### 实验五 HMAX模型实现

Poggio以及Serre等人于2007年提出了一个仿脑、基于特征组合的对象特征提取模型-HMAX模型，该模型的理论基础是生物学中对视皮层神经细胞进行对象识别机制的研究。HMAX模型通过Gabor滤波、求最大值操作以及交替进行模板匹配模拟了人眼视皮层中神经细胞进行对象识别处理过程。HMAX模型在模式识别领域主要作用于识别对象的特征提取，所提取的特征称为HMAX特征。HMAX模型是一个层次式的结构，分为5层：S1层、C1层、S2层、C2层、VTU层。S1层、C1层、S2层、C2层分别对应视皮层中的简单细胞和复杂细胞，而VTU层对应于识别细胞。

**实验目的：**加深对HMAX模型的理解，能够使用HMAX模型解决简单问题

**实验原理：**HMAX模型是一个层次式的结构，分为5层：S1层、C1层、S2层、C2层、VTU层。

S1层。该层事实上是用4个方向及16个尺度的Gabor滤波器组对输入图像进行滤波，得到64个响应图。在尺度上，HMAX算法分的相当细致，它将16个尺度分成8个子带，每个子带包含两个相邻尺度，以便在之后的C1层进行整合。Gabor函数滤波如下所示：

其中，是高斯函数的有效宽度，是方向，是波长。根据V1简单细胞来对参数进行调整和确定。S1响应形成8个波段，每个波段有4个方向和2个级别。每个波段有两个滤波器，一共有64个不同的S1响应，有四个方向（）。

C1层。如前所述，64个响应图依子带编号分成8组，每组包括S1层得到的2个相邻尺度及所属每个尺度上所有的4个方向响应。C1层的操作是依组依方向进行的。具体做法是先将任一个滤波器响应图划分成8\*8的格子，在每个格子中求响应的最大值。这样对每个响应图都能得到一张采样过的最大值图；然后，对组内的两个相邻尺度的最大值图的对应像素再求一次最大值，以最终得到具有不变性质的响应。值得注意的是，以上是8x8区域没有重叠的情况，在这种情况下，减采样倍数为8；实际上经常采用8x8区域相互重叠一半的情况以进一步增加不变性，在这种情况下减采样倍数为4，数据经过C1层之后，我们在每组中得到的是4个方向的不变响应。注意最大值不能跨方向选取。对于物体识别，可以在C1层生成的不变响应中抽取每类特征，并加以保存作为训练结果。其具体做法为：随机找到一个子带，在4方向不变响应图中抽取一块m\*m\*4的立方体，可以称之为patch。

S2层。在该层中，再次使用滤波器对C1层的输出进行滤波，产生新一轮的响应图。只是这次滤波器变成了之前随机抽取的那些，而非第一层硬性规定的Gabor滤波器。滤波的方式也有所不同，不再是卷积运算，而是直接以L2距离作比较，再用高斯核映射成相似度。注意到随机抽取的patch是n\*n\*4的，因此做完滤波后，对每一个patch我们能得到8个组各一张图的响应。假设patch数目是K，则共有8\*K张响应作为S2的输出。

C2层。对每一个patch遍历8个组的响应，找到最大的那个值。如此，对每张输入图片，最后得到一个K维向量。

VTU层。该层可以使用SVM及其他方法，将C2层输出的K维向量放入该层进行训练。

**实验内容：**根据HMAX模型的相关知识，使用Python语言实现一个简单的HMAX模型。简单程序示例如下：

import numpy as np

from scipy.io import loadmat

import torch

from torch import nn

def gabor\_filter(size, wavelength, orientation):

lambda\_ = size \* 2. / wavelength

sigma = lambda\_ \* 0.8

gamma = 0.3 # spatial aspect ratio: 0.23 < gamma < 0.92

theta = np.deg2rad(orientation + 90)

# Generate Gabor filter

x, y = np.mgrid[:size, :size] - (size // 2)

rotx = x \* np.cos(theta) + y \* np.sin(theta)

roty = -x \* np.sin(theta) + y \* np.cos(theta)

filt = np.exp(-(rotx\*\*2 + gamma\*\*2 \* roty\*\*2) / (2 \* sigma \*\* 2))

filt \*= np.cos(2 \* np.pi \* rotx / lambda\_)

filt[np.sqrt(x\*\*2 + y\*\*2) > (size / 2)] = 0

# Normalize the filter

filt = filt - np.mean(filt)

filt = filt / np.sqrt(np.sum(filt \*\* 2))

return filt

class S1(nn.Module):

def \_\_init\_\_(self, size, wavelength, orientations=[90, -45, 0, 45]):

super().\_\_init\_\_()

self.num\_orientations = len(orientations)

self.size = size

# Use PyTorch's Conv2d as a base object. Each "channel" will be an

# orientation.

self.gabor = nn.Conv2d(1, self.num\_orientations, size,

padding=size // 2, bias=False)

# Fill the Conv2d filter weights with Gabor kernels: one for each

# orientation

for channel, orientation in enumerate(orientations):

self.gabor.weight.data[channel, 0] = torch.Tensor(

gabor\_filter(size, wavelength, orientation))

# A convolution layer filled with ones. This is used to normalize the

# result in the forward method.

self.uniform = nn.Conv2d(1, 4, size, padding=size // 2, bias=False)

nn.init.constant\_(self.uniform.weight, 1)

# Since everything is pre-computed, no gradient is required

for p in self.parameters():

p.requires\_grad = False

def forward(self, img):

"""Apply Gabor filters, take absolute value, and normalize."""

s1\_output = torch.abs(self.gabor(img))

norm = torch.sqrt(self.uniform(img \*\* 2))

norm.data[norm == 0] = 1 # To avoid divide by zero

s1\_output /= norm

return s1\_output

class C1(nn.Module):

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

super().\_\_init\_\_()

self.size = size

self.local\_pool = nn.MaxPool2d(size, stride=size // 2,

padding=size // 2)

def forward(self, s1\_outputs):

"""Max over scales, followed by a MaxPool2d operation."""

s1\_outputs = torch.cat([out.unsqueeze(0) for out in s1\_outputs], 0)

# Pool over all scales

s1\_output, \_ = torch.max(s1\_outputs, dim=0)

# Pool over local (c1\_space x c1\_space) neighbourhood

return self.local\_pool(s1\_output)

class S2(nn.Module):

def \_\_init\_\_(self, patches, activation='gaussian', sigma=1):

super().\_\_init\_\_()

self.activation = activation

self.sigma = sigma

num\_patches, num\_orientations, size, \_ = patches.shape

# Main convolution layer

self.conv = nn.Conv2d(in\_channels=num\_orientations,

out\_channels=num\_orientations \* num\_patches,

kernel\_size=size,

padding=size // 2,

groups=num\_orientations,

bias=False)

self.conv.weight.data = torch.Tensor(

patches.transpose(1, 0, 2, 3).reshape(1600, 1, size, size))

# A convolution layer filled with ones. This is used for the distance

# computation

self.uniform = nn.Conv2d(1, 1, size, padding=size // 2, bias=False)

nn.init.constant\_(self.uniform.weight, 1)

# This is also used for the distance computation

self.patches\_sum\_sq = nn.Parameter(

torch.Tensor((patches \*\* 2).sum(axis=(1, 2, 3))))

self.num\_patches = num\_patches

self.num\_orientations = num\_orientations

self.size = size

# No gradient required for this layer

for p in self.parameters():

p.requires\_grad = False

def forward(self, c1\_outputs):

s2\_outputs = []

for c1\_output in c1\_outputs:

conv\_output = self.conv(c1\_output)

# Unstack the orientations

conv\_output\_size = conv\_output.shape[3]

conv\_output = conv\_output.view(

-1, self.num\_orientations, self.num\_patches, conv\_output\_size,

conv\_output\_size)

# Pool over orientations

conv\_output = conv\_output.sum(dim=1)

# Compute distance

c1\_sq = self.uniform(

torch.sum(c1\_output \*\* 2, dim=1, keepdim=True))

dist = c1\_sq - 2 \* conv\_output

dist += self.patches\_sum\_sq[None, :, None, None]

# Apply activation function

if self.activation == 'gaussian':

dist = torch.exp(- 1 / (2 \* self.sigma \*\* 2) \* dist)

elif self.activation == 'euclidean':

dist[dist < 0] = 0 # Negative values should never occur

torch.sqrt\_(dist)

dist = -dist

else:

raise ValueError("activation parameter should be either "

"'gaussian' or 'euclidean'.")

s2\_outputs.append(dist)

return s2\_outputs

class C2(nn.Module):

"""A layer of C2 units operating on a layer of S2 units."""

def forward(self, s2\_outputs):

"""Take the maximum value of the underlying S2 units."""

maxs = [s2.max(dim=3)[0] for s2 in s2\_outputs]

maxs = [m.max(dim=2)[0] for m in maxs]

maxs = torch.cat([m[:, None, :] for m in maxs], 1)

return maxs.max(dim=1)[0]

class HMAX(nn.Module):

def \_\_init\_\_(self, universal\_patch\_set, s2\_act='gaussian'):

super().\_\_init\_\_()

# S1 layers, consisting of units with increasing size

self.s1\_units = [

S1(size=7, wavelength=4),

S1(size=9, wavelength=3.95),

S1(size=11, wavelength=3.9),

S1(size=13, wavelength=3.85),

S1(size=15, wavelength=3.8),

S1(size=17, wavelength=3.75),

S1(size=19, wavelength=3.7),

S1(size=21, wavelength=3.65),

S1(size=23, wavelength=3.6),

S1(size=25, wavelength=3.55),

S1(size=27, wavelength=3.5),

S1(size=29, wavelength=3.45),

S1(size=31, wavelength=3.4),

S1(size=33, wavelength=3.35),

S1(size=35, wavelength=3.3),

S1(size=37, wavelength=3.25),

]

# Explicitly add the S1 units as submodules of the model

for s1 in self.s1\_units:

self.add\_module('s1\_%02d' % s1.size, s1)

# Each C1 layer pools across two S1 layers

self.c1\_units = [

C1(size=8),

C1(size=10),

C1(size=12),

C1(size=14),

C1(size=16),

C1(size=18),

C1(size=20),

C1(size=22),

]

# Explicitly add the C1 units as submodules of the model

for c1 in self.c1\_units:

self.add\_module('c1\_%02d' % c1.size, c1)

# Read the universal patch set for the S2 layer

m = loadmat(universal\_patch\_set)

patches = [patch.reshape(shape[[2, 1, 0, 3]]).transpose(3, 0, 2, 1)

for patch, shape in zip(m['patches'][0], m['patchSizes'].T)]

# One S2 layer for each patch scale, operating on all C1 layers

self.s2\_units = [S2(patches=scale\_patches, activation=s2\_act)

for scale\_patches in patches]

# Explicitly add the S2 units as submodules of the model

for i, s2 in enumerate(self.s2\_units):

self.add\_module('s2\_%d' % i, s2)

# One C2 layer operating on each scale

self.c2\_units = [C2() for s2 in self.s2\_units]

# Explicitly add the C2 units as submodules of the model

for i, c2 in enumerate(self.c2\_units):

self.add\_module('c2\_%d' % i, c2)

def run\_all\_layers(self, img):

s1\_outputs = [s1(img) for s1 in self.s1\_units]

# Each C1 layer pools across two S1 layers

c1\_outputs = []

for c1, i in zip(self.c1\_units, range(0, len(self.s1\_units), 2)):

c1\_outputs.append(c1(s1\_outputs[i:i+2]))

s2\_outputs = [s2(c1\_outputs) for s2 in self.s2\_units]

c2\_outputs = [c2(s2) for c2, s2 in zip(self.c2\_units, s2\_outputs)]

return s1\_outputs, c1\_outputs, s2\_outputs, c2\_outputs

def forward(self, img):

"""Run through everything and concatenate the output of the C2s."""

c2\_outputs = self.run\_all\_layers(img)[-1]

c2\_outputs = torch.cat(

[c2\_out[:, None, :] for c2\_out in c2\_outputs], 1)

return c2\_outputs

def get\_all\_layers(self, img):

s1\_out, c1\_out, s2\_out, c2\_out = self.run\_all\_layers(img)

return (

[s1.cpu().detach().numpy() for s1 in s1\_out],

[c1.cpu().detach().numpy() for c1 in c1\_out],

[[s2\_.cpu().detach().numpy() for s2\_ in s2] for s2 in s2\_out],

[c2.cpu().detach().numpy() for c2 in c2\_out],

)

**实验要求：**

1. 下载MNIST数据集。
2. 构建HMAX模型。
3. 使用MNIST数据集中的训练集训练网络，使用测试集测试训练好的网络。