高级计算机视觉——第五次作业

课程： 高级计算机视觉
姓名： Koorye
学号： xxxxxx
时间： 2024 年 5 月 11 日

目录

1 实验内容和目的

光学神经网络 (ONN) 是一种基于光学器件的神经网络，其基本原理是利用光学器件的并行性和高速性来加速神经网络的训练和推理过程。本实验将使用 PyTorch 实现一个简单的光学神经网络，并使用 MNIST 数据集进行训练和测试。实验目的是通过实现光学神经网络，加深对神经网络的理解，掌握 PyTorch 的基本使用方法，以及了解光学神经网络的基本原理。

本次实验的步骤有：

- `student_code.py` 中四个函数的实现，代码以及相关解释。
- `proj5.ipynb` 中训练流程以及 MNIST 分类准确度以及可视化分析。
- 思考光学神经网络通过梯度下降优化物理硬件参数的过程，理解并分析光学神经网络的推理过程，根据自己的理解，形成报告。

2 实验环境

本实验基于以下环境：

- 操作系统：Windows 11
- 编程语言：Python 3.12.1
- 编程工具：Jupyter Notebook
- Python 库：numpy 1.24.4、matplotlib 3.7.5、torch 2.2.2

3 实验原理

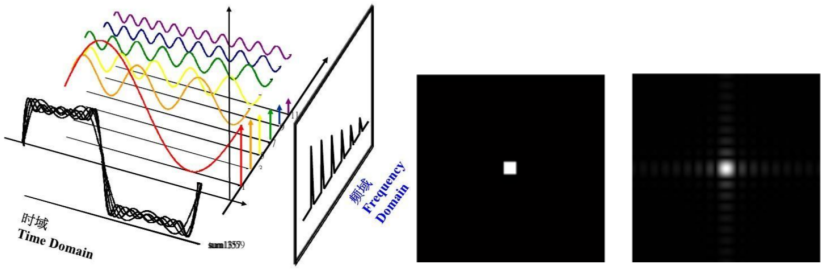
传统的人工神经网络 (ANN) 基于电子器件模拟神经系统的结构，建立神经网络各层神经元之间的联系，然而，电子器件的速度和功耗等方面的限制，使得传统的人工神经网络在训练和推理过程中存在一定的局限性。最初的神经网络基于 CPU 训练，但无法满足深度网络中大量浮点和并行运算的要求；之后，基于 GPU 的神经网络训练方法得到了广泛应用，然而电信号的能耗和物理限制仍然是一个问题。因此，寻找一种新的神经网络训练和推理方法是非常重要的。

作为替代，将光作为媒介是一个可行的方法，光速高达每秒 30 万公里，且并行性高、抗干扰性强，在信息传输和光计算方面具有巨大优势。光学神经网络 (ONN) 天然具有并行性和高速性，因此在训练和推理过程中具有很大的优势。光学神经网络的基本原理是利用光学器件的非线性特性，将神经网络的激活函数和权重参数映射到光学器件的物理参数上，通过光学器件的非线性特性来实现神经网络的训练和推理过程。训练完成之后，整个结构就能以光速进行光信号计算，而无需额外的能量输入。

本章节将介绍光学神经网络的基本原理，包括傅立叶变换、非线性光学器件、光学神经网络等内容。

3.1 傅立叶变换

时域和频域是信号处理中常用的两种表示方法，时域表示信号随时间的变化，频域表示信号的频率特性。傅立叶变换是一种将信号从时域转换到频域的数学工具，可以更好地理解信号的频率特性。而对于二维图像来说，傅立叶变换可以将图像从空域转换到频域，从而可以更好地理解图像的频率特性。



(a) 时域和频域　　(b) 空域和频域

图 1: 时域和频域、空域和频域

如图 1所示，时域信号可以通过傅立叶变换转换为频域信号，空间域图像也可以通过傅立叶变换转换为频域图像。对于一维信号来说，时域信号表示信号随时间的变化，频域信号则表示信号在不同频率上的分量；对于二维图像来说，空域图像表示图像的像素值，频域图中的每个像素表示图像在不同方向和频率上的分量。

傅立叶变换是一种信号处理中常用的数学工具，它可以将一个信号从时域转换到频域。具体来说，任意信号都可以表示为不同频率的正弦波的叠

加，傅立叶变换可以将信号分解为不同频率的正弦波，从而可以更好地理解信号的频率特性。傅立叶变换可以表示为公式 1：

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t} dt \tag{1}$$

其中，$F(\omega)$ 是信号 $f(t)$ 的傅立叶变换，ω 是频率，j 是虚数单位。傅立叶变换可以将信号从时域转换到频域，从而可以更好地理解信号的频率特性。

对于图像来说，傅立叶变换同样可以将图像从空间域转换到频域。这是因为任意图像都可以看作是一个二维的信号，傅立叶变换同样可以将图像分解为各个不同方向和频率的正弦波，从而可以更好地理解图像的频率特性。二维图像的傅立叶变换可以表示为公式 2：

$$F(u, v) = \iint f(x, y) e^{-j2\pi(ux+vy)} dx dy \tag{2}$$

其中，$F(u, v)$ 是图像 $f(x, y)$ 的二维傅立叶变换，u 和 v 是频率，j 是虚数单位。二维傅立叶变换可以将图像从空域转换到频域，从而可以更好地理解图像的频率特性。

3.2 非线性光学效应

非线性光学效应是指光信号在非线性介质中传播时，光信号的强度、相位等参数会发生非线性变化。当光通过晶体进行传播时，会引起晶体的电极化。对于某些材料来说，在光强不太大时，晶体的电极化强度与光频电场之间呈线性关系，这种非线性关系可以被忽略；但是，当光强很大时，例如激光通过晶体传播时，电极化强度与光频电场之间的非线性关系变得显著。

1961 年，美国科学家 Franken 首次发现了晶体的非线性光学效应，他将红宝石产生的激光入射到石英晶体上，发现射出的光束除了红宝石的 693.4nm 波长外，还有 347.2nm 的波长。这种现象被称为二次谐波产生，是晶体的非线性光学效应的一个典型例子。之后，人们发现了更多的非线性光学材料，如 BBO、LBO 等，这些材料可以实现更多种类的非线性光学效应。

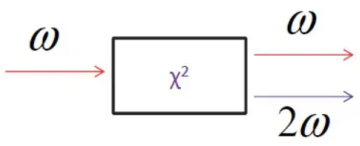


图 2: 二次谐波产生

极化现象是非线性光学效应的基础，当光激发材料时，材料中的电子会发生位移，从而产生电偶极矩。材料在光激发时会发生极化，极化强度与光频电场之间的关系可以表示为公式 3：

$$P(t) = \varepsilon_0 \chi^{(1)} E(t) + \varepsilon_0 \chi^{(2)} E^2(t) + \varepsilon_0 \chi^{(3)} E^3(t) + \cdots = P^L + P^{NL} \tag{3}$$

其中，$P(t)$ 是极化强度，ε_0 是真空介电常数，$\chi^{(1)}$、$\chi^{(2)}$、$\chi^{(3)}$ 等是非线性极化系数，$E(t)$ 是光频电场，P^L 是线性极化，P^{NL} 是非线性极化。当光强不太大时，非线性极化可以忽略，此时极化强度与光频电场之间呈线性关系；但是，当光强很大时，非线性极化变得显著，此时极化强度与光频电场之间呈非线性关系。

3.3 光学神经网络

光学神经网络 (ONN) 是一种基于光学器件的神经网络，利用光学器件的非线性特性来实现神经网络的训练和推理过程。光学神经网络的基本原理是将神经网络的激活函数和权重参数映射到光学器件的物理参数上，通过光学器件的非线性特性来实现神经网络的训练和推理过程。

光学神经网络可以在空间域中进行计算，即将输入信号通过光学器件进行传播，然后通过光学器件的非线性特性来实现神经网络的训练和推理过程。传统的电子神经网络中的每一层输出通过电子器件计算：

$$X^{l+1} = F^l(W^l \cdot X^l + B^l) \tag{4}$$

其中 W^l, B^l 是可学习参数。而全光深度衍射神经网络 (D2NN) 是一种基于光学器件的神经网络，该网络由一个输入层、若干个中间衍射层和一个输出层组成，网络的输入是相干光，并且在每个衍射层都会发生衍射产生二次波。具体来说，通过控制光相位和幅值，可以改变衍射层的投射和反射系数，从而控制衍射层的权重参数进行计算：

$$Y^{l+1} = F^l(W^l \cdot X^l + B^l), Y^l = X^l e^{j\psi^l} \tag{5}$$

其中 ψ^l 是相位参数。全光深度衍射神经网络的结构如图 3所示，D2NN 通过对相干光进行衍射，控制光的相位和幅值，最后通过将光信号划分为多个区域，根据最大值光信号的位置来判断输出结果。对于模型训练，可以通过计算机模拟光传播的过程，然后通过梯度下降等方法来更新光学器件的参数。

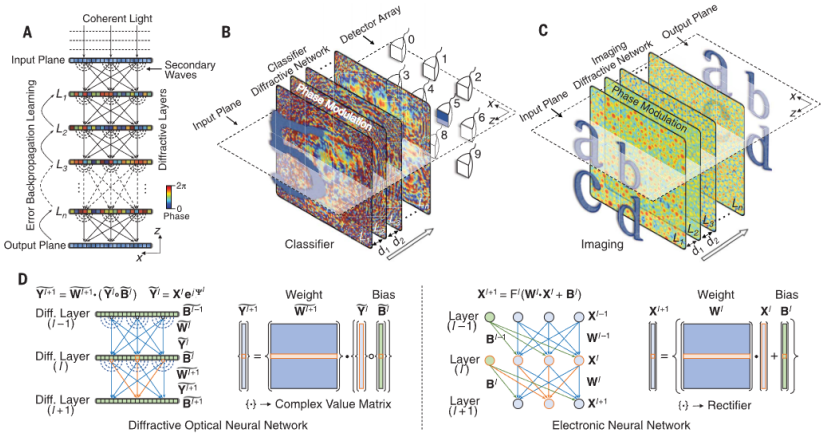


图 3: 全光深度衍射神经网络

此外，光学神经网络还可以在频域中进行计算，即将输入信号通过傅立叶变换转换到频域，然后通过光学器件的非线性特性来实现神经网络的训练和推理过程。傅立叶衍射网络 (F-D2NN) 同样由一个输入层、若干个中间衍射层和一个输出层组成，不同的是，F-D2NN 的输入是经过傅立叶变换的信号，中间衍射层加入了非线性器件，而输出层则是通过傅立叶逆变换将信号转换回空间域。

傅立叶变换域逆变换是通过 2f 系统实现的，2f 系统是一种光学系统，可以将输入信号通过透镜成像到傅立叶平面，然后通过另一个透镜将信号成像到输出平面。输入光信号首先通过透镜成像到傅立叶平面，然后通过 D2NN 网络进行计算，之后通过非线性单元，最后通过透镜将信号成像到输出平面。这一过程可以表示为：

$$\hat{U}_0 = FU_0, \hat{U}_1 = \hat{M}\hat{U}_0, \hat{U}_2 = \phi(\hat{U}_1), O = |F\hat{U}_2|^2 \tag{6}$$

其中，U_0 是输入信号，F 是傅立叶变换，$\hat{M}$ 是 D2NN 网络，ϕ 是非线性单元，O 是输出信号。F-D2NN 的结构如图 4所示，经过变换，光信号转换为输出信号，然后可以用于图像分类、显著目标检测等任务。

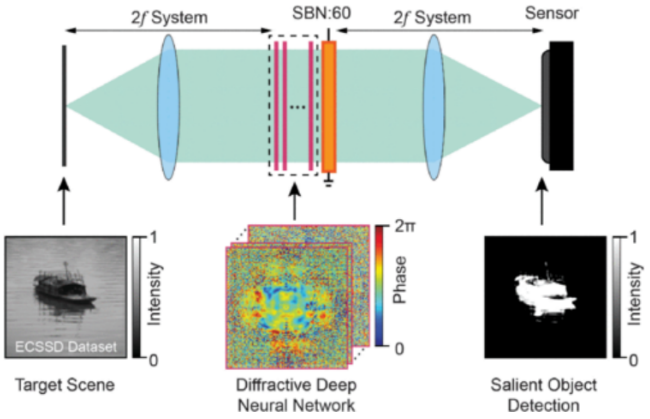


图 4: 傅立叶衍射网络

4 实验步骤

本章节将展示实验步骤，包括生成复数型张量、傅立叶移位操作、傅立叶变换、傅立叶逆变换等。

4.1 `commplex_exp_torch` 函数的实现

Pytorch 中拥有不同的数据类型，包括整型、浮点型、复数型等。在 Pytorch 中，复数型数据类型为 `torch.complex64` 和 `torch.complex128`。为了方便地生成复数型张量，本章节实现了 `commplex_exp_torch` 函数，用于生成复数型张量。

```
1 def commplex_exp_torch(phase, dtype=torch.complex128):
2     return torch. complex(torch.cos(phase), torch.sin(phase))
```

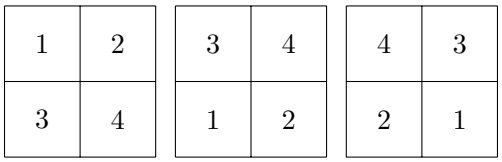
如上述代码所示，`commplex_exp_torch` 函数接受一个 `phase` 参数，用于指定复数的相位。函数返回一个 `torch.complex` 类型的张量，其中实部为 cos(phase)，虚部为 sin(phase)。

4.2 fftshift2d_tf 函数的实现

傅立叶移位操作是指将频谱的零频率分量移动到频谱的中心位置。在 Pytorch 中，torch.fft.fft2 函数返回的频谱是未移位的，即零频率分量位于频谱的左上角。为了方便地进行频谱的可视化，本章节实现了 fftshift2d_tf 函数，用于对频谱进行移位操作。

```
def ifftshift2d_tf(a_tensor):
    # (B, H, W, C)
    import math
    _, H, W, _ = a_tensor.size()
    H_split, W_split = math.ceil(H / 2), math.ceil(W / 2)
    a_tensor_up, a_tensor_down = a_tensor.split([H_split, H - H_split], dim
        =1)
    a_tensor = torch.cat([a_tensor_down, a_tensor_up], dim=1)
    a_tensor_left, a_tensor_right = a_tensor.split([W_split, W - W_split],
        dim=2)
    a_tensor = torch.cat([a_tensor_right, a_tensor_left], dim=2)
    return a_tensor
```

一维傅立叶移位操作是指将频谱的前一半数据移动到频谱的后一半位置。而二维如上述代码所示，ifftshift2d_tf 函数接受一个 a_tensor 参数，用于指定需要移位的频谱。函数返回一个移位后的频谱。具体来说，函数首先将频谱沿着垂直方向分为上下两部分，然后将这两部分交换位置；接着将频谱沿着水平方向分为左右两部分，然后将这两部分交换位置。



(a) 原始频谱 (b) 上下翻转 (c) 左右翻转

图 5: 二维傅立叶移位操作

4.3 transp_fft2d 函数的实现

傅立叶变换是指将空间域信号转换为频谱。在 Pytorch 中，torch.fft.fft2 函数用于计算二维傅立叶变换。为了方便地进行傅立叶变换操作，本章节实

现了 transp_fft2d 函数。

```
def transp_fft2d(a_tensor, dtype=torch.complex64):
    # (B, H, W, C)
    return torch.fft.fft2(a_tensor, dim=(1, 2))
```

如上述代码所示，transp_fft2d 函数接受一个 a_tensor 参数，用于指定需要进行傅立叶变换的信号。该函数对信号的高度、宽度维度进行傅立叶变换操作，返回一个复数类型的张量。

4.4 transp_ifft2d 函数的实现

傅立叶逆变换是指将频谱转换为空间域信号。在 Pytorch 中，torch.fft.ifft2 函数用于计算二维傅立叶逆变换。为了方便地进行傅立叶逆变换操作，本章节实现了 transp_ifft2d 函数。

```
def transp_ifft2d(a_tensor, dtype=torch.complex64):
    # (B, H, W, C)
    return torch.fft.ifft2(a_tensor, dim=(1, 2))
```

如上述代码所示，transp_ifft2d 函数接受一个 a_tensor 参数，用于指定需要进行傅立叶逆变换的频谱。该函数对频谱的高度、宽度维度进行傅立叶逆变换操作，返回一个复数类型的张量。

4.5 运行脚本

在编写完函数之后，可以在 proj5.ipynb 中进行测试。首先在目录下运行如下命令启动 Jupyter Notebook：

```
jupyter notebook
```

之后在浏览器中打开 http://localhost:8888，点击 proj5.ipynb 文件，点击 Kernel 菜单下的 Restart & Run All，即可运行整个脚本，如图 6所示。该脚本执行了模型训练和测试过程，与电子神经网络类似，模型通过 SGD 优化，损失函数为交叉熵损失函数，并采用 ReduceLROnPlateau 作为学习率调整策略，具体来说，当测试准确率连续 2 个 epoch 不再上升时，学习率将减小为原先的 0.5 倍。

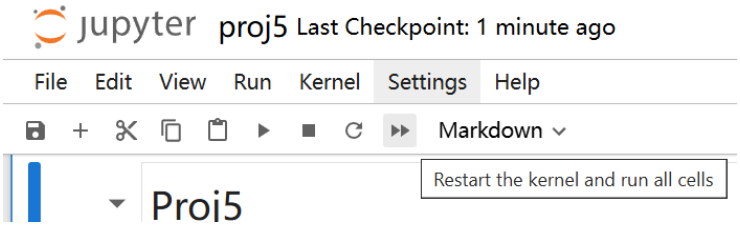


图 6: Jupyter Notebook 界面

5 实验结果与分析

本章节将展示实验结果，并对实验结果进行分析。

5.1 实验结果

编写完成所有函数之后，在 `proj5.ipynb` 中对光学神经网络模型在 MNIST 数据集上进行训练和测试。其中 batch size 为 64，学习率为 0.00001，训练轮数为 30。图 7展示了训练过程中的损失和准确率变化。

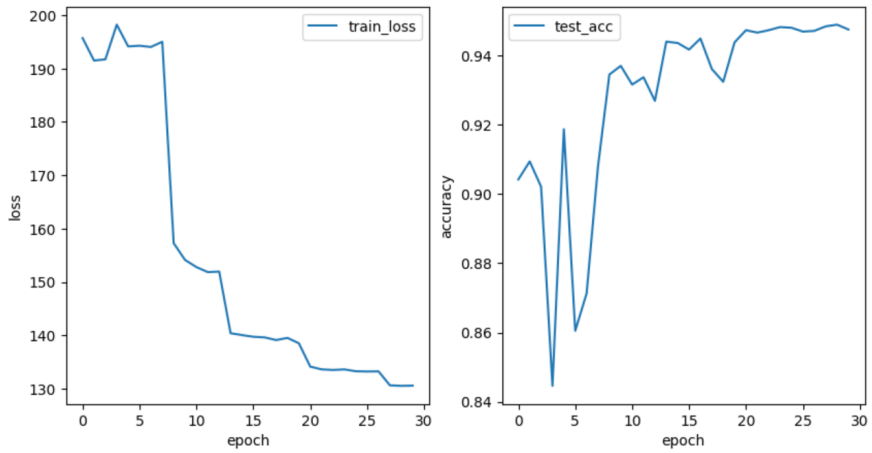


图 7: 训练过程中的损失和准确率变化

如图所示，模型在训练过程中损失逐渐下降，准确率逐渐上升。在训练结束时，模型的准确率达到了 94.9%，完成了对光学神经网络模型的训练过程。

5.2 实验分析

光学神经网络是一种基于光学器件的神经网络模型，可以实现对光学信号的处理。为了训练和测试光学神经网络模型，通常在计算机上进行模拟。本次实验搭建了一个简单的光学神经网络模型，将图像转换到频域进行处理，然后再转换回空间域。本实验中的光学神经网络的代码如下所示：

```
class onn(nn.Module):
    def __init__(self):
         super(onn, self).__init__()
        self.height_map_var = torch.randn([otf_size, otf_size, 1, 1])
        self.height_map_var = self.height_map_var.div(1000)
        self.height_map_var = nn.Parameter(self.height_map_var)
        self.refractive_index = 1.5
        self.delta_N = self.refractive_index - 1.000277
        self.wave_lengths = 550e-9
        self.wave_nos = 2. * np.pi / self.wave_lengths

    def forward(self, x):
        height_map = torch.square(self.height_map_var)
        phi = self.wave_nos * self.delta_N * height_map
        phase_shifts = commplex_exp_torch(phi)
        atf = phase_shifts

        x = torch.reshape(x, [-1, 32, 32, 1])
        paddings = (0,0, padamt,padamt, padamt,padamt, 0,0)
        x = F.pad(x, paddings, "constant", 0)
        input_img = x
        img_shape = input_img.shape
        target_side_length = 2 * img_shape[1]
        height_pad = (target_side_length - img_shape[1]) / 2
        width_pad = (target_side_length - img_shape[1]) / 2
        pad_top, pad_bottom =  int(np.ceil(height_pad)),
            int(np.floor(height_pad))
        pad_left, pad_right =  int(np.ceil(width_pad)),
            int(np.floor(width_pad))
        img1 = F.pad(input_img, (0, 0, pad_top, pad_bottom, pad_left,
            pad_right, 0, 0), "constant", 0)
        img_shape = img1.shape

```

```
31          output_img1 = transp_fft2d(img1)
32          output_img1 = ifftshift2d_tf(output_img1)
33
34          otf1 = psf2otf(atf, output_size=img_shape[1:3])
35          otf1 = otf1.transpose(0,1)
36          otf1 = otf1.transpose(0,2)
37          otf1 = otf1.to(torch.complex64)
38
39          img_fft1 = output_img1.to(torch.complex64)
40          result1 = transp_ifft2d(img_fft1 * otf1)
41          result1 = torch. abs(result1).to(torch.float32)
42          output_img1 = result1[:, pad_top:-pad_bottom, pad_left:-pad_right,
                :]
43          return output_img1
```

上述代码中，onn 类实现了光学神经网络模型，定义了可学习的高度图参数，以及折射率、波长等参数。在 forward 函数中，首先根据高度图、折射率、波长等参数计算出相位信息，然后根据相位信息计算出光调制函数。输入图像经过傅立叶变换后转换后频域图像，然后与光调制函数进行点乘操作，最后通过逆傅立叶变换得到输出图像。该网络本质上是一个卷积层，通过傅立叶变换和逆傅立叶变换，将空间域上的卷积操作转化为频域上的相乘操作，提供了一个光学容易实现的卷积方法。

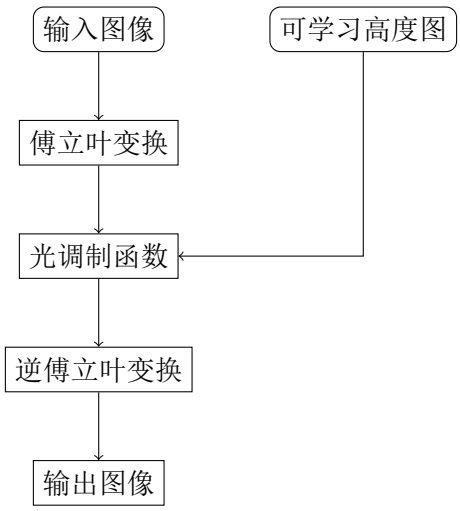


图 8: 光学神经网络模型

上述网络可以利用光学器件实现，首先傅立叶和逆傅立叶变换可以通过 f2 透镜实现，而高度图和频域图的点乘则可以通过衍射光学器件 (DOE) 实现。具体来说，DOE 器件可以通过激光照射光阑，通过光的干涉和衍射效应，实现对光场的调制。在计算机上完成模拟后，可以通过光学器件实现光学神经网络模型的部署，实现对图像的分类、检测等任务。

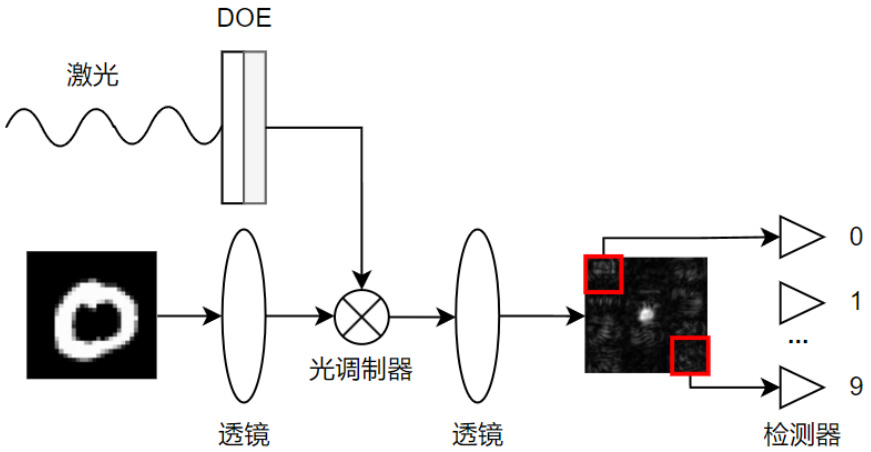


图 9: 光学神经网络模型

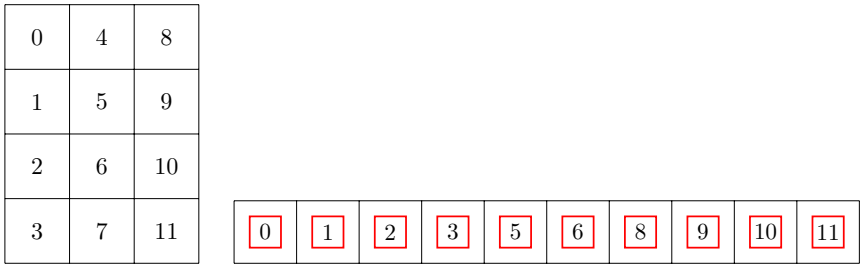
如图 9所示，该光学神经网络的硬件部分包括透镜、DOE 器件、光调制器、检测器等，可以实现对图像的处理。其中透镜负责傅立叶变换和逆傅立叶变换，DOE 器件负责点光源转换，光调制器负责频谱图与高度图的点乘操作。最后，检测器负责接收处理后的图像，对每个区域的光强进行检测，实现对图像的分类任务。

在完成特征提取后，需要对特征图进行分类操作。传统的神经网络模型通常使用全连接层进行分类，而光学神经网络模型则可以将特征图划分为多个区域，计算每个区域的平均值作为概率分布。本实验中通过计算机模拟了这一过程，代码如下所示：

```python
def center(a_tensor):
    squeeze_atensor = torch.squeeze(a_tensor)
    begin = end = 15
    center_tensor = squeeze_atensor[:, begin:-end, begin:-end]
    return torch.unsqueeze(center_tensor, -1)

def img_split(img):
    splitted_1d = torch.stack(torch.chunk(img, 4, dim=1), 0)
    splitted = torch.concat(torch.chunk(splitted_1d, 3, dim=3), 0)
    result = torch.stack(
        (center(splitted[0]), center(splitted[1]), center(splitted[2]),
         center(splitted[3]), center(splitted[5]), center(splitted[6]),
         center(splitted[8]), center(splitted[9]), center(splitted[10]),
         center(splitted[11])), 0)
    result = result.mean(dim=(2,3,4))
    result = torch.transpose(result, 0, 1)
    return result
```

上述 `img_split` 函数实现了特征图的划分和分类操作，将图像在高度维度上拆分为 4 个区域，对于每个区域再在宽度维度上拆分为 3 个区域，总共得到 12 个区域，再选取除中间两个区域以外的 10 个区域。然后，通过 `center` 函数计算每个区域的中心部分，最后计算每个区域的平均值，得到一个长度为 10 的向量，作为分类的概率分布。

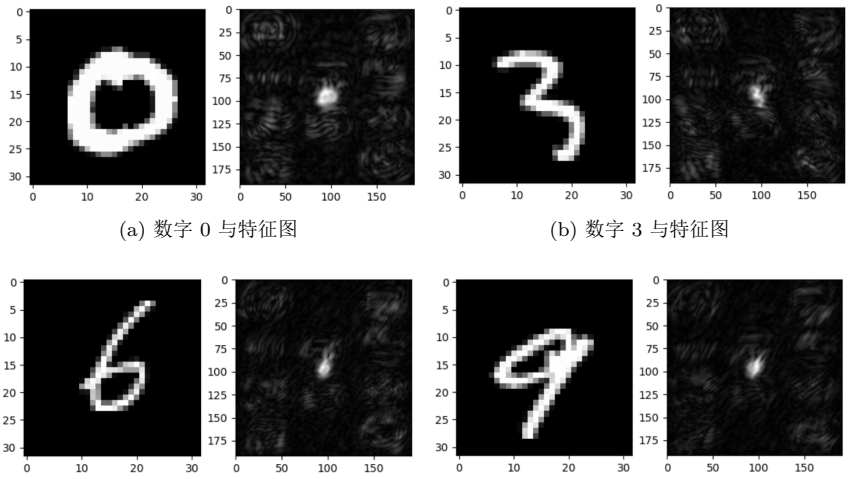


(a) 特征图划分 (b) 选择区域

图 10: 特征图划分和分类

如图 10所示，首先将特征图划分为 3x4 的网格，然后选择除中间两个区域以外的 10 个区域，选择每个区域中心标红部分计算平均值，得到一个长度为 10 的向量，作为分类的概率分布。

图 11展示了特征图的可视化结果。可以看出，不同数字在不同网格区域的特征具有不同程度的激活，例如数字 0 在左上角区域有较强的激活，而数字 9 在右下角区域有较强的激活。这说明光学神经网络模型可以有效地提取图像的特征，实现图像分类任务。



(a) 数字 0 与特征图 (b) 数字 3 与特征图

(c) 数字 6 与特征图 (d) 数字 9 与特征图

图 11: 特征图可视化

通过上述步骤，完成了光学神经网络的特征提取器和分类器的搭建，之后通过计算机模拟结构，对光学神经网络模型进行训练和测试。在 MNIST 数据集上，模型的准确率达到了 94.9%，表明光学神经网络模型可以进行有效的图像分类任务。

6 实验结论

本次实验搭建了一个简单的光学神经网络模型，将图像转换到频域进行处理，然后再转换回空间域，该网络本质上是学习了一个快速傅立叶卷积操作。通过对 MNIST 数据集的训练，模型在训练结束时的准确率达到了 94.9%。光学神经网络模型的训练过程中，损失逐渐下降，准确率逐渐上升，说明模型在训练过程中逐渐收敛。光学神经网络模型的训练结果表明，光学神经网络模型可以实现对图像的处理，具有一定的应用前景。通过本次实验，我对光学神经网络模型的原理和实现有了更深入的了解，对深度学习和

光学神经网络的研究有了更多的兴趣。

对于本次实验，我有以下建议：

- 本次实验中，我使用了 PyTorch 框架搭建了光学神经网络模型，但是目前关于光学神经网络的资料较少，希望能够有更多的资料和教程，帮助更多的研究者了解和使用光学神经网络。
- 本次实验中，我使用了 MNIST 数据集进行训练，测试图像分类能力。希望能在更多任务上进行测试，如显著目标检测、图像重建等任务，验证光学神经网络模型的性能。
- 本次实验中，我搭建的光学神经网络模型较为简单，希望能够进一步优化模型结构，提高模型的性能，使其在更多任务上取得更好的效果。

A 完整代码

```python
from torch.utils.data.dataset import Dataset
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from matplotlib import pyplot as plt
import numpy as np
import torch
import torch.nn.functional as F
from tqdm.notebook import tqdm


def center(a_tensor):
    squeeze_atensor = torch.squeeze(a_tensor)
    begin = end = 15
    center_tensor = squeeze_atensor[:, begin:-end, begin:-end]
    return torch.unsqueeze(center_tensor, -1)


def commplex_exp_torch(phase, dtype=torch.complex128):
    """
    通过commplex_exp_torch函数将phase转换为复数，这个复数的实部是cos(phase)，虚部是sin(phase)

    Useful functions:
    -   type(torch.float64)
    -   torch.complex()
    -   torch.cos()
    -   torch.sin()
    """
    return torch. complex(torch.cos(phase), torch.sin(phase))


def ifftshift2d_tf(a_tensor):
    """
    通过ifftshift2d_tf实现tensor的第二和第三维度的逆傅里叶移位(ifftshift)操
```

```
        作
37
38      Useful functions:
39      -   torch.index_select()
40      """
41      # (B, H, W, C)
42      import math
43      _, H, W, _ = a_tensor.size()
44      H_split, W_split = math.ceil(H / 2), math.ceil(W / 2)
45      a_tensor_up, a_tensor_down = a_tensor.split([H_split, H - H_split], dim
            =1)
46      a_tensor = torch.cat([a_tensor_down, a_tensor_up], dim=1)
47      a_tensor_left, a_tensor_right = a_tensor.split([W_split, W - W_split],
            dim=2)
48      a_tensor = torch.cat([a_tensor_right, a_tensor_left], dim=2)
49      return a_tensor
50
51
52  def transp_ifft2d(a_tensor, dtype=torch.complex64):
53      """
54      通过transp_ifft2d将tensor的第二、第三维度进行逆傅里叶变换
55
56      Useful functions:
57      -   torch.fft.ifft2()
58      """
59      # (B, H, W, C)
60      return torch.fft.ifft2(a_tensor, dim=(1, 2))
61
62
63  def transp_fft2d(a_tensor, dtype=torch.complex64):
64      """
65      通过transp_fft2d将tensor的第二、第三维度进行傅里叶变换
66
67      Useful functions:
68      -   torch.fft.fft2()
69      """
70      # (B, H, W, C)
71      return torch.fft.fft2(a_tensor, dim=(1, 2))
72
```

```
73
74  def psf2otf(input_filter, output_size):
75      fh, fw, _, _ = input_filter.size()
76
77      if output_size[0] != fh:
78          pad = (output_size[0] - fh) / 2
79
80          if (output_size[0] - fh) % 2 != 0:
81              pad_top = pad_left =  int(np.ceil(pad))
82              pad_bottom = pad_right =  int(np.floor(pad))
83          else:
84              pad_top = pad_left =  int(pad) + 1
85              pad_bottom = pad_right =  int(pad) - 1
86
87          padded = F.pad(torch.permute(input_filter, (2, 3, 0, 1)),  [pad_top,
                 pad_bottom, pad_left, pad_right, 0, 0, 0, 0],  mode="constant"
                )
88
89          padded = torch.permute(padded, (2, 3, 0, 1))
90      else:
91          padded = input_filter
92      return padded
93
94
95
96
97
98  class MyDataset(Dataset):
99      def __init__(self, train=True, transform=None):
100         train_data_path = '../data/mnist10-train-data.npy'
101         train_label_path = '../data/mnist10-train-label.npy'
102
103         test_data_path = '../data/mnist10-test-data.npy'
104         test_label_path = '../data/mnist10-test-label.npy'
105
106         self.train = train
107         self.transform = transform
108
109         self.train_data = np.load(train_data_path)
```

```
110         self.train_label = np.load(train_label_path)
111 
112         self.test_data = np.load(test_data_path)
113         self.test_label = np.load(test_label_path)
114 
115     def __getitem__(self, index):
116         if self.train:
117             self.label = self.train_label[index]
118             self.data = self.train_data[index]
119 
120         else:
121             self.label = self.test_label[index]
122             self.data = self.test_data[index]
123 
124         return self.data, self.label
125 
126     def __len__(self):
127         if self.train:
128             return self.train_data.shape[0]
129         else:
130             return self.test_data.shape[0]
131 
132 
133 
134 otf_size = 142
135 padamt = 80
136 
137 # 图像划分为12块，去掉中间的两块后剩下的10块对应10个数字，再将15块上下左右裁
        去10个像素，剩下大小为18*34
138 def img_split(img):
139     splitted_1d = torch.stack(torch.chunk(img, 4, dim=1), 0)
140     splitted = torch.concat(torch.chunk(splitted_1d, 3, dim=3), 0)
141 
142     result = torch.stack((center(splitted[0]),
143                           center(splitted[1]),
144                           center(splitted[2]),
145                           center(splitted[3]),
146                           center(splitted[5]),
147                           center(splitted[6]),
```

```
148                         center(splitted[8]),
149                         center(splitted[9]),
150                         center(splitted[10]),
151                         center(splitted[11])), 0)
152 
153     # 均值池化
154     result = result.mean(dim=(2,3,4))
155     result = torch.transpose(result, 0, 1)
156     # print(result.shape)
157 
158     return result
159 
160 class onn(nn.Module):
161     def __init__(self):
162          super(onn, self).__init__()
163 
164         # height map
165         self.height_map_var = torch.randn([otf_size, otf_size, 1, 1])
166         self.height_map_var = self.height_map_var.div(1000)
167         self.height_map_var = nn.Parameter(self.height_map_var)
168 
169         # parameters
170         self.refractive_index = 1.5
171         self.delta_N = self.refractive_index - 1.000277
172 
173         self.wave_lengths = 550e-9
174         self.wave_nos = 2. * np.pi / self.wave_lengths
175 
176     def forward(self, x):
177         ######################################################
178         # height_map是一个随机的高度图，大小为[142, 142, 1, 1]
179         # 用于调控光场相位，是DOE的制造参数，通过梯度下降优化
180         height_map = torch.square(self.height_map_var)
181         phi = self.wave_nos * self.delta_N * height_map
182         phase_shifts = commplex_exp_torch(phi)
183         atf = phase_shifts
184 
185 
186         ######################################################
```

```
187         # [bs, w, h, c]
188         x = torch.reshape(x, [-1, 32, 32, 1])
189         paddings = (0,0, padamt,padamt, padamt,padamt, 0,0)
190         x = F.pad(x, paddings, "constant", 0)
191
192         input_img = x
193         img_shape = input_img.shape
194
195
196         ######################################################
197         target_side_length = 2 * img_shape[1]
198
199         height_pad = (target_side_length - img_shape[1]) / 2
200         width_pad = (target_side_length - img_shape[1]) / 2
201
202         pad_top, pad_bottom =  int(np.ceil(height_pad)),
                int(np.floor(height_pad))
203         pad_left, pad_right =  int(np.ceil(width_pad)),
                int(np.floor(width_pad))
204
205         img1 = F.pad(input_img, (0, 0, pad_top, pad_bottom, pad_left,
                pad_right, 0, 0), "constant", 0)
206         img_shape = img1.shape
207
208
209
210         ######################################################
211         output_img1 = transp_fft2d(img1)
212         output_img1 = ifftshift2d_tf(output_img1)
213
214         otf1 = psf2otf(atf, output_size=img_shape[1:3])
215         otf1 = otf1.transpose(0,1)
216         otf1 = otf1.transpose(0,2)
217         otf1 = otf1.to(torch.complex64)
218
219         img_fft1 = output_img1.to(torch.complex64)
220         result1 = transp_ifft2d(img_fft1 * otf1)
221         result1 = torch. abs(result1).to(torch.float32)
222
223
```

```
224         output_img1 = result1[:, pad_top:-pad_bottom, pad_left:-pad_right,
                :]
225
226         ######################################################
227         # return img_split(output_img1)
228         return output_img1
```