# 基于模式匹配的数字识别实验报告

## 一、实验目标

本次实验的目标是通过模式匹配的方法，进行数字识别。我们将创建两组不同字体的数字图片，一组作为模板，另一组作为测试数据，然后通过各种方法进行图片匹配，以实现数字识别。

## 二、实验步骤与方案

### 1. 创建数据

我们使用Python的PIL库生成了两组数字图片，一组使用宋体字体，作为模板；另一组使用Furore字体，作为测试数据。

```python

import os

from PIL import Image, ImageDraw, ImageFont

# 创建模板和测试文件夹

if not os.path.exists('templates'):

os.makedirs('templates')

if not os.path.exists('tests'):

os.makedirs('tests')

# 生成宋体的模板图片

for i in range(10):

img = Image.new('RGB', (32, 32), color=(255, 255, 255))

d = ImageDraw.Draw(img)

font = ImageFont.truetype('simsun.ttc', 15) # 请替换为你的宋体字体文件的路径

# 多次绘制文本以产生加粗的效果

for j in range(-1, 1):

for k in range(-1, 1):

d.text((10+j, 10+k), str(i), font=font, fill=(0, 0, 0))

# d.text((10, 10), str(i), font=font, fill=(0, 0, 0))

img = img.convert('L') # 转换为灰度模式

img.save(f'templates/{i}.png')

# 生成Furore的测试图片

for i in range(10):

img = Image.new('RGB', (32, 32), color=(255, 255, 255))

d = ImageDraw.Draw(img)

font = ImageFont.truetype('Furore/Furore.otf', 15) # 请替换为你的Furore字体文件的路径

d.text((10, 10), str(i), font=font, fill=(0, 0, 0))

img = img.convert('L') # 转换为灰度模式

img.save(f'tests/{i}.png')

```



### 2. 方案一：欧式距离匹配

我们首先尝试的是最直接的方法——欧式距离匹配。我们用numpy读取图片文件，然后计算测试图片与模板图片的欧式距离，按照距离进行分类。

```python

import numpy as np

import os

from PIL import Image

def load\_image(path):

return np.array(Image.open(path))

# 第一步：从模板文件夹中读取模板

templates = {}

for i in range(10):

img = load\_image(f'templates/{i}.png')

templates[i] = img

# 第二步：从测试文件夹中读取测试图片，并进行匹配

for filename in os.listdir('tests'):

test\_img = load\_image(f'tests/{filename}')

min\_distance = float('inf')

predicted\_digit = -1

# 与所有模板进行匹配

for digit, template in templates.items():

# 计算欧氏距离

distance = np.sqrt(np.sum((test\_img - template) \*\* 2))

# print(f"\t{filename} 与 {digit} 的距离为：{distance}")

if distance < min\_distance:

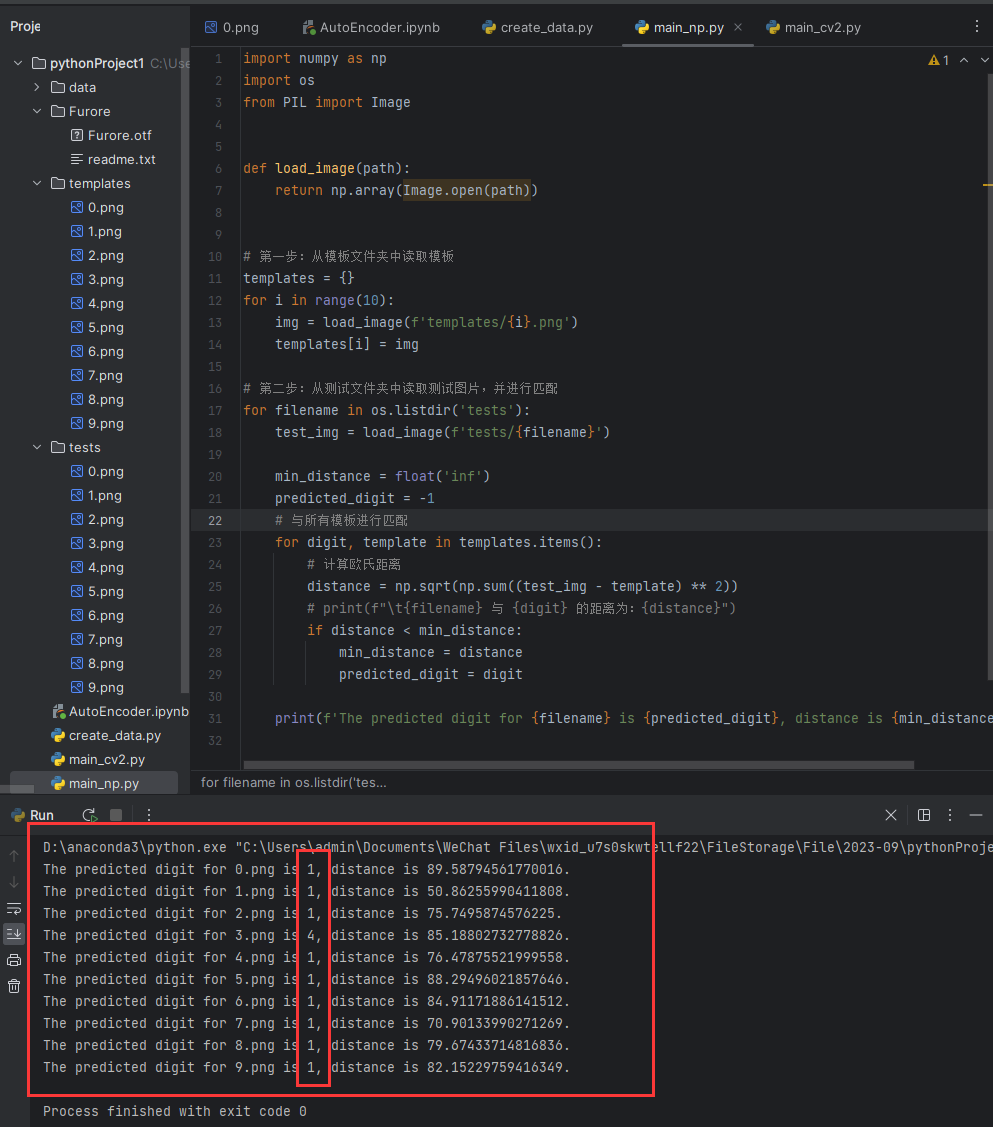
min\_distance = distance

predicted\_digit = digit

print(f'The predicted digit for {filename} is {predicted\_digit}, distance is {min\_distance}.')

```

然而，这种方法的效果并不理想，将大部分图片都分类为了数字1，准确率仅有10%。



### 3. 方案二：模板匹配

接下来我们使用了opencv的模板匹配方法。通过cv2.matchTemplate函数，我们可以计算测试图片与模板图片的相似度，然后按照相似度进行分类。

```python

import os

import cv2

# 第一步：从模板文件夹中读取模板

templates = {}

for i in range(10):

img = cv2.imread(f'templates/{i}.png', 0) # 这里假设你的模板图片是黑白的，并且文件名是数字.png

templates[i] = img

# 第二步：从测试文件夹中读取测试图片，并进行匹配

for filename in os.listdir('tests'):

test\_img = cv2.imread(f'tests/{filename}', 0)

max\_similarity = -1

predicted\_digit = -1

# 与所有模板进行匹配

for digit, template in templates.items():

# 使用模板匹配

res = cv2.matchTemplate(test\_img, template, cv2.TM\_CCOEFF\_NORMED)

min\_val, max\_val, min\_loc, max\_loc = cv2.minMaxLoc(res)

if max\_val > max\_similarity:

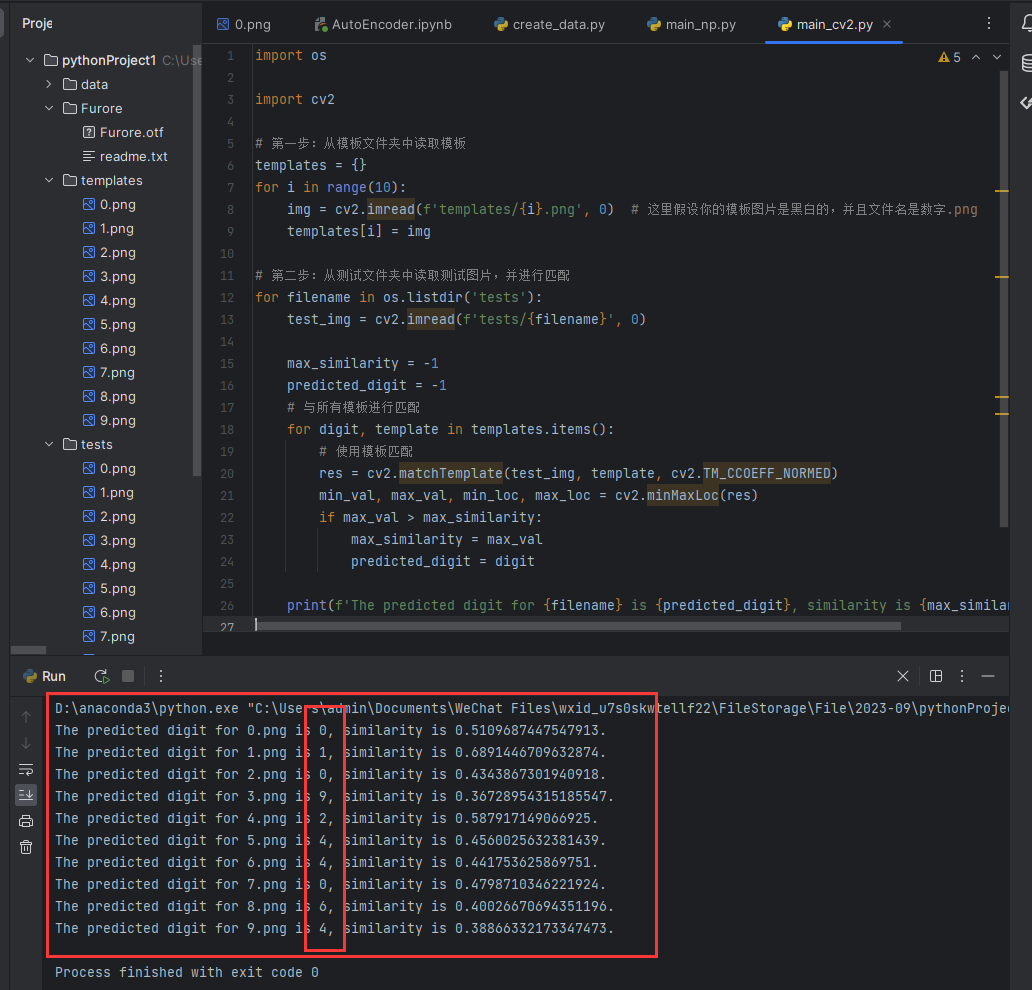
max\_similarity = max\_val

predicted\_digit = digit

print(f'The predicted digit for {filename} is {predicted\_digit}, similarity is {max\_similarity}.')

```

然而，这种方法的效果依然不佳，准确率只有20%。



由于以上两种方法均基于原始像素值进行匹配，没有考虑到图片的特征，因此效果不尽人意。为了提取图片的特征，我们尝试了第三种方法。

### 4. 方案三：自编码器特征提取

我们创建了一个自编码器模型，并在MNIST数据集上进行训练。训练好后，我们用该模型提取模板图片和测试图片的特征，然后通过欧氏距离进行匹配。

```python

#%%

import torch as th

import torch.nn as nn

import torch.nn.functional as F

#%%

class DownConvLayer(nn.Module):

def \_\_init\_\_(self, dim):

super(DownConvLayer, self).\_\_init\_\_()

self.conv = nn.Conv2d(dim, dim, 3, padding=1)

self.pool = nn.MaxPool2d(2)

def forward(self, x):

x = F.relu(self.conv(x))

x = self.pool(x)

return x

#%% md

以上代码定义了一个名为DownConvLayer的类，主要功能是实现了一个下采样卷积层，用于在卷积神经网络中进行特征提取和降维。在类的初始化方法中，定义了两个子模块。第一个子模块是使用nn.Conv2d函数定义的卷积层，输入和输出的通道数都是dim，卷积核大小为3x3，padding为1，表示在输入特征图周围填充一圈0，使得卷积操作后特征图大小不变。第二个子模块是nn.MaxPool2d，用于进行最大池化操作，将输入特征图尺寸缩小一半。在forward方法中，接收输入x，并通过卷积操作、ReLU激活函数和最大池化操作进行特征提取和降维，最后返回降维后的特征图。

#%%

# 创建一个DownConvLayer对象

down\_conv = DownConvLayer(dim=1)

# 创建一个输入张量，假设尺寸为[batch\_size, channel, height, width]

input\_tensor = th.randn(1, 1, 28, 28)

# 将输入张量传递给DownConvLayer的forward方法进行前向计算

output\_tensor = down\_conv(input\_tensor)

# 输出降维后的特征图尺寸

print(output\_tensor.shape)

#%% md

在测试用例中，我们创建了一个包含3个通道的输入张量，尺寸为[1, 3, 32, 32]。然后将输入张量传递给DownConvLayer的forward方法进行前向计算，得到输出张量output\_tensor。最后打印输出张量的形状，可以看到输出特征图的尺寸被压缩了一半，即成为[1, 3, 16, 16]。

#%%

class UpConvLayer(nn.Module):

def \_\_init\_\_(self, dim):

super(UpConvLayer, self).\_\_init\_\_()

self.conv = nn.Conv2d(dim, dim, 3, padding=1)

self.upsample = nn.Upsample(scale\_factor=2, mode="nearest")

def forward(self, x):

x = F.relu(self.conv(x))

x = self.upsample(x)

return x

#%% md

以上代码定义了一个名为UpConvLayer的类，主要功能是实现了一个上采样卷积层，用于在卷积神经网络中进行特征上采样和提取更细致的特征。在类的初始化方法中，定义了两个子模块。第一个子模块是使用nn.Conv2d函数定义的卷积层，具体参数和作用与DownConvLayer一致。第二个子模块是nn.Upsample，用于进行上采样操作，通过指定scale\_factor参数来确定上采样的倍数，mode参数设置为"nearest"表示使用最近邻插值。在forward方法中，接收输入x，并通过卷积操作、ReLU激活函数和上采样操作进行特征提取和上采样，最后返回上采样后的特征图。

#%%

# 创建一个UpConvLayer对象

up\_conv = UpConvLayer(dim=1)

# 创建一个输入张量，假设尺寸为[batch\_size, channel, height, width]

input\_tensor = th.randn(1, 1, 14, 14)

# 将输入张量传递给UpConvLayer的forward方法进行前向计算

output\_tensor = up\_conv(input\_tensor)

# 输出上采样后的特征图尺寸

print(output\_tensor.shape)

#%% md

在测试用例中，我们创建了一个包含3个通道的输入张量，尺寸为[1, 3, 16, 16]。然后将输入张量传递给UpConvLayer的forward方法进行前向计算，得到输出张量output\_tensor。最后打印输出张量的形状，可以看到输出特征图的尺寸被放大了一倍，即成为[1, 3, 32, 32]。

#%%

class Encoder(nn.Module):

def \_\_init\_\_(self, dim, layer\_num=3):

super(Encoder, self).\_\_init\_\_()

self.convs = nn.ModuleList([DownConvLayer(dim) for \_ in range(layer\_num)])

def forward(self, x):

for conv in self.convs:

x = conv(x)

return x

#%%

class Decoder(nn.Module):

def \_\_init\_\_(self, dim, layer\_num=3):

super(Decoder, self).\_\_init\_\_()

self.convs = nn.ModuleList([UpConvLayer(dim) for \_ in range(layer\_num)])

self.final\_conv = nn.Conv2d(dim, 1, 3, stride=1, padding=1)

def forward(self, x):

for conv in self.convs:

x = conv(x)

reconstruct = self.final\_conv(x)

return reconstruct

#%% md

以上代码定义了一个Encoder类和一个Decoder类，分别用于实现编码器和解码器的功能。Encoder类中包含一个nn.ModuleList对象self.convs，根据layer\_num参数的设置，通过循环创建layer\_num个DownConvLayer实例，并将它们存储在self.convs中。在Encoder类的forward方法中，对于self.convs中的每个DownConvLayer实例，将输入x传递给它们进行特征提取和降维操作，并将结果作为下一个DownConvLayer的输入。最后返回最后一个DownConvLayer的输出结果。Decoder类中也包含一个nn.ModuleList对象self.convs，用于存储多个UpConvLayer类的实例，以及一个最终的卷积层self.final\_conv。初始化方法与Encoder类类似，根据layer\_num参数的设置，通过循环创建layer\_num个UpConvLayer实例，并将它们存储在self.convs中。同时，创建一个卷积层self.final\_conv，用于将解码器的输出特征图映射为最终的重建图像。在Decoder类的forward方法中，对于self.convs中的每个UpConvLayer实例，将输入x传递给它们进行特征上采样和提取更细致的特征操作，并将结果作为下一个UpConvLayer的输入。最后将最后一个UpConvLayer的输出结果通过self.final\_conv进行卷积运算，得到最终的重建图像。

#%%

# 创建一个Encoder对象和一个Decoder对象

encoder = Encoder(dim=1, layer\_num=2)

decoder = Decoder(dim=1, layer\_num=2)

# 创建一个输入张量，假设尺寸为[batch\_size, channel, height, width]

input\_tensor = th.randn(1, 1, 28, 28)

# 将输入张量传递给Encoder的forward方法进行前向计算

encoded\_tensor = encoder(input\_tensor)

# 输出编码后图像的尺寸

print(encoded\_tensor.shape)

# 将编码后的张量传递给Decoder的forward方法进行前向计算

reconstructed\_tensor = decoder(encoded\_tensor)

# 输出重建图像的尺寸

print(reconstructed\_tensor.shape)

#%%

class AutoEncoderModel(nn.Module):

def \_\_init\_\_(self):

super(AutoEncoderModel, self).\_\_init\_\_()

self.encoder = Encoder(1, layer\_num=1)

self.decoder = Decoder(1, layer\_num=1)

def forward(self, inputs):

latent = self.encoder(inputs)

reconstruct\_img = self.decoder(latent)

return latent, reconstruct\_img

#%%

import torchvision

import torchvision.transforms as transforms

#%%

# 加载和预处理MNIST数据集

transform = transforms.Compose([

transforms.Resize((32, 32)),

transforms.ToTensor()

])

train\_dataset = torchvision.datasets.MNIST(root="./data", train=True, transform=transform, download=True)

train\_loader = th.utils.data.DataLoader(train\_dataset, batch\_size=128, shuffle=True)

# 创建自编码器模型实例、优化器和损失函数

model = AutoEncoderModel()

optimizer = th.optim.Adam(model.parameters(), lr=1e-4, weight\_decay=0.01)

criterion = nn.MSELoss()

# 训练自编码器模型

num\_epochs = 20

device = th.device("cuda" if th.cuda.is\_available() else "cpu")

model.to(device)

#%%

for epoch in range(num\_epochs):

running\_loss = 0.0

for data in train\_loader:

images, \_ = data

images = images.to(device)

optimizer.zero\_grad()

\_, reconstructed\_images = model(images)

loss = criterion(images, reconstructed\_images)

loss.backward()

optimizer.step()

running\_loss += loss.item()

epoch\_loss = running\_loss / len(train\_loader)

print(f"Epoch [{epoch+1}/{num\_epochs}], Loss: {epoch\_loss:.4f}")

print("Training finished!")

#%%

import matplotlib.pyplot as plt

#%%

image = images[1].cpu().numpy()

image = image.squeeze()

plt.imshow(image, cmap="gray")

plt.show()

#%%

image.shape

#%%

image = reconstructed\_images[1].cpu().detach().numpy()

image = image.squeeze()

plt.imshow(image, cmap="gray")

plt.show()

#%%

import numpy as np

#%%

def load\_image(path):

return np.array(Image.open(path))

#%%

model.eval()

#%%

from PIL import Image

import os

model.eval()

# 读取templates文件夹下的所有图片

template\_dir = './templates'

template\_images = os.listdir(template\_dir)

# 创建一个空的list，用于存储所有的图片的latent representation

all\_latents = []

for img\_name in template\_images:

# 打开图片并进行预处理

img\_path = os.path.join(template\_dir, img\_name)

img = Image.open(img\_path).convert('L') # 转换为灰度图，如果你的模型输入为彩色图像，这里需要进行相应改变

img = transform(img) # 使用和训练模型时相同的转换

img = img.unsqueeze(0).to(device) # 增加一个批量维度，并将数据发送到设备

# 通过模型得到latent representation

with th.no\_grad():

latent, \_ = model(img)

all\_latents.append(latent)

# 此时，all\_latents列表中存储的就是所有模板图片的latent representation

#%%

len(all\_latents)

#%%

from scipy.spatial import distance

# 读取tests文件夹下的所有图片

test\_dir = './tests'

test\_images = os.listdir(test\_dir)

# 遍历所有测试图片

for img\_name in test\_images:

# 打开图片并进行预处理

img\_path = os.path.join(test\_dir, img\_name)

img = Image.open(img\_path).convert('L') # 转换为灰度图，如果你的模型输入为彩色图像，这里需要进行相应改变

img = transform(img) # 使用和训练模型时相同的转换

img = img.unsqueeze(0).to(device) # 增加一个批量维度，并将数据发送到设备

# 通过模型得到latent representation

with th.no\_grad():

test\_latent, \_ = model(img)

# 计算测试图片的latent representation与所有模板图片的latent representation的距离

distances = [distance.euclidean(test\_latent.cpu().numpy().flatten(), template\_latent.cpu().numpy().flatten()) for template\_latent in all\_latents]

# 找到距离最小的模板图片

min\_distance\_idx = np.argmin(distances)

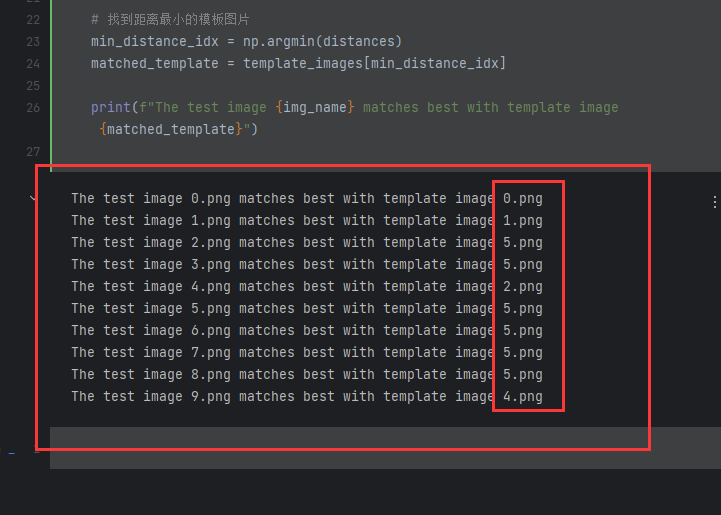
matched\_template = template\_images[min\_distance\_idx]

print(f"The test image {img\_name} matches best with template image {matched\_template}")

#%%

```

此方案的准确率达到了30%，相对来说效果较好。



由于训练数据集和测试数据集并不属于同一分布，因此此方案还有很大的提升空间。

## 三、实验结果

通过使用自编码器进行特征提取，我们得到了更好的匹配结果。我们计算了测试图片的latent representation与所有模板图片的latent representation的欧氏距离，然后找到距离最小的模板图片，认为该模板图片所对应的数字就是测试图片所表示的数字。

## 四、实验总结

本次实验我们通过三种不同的方法进行了数字识别的实验。结果表明，直接使用欧氏距离或模板匹配的方法，效果并不理想。而通过自编码器进行特征提取后，再使用欧氏距离进行匹配，效果有所提升。

这表明，在进行图片匹配时，简单地比较原始像素值并不总是有效的，特别是当图片之间存在样式差异时。而通过一些更复杂的模型，如自编码器，我们可以提取到更高级别的特征，从而得到更好的匹配结果。

尽管如此，本次实验的结果依然不完美，在未来的工作中，我们可以尝试使用更复杂的模型，或者增加训练的数据量和次数，以期得到更好的结果。

总的来说，本次实验是一次有意义的尝试，它让我们了解了数字识别的基本方法，也让我们见识了深度学习在此类问题上的强大能力。