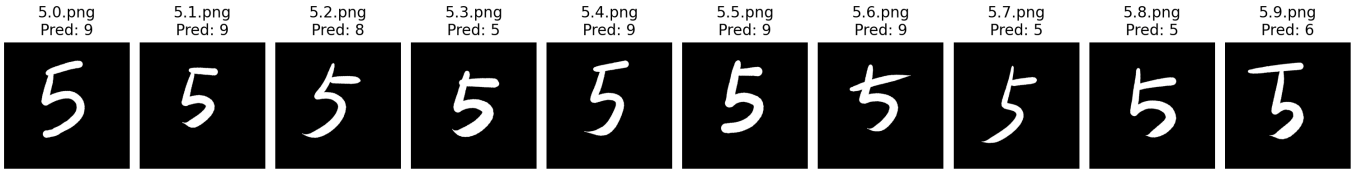


其比例z检验得到的p值为 1.13×10^{-43} ，可拒绝 H_0 假设，认为此方法显著优于随机猜测。

不过，非常诡异的是，这一算法对0和5的识别能力仍然比较差。

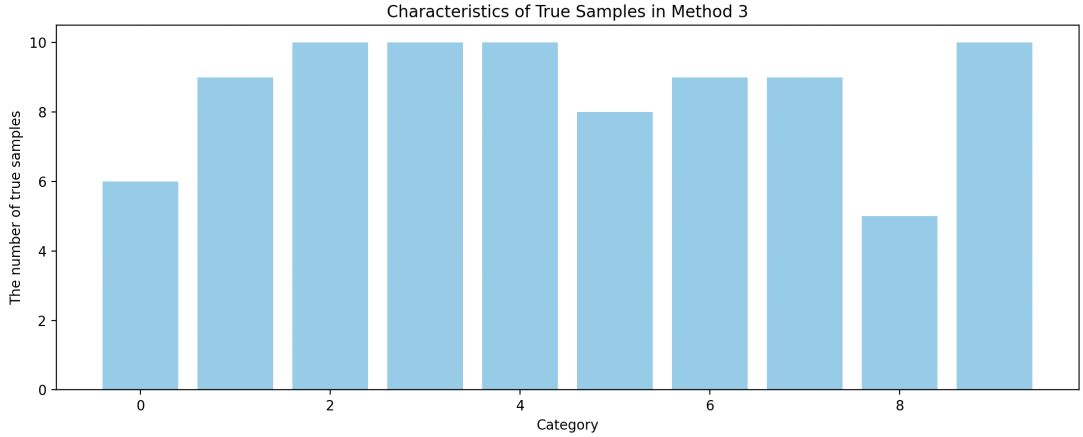


0：被大批量地识别成了9。原因不明。



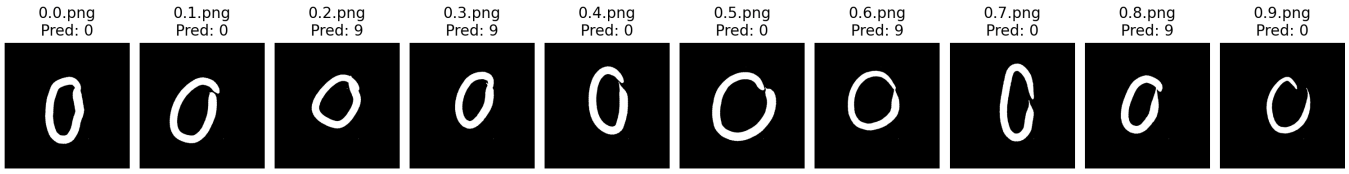
5：同样，被大批量地识别为了9。

- CNN

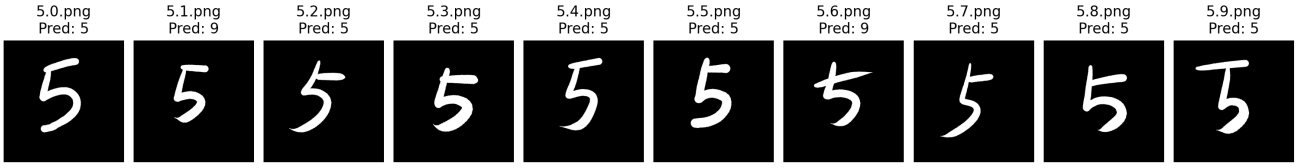


其比例z检验得到的p值为 1.22×10^{-106} ，可拒绝 H_0 假设，认为此方法显著优于随机猜测。

不出意外地，尽管从 **总体** 而言，t检验的p值约为0.1，尚未以达成非常大的提升效果，然而在一些老大难数字上，CNN方法要明显优于FCNN。这显然是由模型的复杂度决定的，体现了“大就是好”的哲学思想。



- 0：9的数量被明显地抑制。



- 5：解决了两任算法都没有办法识别准5的问题，甚至是非常刁钻的最后一个图样。

可以预见的是，如果再对模型进行一定的优化，CNN将有机会达成百发百中的效果。然而，因为现在的对比已经足够明显，且其余变量得到了很好的控制，因此本报告不再改变模型的长相。

5. 后日谈

5.1 选一种别的特征匹配方法

就如2所分析的原因所导致的那样，如此简单的特征匹配的效果人神共愤。那么，有没有什么更好的方法？SIFT也许可以。其原理如下：

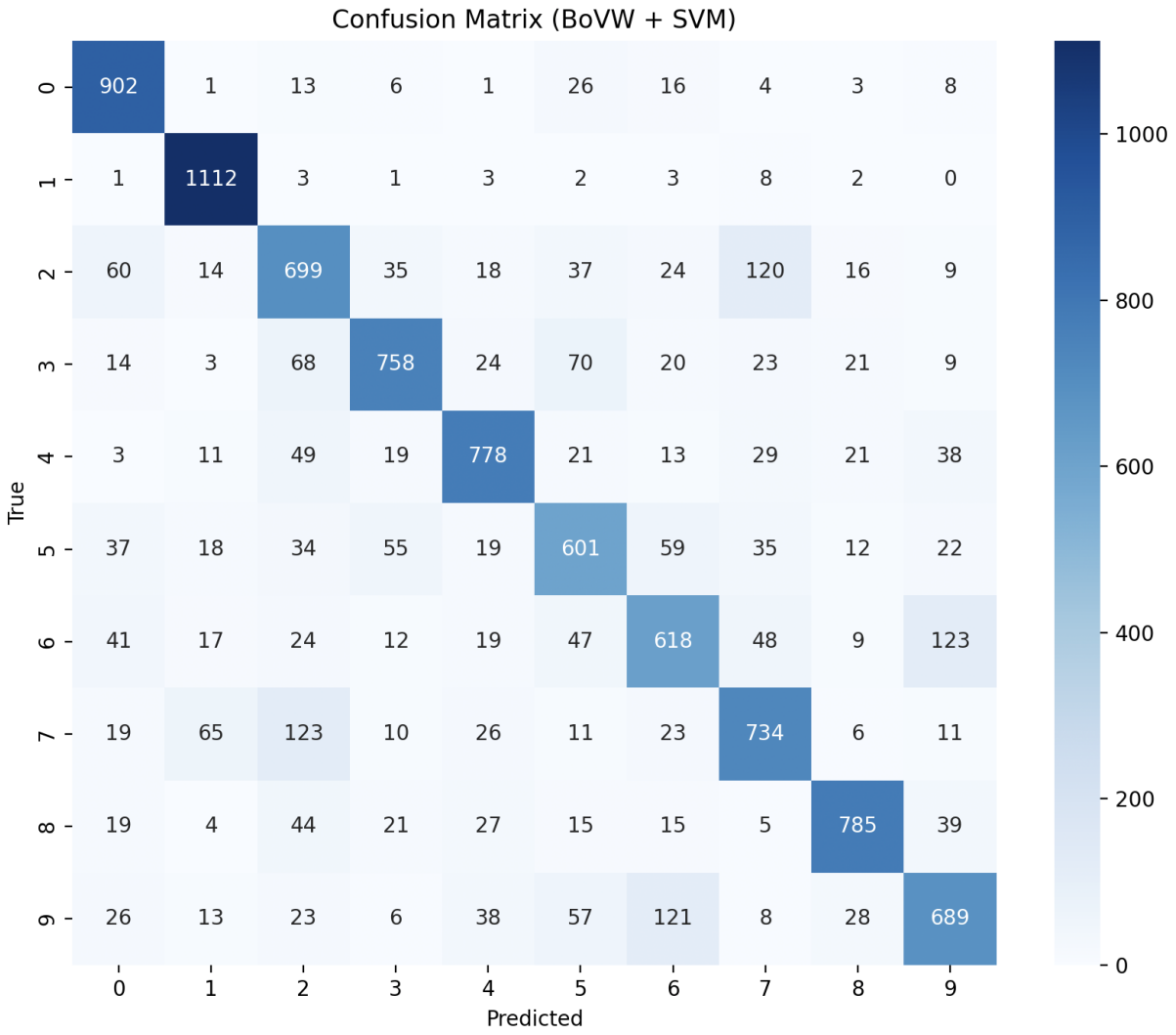
- 关键点检测（Keypoint Detection）
- 关键点精确定位（Keypoint Localization）
- 方向分配（Orientation Assignment）
- 特征描述子生成（Descriptor Generation）
- 特征匹配

听起来挺强的。来试试。

在实验中，笔者采用了SIFT（特征检测）+BoVM（特征直方图生成）+SVM（分类器）的方法进行操作。原因无他，KNN分类得到的结果是倒挂的，仅有 60% 左右，同时也并不容易调整参数。经调整，其最好结果大概在如下参数时达成：

```
n_clusters = 200
svm = SVC(kernel='rbf', C=100, gamma=0.01)
```

约为 77%。 这是一个比较令人满意的结果，鉴于SIFT事实上更适合处理自然图像，而非MNSIT数据集这样低精度的图样。



通过观察混淆矩阵图样，我们也可以发现和普通匹配方法一样的规律。整个矩阵对于1的匹配效果极好，远超别的数字。其余的准确率略多于一半，错误主要集中在一些非常圆润的数字的识别上，譬如6，9，2。

5.2 神经网络——也许一点预处理会有帮助……

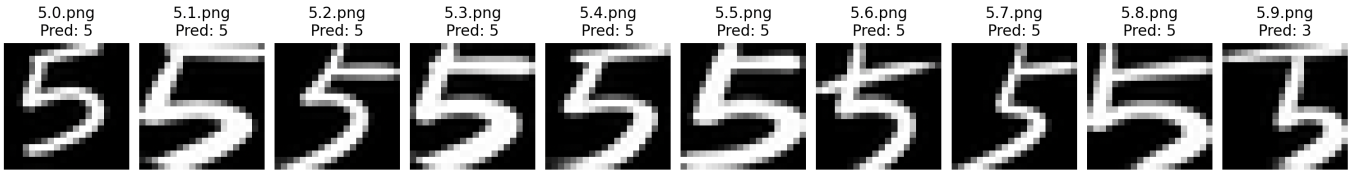
……那么，既然MNIST测试集的查准率可以达到惊人的 99% ，这些没有查准的图样会是什么因素产生的呢？自然可以想到，是预处理的步骤没有做好。MNIST数据集的数字图样是正放居中的，而笔者自己所写的字样显然（是故意的）没有达到这一点。

如何处理？LeCun等人在1998年（是的，你怎么知道我比不上人家三十年前的工作）发表的文献中，提到了一个非常简单的处理方式：

```
def process_image(image_path):
    img = Image.open(image_path).convert("L")
    # optional: Invert if your image is black background
    img = ImageOps.invert(img)
    # Crop + resize + center
```

```
    bbox = img.getbbox()
    img = img.crop(bbox)
    img = img.resize((20, 20), Image.ANTIALIAS)
    new_img = Image.new('L', (28, 28), (0))
    new_img.paste(img, (4, 4))
    # ToTensor + Normalize
    img_tensor = transform(new_img).unsqueeze(0)
    return img, img_tensor
```

通过裁剪空白，缩放为 20×20 并居中填充至 28×28 的方法，字能够以更加延展而对齐的方式充塞四虚，从而提升识别率。事实证明，的确是这样的。譬如，我们来看经过此方法预处理后的图像经手FCNN网络的识别结果：



有了极为惊人效果的提升！除了最后一张图被识别为了3（不能怪他，如果把这张图给我，我也会在3和5之间纠结很久——这张图的意义就在于此）。整体的表现极大优化。

5.3 神经网络参数影响?

理论上，在网络设计和训练过程中，不同的神经网络结构及超参数设置会对最终的识别性能产生显著影响。经查阅资料，适当增加网络深度和宽度可以提升模型对复杂笔迹变化的表征能力，但过深的网络若未加以正则化处理，容易导致 **过拟合** （但是，本篇报告中所采用的模型似乎并没有达到能够讨论过拟合的程度）。此外，优化器的选择对收敛速度和最终精度也有明显影响。在本实验中，我所采用的优化器为：

```
optimizer = optim.Adadelta(model.parameters(), lr=args.lr)
scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
```

但是，采用别的优化器，譬如Adam优化器，可能会达到更好的效果。

另一方面，学习率策略和数据增强在模型泛化能力提升中起到了关键作用。通过采用动态学习率衰减策略，训练过程中的稳定性和最终模型性能均得到改善；同时，合理的数据增强，正如名字所昭示的，也应当可以增强模型对手写数字形态变化的鲁棒性。然而，对于这些方法的讨论超出了本篇报告的内容，因此不予展示。