高级计算机视觉——第四次作业

课程： 高级计算机视觉
姓名： Koorye
学号： xxxxxx
时间： 2024 年 4 月 29 日

目录

1 实验内容和目的

深度学习是机器学习的一个分支，通过神经网络模型实现对数据的学习和预测，具体来说，深度学习通过设计网络结构和优化算法，实现对数据的特征提取和模型训练，从而实现对数据的预测和分类。本实验通过 PyTorch 框架实现一个简单的神经网络模型，通过训练数据集，实现对数据的分类任务。

具体来说，本实验包含以下内容：

- 通过 PyTorch 实现数据集的加载和预处理；
- 使用 PyTorch 搭建一个简单的神经网络模型；
- 编写预测和损失计算函数；
- 使用数据增强和 Dropout 技术，提高模型的泛化能力；
- 测试 AlexNet 的性能。

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

深度学习是机器学习的一个分支，它的目标是模拟人脑的神经元从而识别模式。深度学习的核心是神经网络，神经网络通过多层神经元进行连接，每一层神经元都会对输入数据进行处理，最终输出结果。深度学习的训练过程是通过大量的数据进行训练，通过不断调整神经元之间的连接权重，使得神经网络的输出结果与实际结果尽可能接近。本章节将介绍神经网络的结构、优化方法和数据增强技术。

3.1 神经网络的结构

神经网络中的一些常见模块包括全连接层、卷积层、池化层、激活函数等。其中，全连接层是最简单的神经网络模块，它将输入数据的每一个特征都与神经元进行连接，通过调整权重来实现特征的提取。全连接是一种线性变换操作，其中需要学习的参数有权重和偏置。具体来说，全连接操作可以表示为公式 1：

$$y = Wx + b \tag{1}$$

其中，x 是输入数据，W 是权重，b 是偏置，y 是输出数据。全连接层结构简单，但是参数量大，容易过拟合。

卷积层是一种常用的神经网络模块，它通过卷积操作实现对输入数据的特征提取。卷积层的核心是卷积操作，后者通过滑动窗口的方式对输入数据进行处理，从而提取特征。卷积操作可以表示为公式 2：

$$y_i = \sum_{i=1}^{n} x_i \cdot W_i + b \tag{2}$$

其中，x_i 是输入数据的第 i 个特征，W_i 是卷积核的第 i 个权重，b 是偏置，y 是输出数据。卷积操作通过共享卷积核权重和局部连接的方式，减少了参数量，提高了模型的泛化能力。同时，卷积操作模仿了人类视觉系统的工作原理，提取了输入数据的空间特征。

池化层是一种常用的神经网络模块，它通过池化操作实现对输入数据的降维处理。池化层的核心是池化操作，后者通过滑动窗口的方式对输入数据进行处理，从而降低数据的维度。池化操作可以表示为公式 3：

$$y_i j = \phi_{(p,q)\in R_{ij}} x_{pq} \tag{3}$$

其中，x_{pq} 是输入数据的第 (p, q) 个特征，R_{ij} 是池化范围，ϕ 是池化操作，可以使用最大池化 max、平均池化 $\frac{1}{|R_{ij}|}\sum$ 等。池化操作通过减少数据的维度，提高了模型的计算效率，同时保留了输入数据的主要特征。

激活函数是一种常用的神经网络模块，它通过非线性变换实现对输入数据的处理，使得网络拥有更强的表达能力。激活函数的常见类型包括 ReLU、Sigmoid、Tanh 等。这些激活函数如图 1所示。其中，ReLU 是一种简单的激活函数，它将所有负数输入映射为 0，保留正数输入。Sigmoid 是一种常用的激活函数，它将输入映射到 $(0, 1)$ 之间，可以用于二分类问题。Tanh 是

一种常用的激活函数，它将输入映射到 $(-1, 1)$ 之间。此外，还有其他激活函数，如 Softmax、Leaky ReLU、ELU 等，这些激活函数是为了用于多分类，以及解决 ReLU 的失活问题而提出的。

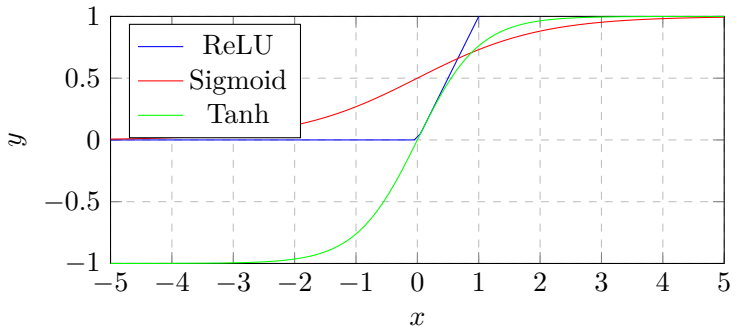


图 1: 激活函数图像

3.2 神经网络的优化方法

神经网络的优化方法是指通过调整神经网络的参数，使得神经网络的输出结果与实际结果尽可能接近。神经网络最通用的优化方法是梯度下降法，它通过计算损失函数的梯度，调整参数的值，使得损失函数的值不断减小，梯度下降法可以表示为算法 1：

Algorithm 1 梯度下降法

初始化参数 θ
while 模型未收敛 **do**
    计算损失函数 $J(\theta)$ 和梯度 $\nabla J(\theta)$
    更新参数 $\theta = \theta - \alpha \nabla J(\theta)$
end while

其中，θ 是参数，$J(\theta)$ 是损失函数，$\nabla J(\theta)$ 是损失函数的梯度，α 是学习率。具体来说，梯度下降法具体有批量梯度下降法、随机梯度下降法、小批量梯度下降法等。这些变种的区别在于计算梯度的方式，批量梯度下降法是在整个数据集上计算梯度，随机梯度下降法是在单个样本上计算梯度，小批量梯度下降法是在一小部分样本上计算梯度。

梯度下降法是一种常用的优化方法，但是它存在一些问题，如收敛速度慢、容易陷入局部最优解等。此外，梯度下降法对学习率敏感，学习率过大

容易导致震荡，学习率过小容易导致收敛缓慢。为了解决这些问题，研究者提出了一些改进的优化方法，如动量法、Adagrad、RMSprop、Adam 等。这些方法通过引入动量、自适应学习率等机制，提高了优化的效率和稳定性。

动量法是一种常用的优化方法，它通过引入动量项，使得参数更新更加平滑。动量法可以表示为算法 2：

Algorithm 2 动量法

初始化参数 θ，动量参数 β
初始化动量 $v = 0$
while 模型未收敛 **do**
    计算损失函数 $J(\theta)$ 和梯度 $\nabla J(\theta)$
    更新动量 $v = \beta v + (1 - \beta)\nabla J(\theta)$
    更新参数 $\theta = \theta - \alpha v$
end while

其中，β 是动量参数，v 是动量，α 是学习率。动量法通过引入动量机制，动量像是现实生活中的“惯性”，使得当前参数的更新方向不仅取决于当前梯度，还取决于历史梯度，从而减少了参数更新的震荡，提高了优化的效率。

Adam 是目前最常用的优化方法之一，它综合了动量法和 RMSprop 的优点，具有较好的优化效果。Adam 可以表示为算法 3：

Algorithm 3 Adam

初始化参数 θ，动量参数 β_1，RMSprop 参数 β_2
初始化动量 $v = 0$，RMSprop$s = 0$
while 模型未收敛 **do**
    计算损失函数 $J(\theta)$ 和梯度 $\nabla J(\theta)$
    更新动量 $v = \beta_1 v + (1 - \beta_1)\nabla J(\theta)$
    更新 RMSprop$s = \beta_2 s + (1 - \beta_2)(\nabla J(\theta))^2$
    修正动量和 RMSprop$\hat{v} = \frac{v}{1-\beta_1^t}, \hat{s} = \frac{s}{1-\beta_2^t}$
    更新参数 $\theta = \theta - \alpha \frac{\hat{v}}{\sqrt{\hat{s}}+\epsilon}$
end while

其中，β_1 是动量参数，β_2 是 RMSprop 参数，v 是动量，s 是 RMSprop，α 是学习率，ϵ 是平滑项。Adam 通过综合动量法和 RMSprop 的优点，前

者使得参数更新更为平滑，减少参数震荡；后者使得学习率自适应，避免了难以收敛或者陷入局部最优解的问题。

3.3 数据增强

数据增强是一种常用的数据预处理方法，它通过对原始数据进行变换，生成新的数据，从而扩充数据集。数据增强可以提高模型的泛化能力，减少过拟合。数据增强的常见方法包括旋转、翻转、缩放、裁剪、平移、亮度调整等。这些方法可以使得模型对输入数据的变化更加鲁棒，提高模型的泛化能力。

数据增强技术简单而有效，目前取得了巨大的发展。一些新兴的数据增强方法，如 Mixup、CutMix、AutoAugment 等，进一步提高了模型的泛化能力。这些方法通过图像混合和拼接、自动搜索最佳数据增强策略等方式，提高了模型的泛化能力，取得了优异的性能。此外，数据增强技术还可以结合生成对抗网络（GAN）等方法，生成更加真实的数据，提高模型的泛化能力。

4 实验步骤

本章节将介绍本实验的步骤，包括数据预处理和数据增强函数、SimpleNet 模型、模型预测和损失计算函数、SimpleNet with Dropout 模型等。

4.1 数据预处理和数据增强函数

数据预处理和数据增强是神经网络训练的重要步骤，它可以提高模型的泛化能力，减少过拟合。在本实验中，使用了数据增强函数，对数据进行了预处理和增强。数据增强函数的代码如下：

```
def get_fundamental_transforms(inp_size, pixel_mean, pixel_std):
    return transforms.Compose([
        transforms.Resize(inp_size),
        transforms.ToTensor(),
        transforms.Normalize(mean=pixel_mean, std=pixel_std),
    ])

def get_data_augmentation_transforms(inp_size, pixel_mean, pixel_std):
```

```
    return transforms.Compose([
        transforms.RandomResizedCrop(inp_size),
        # transforms.Resize(inp_size),
        transforms.RandomHorizontalFlip(),
        # transforms.RandomVerticalFlip(),
        transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1,
            hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=pixel_mean, std=pixel_std),
    ])
```

上述代码中，`get_fundamental_transforms` 函数用于获取基本的数据预处理函数，包括将图像缩放到指定大小、转换为张量、归一化等操作；`get_data_augmentation_transforms` 函数用于获取数据增强函数，其中可选的操作有随机裁剪、随机水平翻转、随机垂直翻转、颜色抖动等。这些操作可以增加数据的多样性，提高模型的泛化能力。Torchvision 库提供了丰富的数据增强函数，可以根据实际需求选择合适的操作，而且可以轻松实现数据增强函数的聚合。

4.2 SimpleNet

SimpleNet 是一个简单的卷积神经网络，由 2 个卷积层、2 个池化层和全连接层组成。SimpleNet 的代码如下：

```
class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn_layers = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1)),
            nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3)),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1)),
            nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3)),
            nn.ReLU())
        self.fc_layers = nn.Sequential(
            nn.Linear(500, 100),
            nn.Linear(100, 15))

    def forward(self, x: torch.tensor) -> torch.tensor:
```

```
16          x = self.cnn_layers(x)
17          x = x.view(x.size(0), -1)
18          return self.fc_layers(x)
```

上述代码中，SimpleNet 包含了 cnn_layers 和 fc_layers 两个模块。其中，cnn_layers 是卷积神经网络的卷积层和池化层，fc_layers 是全连接层。在 forward 函数中，首先对输入数据进行卷积和池化操作，然后将输出数据展平，最后通过全连接层得到输出结果。

由于 Pytorch 提供了丰富的卷积、池化和全连接层的实现，因此可以轻松构建卷积神经网络模型。例如，网络只需要继承 nn.Module 类，通过 nn.Conv2d、nn.MaxPool2d、nn.Linear 等类实现卷积、池化和全连接层，然后在 forward 函数中定义网络的前向传播过程即可。此外，Pytorch 提供了 nn.Sequential 类，可以方便地构建模块的顺序结构，简化了网络的搭建过程。SimpleNet 的结构如图 2所示。

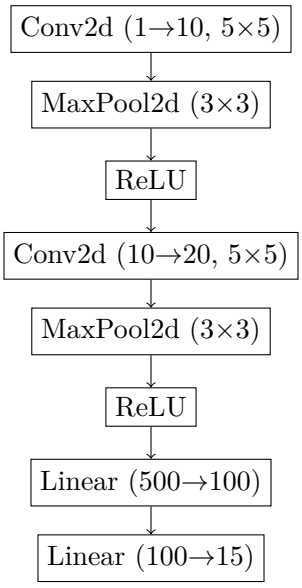


图 2: SimpleNet 网络结构

4.3 模型预测和损失计算函数

```python
def predict_labels(model, x):
    logits = model(x)
    _, predicted_labels = torch. max(logits, 1)
    return predicted_labels


def compute_loss(model, model_output, target_labels, is_normalize):
    # loss = torch.nn.functional.cross_entropy(model_output, target_labels,
        reduction='sum')
    targets = torch.nn.functional.one_hot(target_labels, num_classes=
        model_output.size(1))
    loss = - torch.
        sum(targets * torch.log_softmax(model_output, dim=1), dim=1). sum()
    if is_normalize:
        loss /= model_output.size(0)
    return loss
```

上述代码中，`predict_labels` 函数用于预测模型的输出标签，`compute_loss` 函数用于计算模型的损失。在 `predict_labels` 函数中，首先通过模型计算输出结果，然后取输出结果中概率最大的标签作为预测标签。在 `compute_loss` 函数中，选择了一种手动的方式来计算损失，而不是通过自带的 `cross_entropy` 函数，从而更便于理解计算过程。交叉熵损失是一种常用的损失函数，用于衡量模型输出与真实标签之间的差异，可以表示为公式 4：

$$L = -\sum_{i=1}^{N} y_i \log(p_i) \tag{4}$$

其中，y_i 是真实标签的 one-hot 编码，p_i 是模型输出的概率。在计算之前，需要将预测分布转换为对数概率，然后与真实标签的独热编码相乘，最后求和得到损失。此外，还可以选择是否对损失进行归一化，以便更好地比较不同批次的损失值。

4.4 SimpleNet with Dropout

Dropout 是一种常用的正则化技术，用于减少神经网络的过拟合。在本实验中，对 SimpleNet 模型添加了 Dropout 层，代码如下：

```python
class SimpleNetDropout(nn.Module):
    def __init__(self):
         super().__init__()
        self.cnn_layers = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1), padding=(0, 0)),
            nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3), padding=0),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1), padding=(0, 0)),
            nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3), padding=0),
            nn.ReLU())
        self.fc_layers = nn.Sequential(
            nn.Linear(500, 100),
            nn.Dropout(p=0.5),
            nn.Linear(100, 15))

    def forward(self, x: torch.tensor) -> torch.tensor:
        x = self.cnn_layers(x)
        x = x.view(x.size(0), -1)
        return self.fc_layers(x)
```

上述代码中，`SimpleNetDropout` 是一个带有 Dropout 层的 SimpleNet 模型。在 `fc_layers` 中，添加了一个 Dropout 层，用于随机丢弃一部分神经元，从而减少模型的过拟合。Dropout 层可以在训练过程中随机丢弃神经元，但在测试过程中保留所有神经元，从而提高模型的泛化能力。Dropout 层的参数 p 表示丢弃的概率，通常设置为 0.5。SimpleNet with Dropout 的结构如图 3所示。

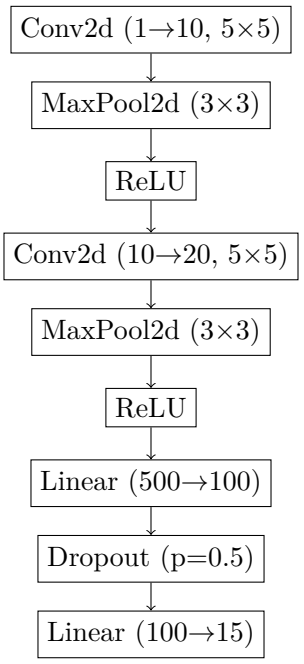


图 3: SimpleNet with Dropout 网络结构

4.5 My Transformer

Transformer 是一种强大的神经网络模型，用于处理序列数据。在本实验中，实现了一个简单的 Transformer 模型，代码如下：

```
import math
import torch
from functools import partial
from torch import nn
from torch.nn.init import trunc_normal_, _calculate_fan_in_and_fan_out


class MLP(nn.Module):
    def __init__(self,
                 in_features,
                 hid_features,
                 out_features,
                 dropout=0.):
        super(MLP, self).__init__()
```

```
15
16          out_features = out_features or in_features
17          hid_features = hid_features or in_features
18
19          self.fc1 = nn.Linear(in_features, hid_features)
20          self.act = nn.GELU()
21          self.fc2 = nn.Linear(hid_features, out_features)
22          self.dropout = nn.Dropout(dropout)
23
24      def forward(self, x):
25          x = self.fc1(x)
26          x = self.act(x)
27          x = self.dropout(x)
28          x = self.fc2(x)
29          x = self.dropout(x)
30          return x
31
32
33  class Attention(nn.Module):
34      def __init__(self,
35                   dim,
36                   n_heads=8,
37                   qkv_bias=False,
38                   attn_dropout=0.,
39                   proj_dropout=0.):
40           super(Attention, self).__init__()
41
42          self.n_heads = n_heads
43          head_dim = dim // n_heads
44          self.scale = head_dim ** -.5
45
46          self.qkv = nn.Linear(dim, dim*3, bias=qkv_bias)
47          self.attn_dropout = nn.Dropout(attn_dropout)
48
49          self.proj = nn.Linear(dim, dim)
50          self.proj_dropout = nn.Dropout(proj_dropout)
51
52      def forward(self, x):
53          b, n, c = x.shape
```

```
54          qkv = self.qkv(x).reshape(b, n, 3, self.n_heads, c//self.n_heads).
                permute(2, 0, 3, 1, 4)
55          q,k,v = qkv[0], qkv[1], qkv[2]
56
57          attn = (q @ k.transpose(-2,-1)) * self.scale
58          attn = attn.softmax(dim=-1)
59          attn = self.attn_dropout(attn)
60
61          x = (attn @ v).transpose(1,2).reshape(b,n,c)
62          x = self.proj(x)
63          x = self.proj_dropout(x)
64          return x
65
66
67  class EncoderBlock(nn.Module):
68      def __init__(self,
69                   dim,
70                   n_heads,
71                   mlp_ratio,
72                   qkv_bias=False,
73                   dropout=0.,
74                   attn_dropout=0.,
75                   proj_dropout=0.,
76                   norm_layer=nn.LayerNorm):
77           super(EncoderBlock, self).__init__()
78
79          self.norm1 = norm_layer(dim)
80          self.attn = Attention(dim, n_heads, qkv_bias,
81                                attn_dropout, proj_dropout)
82          self.dropout = nn.Dropout(dropout)
83
84          self.norm2 = norm_layer(dim)
85          mlp_hid_dim =  int(dim * mlp_ratio)
86          self.mlp = MLP(dim, mlp_hid_dim, dim, dropout)
87
88      def forward(self, x):
89          x = x + self.dropout(self.attn(self.norm1(x)))
90          x = x + self.dropout(self.mlp(self.norm2(x)))
91          return x
```

```
92
93
94  class PatchEmbed(nn.Module):
95
96      def __init__(self, img_size=224,
97                   patch_size=16,
98                   in_c=3,
99                   embed_dim=768,
100                  norm_layer=None):
101          super(PatchEmbed, self).__init__()
102
103         img_size = (img_size, img_size)
104         patch_size = (patch_size, patch_size)
105         self.img_size = img_size
106         self.patch_size = patch_size
107         self.grid_size = (img_size[0] // patch_size[0],
108                           img_size[1] // patch_size[1])
109         self.n_patches = self.grid_size[0] * self.grid_size[1]
110
111         self.proj = nn.Conv2d(in_c, embed_dim,
112                               kernel_size=patch_size,
113                               stride=patch_size)
114         self.norm = norm_layer(embed_dim) if norm_layer else None
115
116     def forward(self, x):
117         # (b,c,h,w) -> (b,c,hw) -> (b,hw,c)
118         x = self.proj(x).flatten(2).transpose(1, 2)
119         if self.norm is not None:
120             x = self.norm(x)
121         return x
122
123
124 def variance_scaling_(tensor, scale=1.0, mode='fan_in', distribution='normal
        '):
125     fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
126     if mode == 'fan_in':
127         denom = fan_in
128     elif mode == 'fan_out':
129         denom = fan_out
```

```
130     elif mode == 'fan_avg':
131         denom = (fan_in + fan_out) / 2
132 
133     variance = scale / denom
134 
135     if distribution == "truncated_normal":
136         # constant is stddev of standard normal truncated to (-2, 2)
137         trunc_normal_(tensor, std=math.sqrt(variance) / .87962566103423978)
138     elif distribution == "normal":
139         tensor.normal_(std=math.sqrt(variance))
140     elif distribution == "uniform":
141         bound = math.sqrt(3 * variance)
142         tensor.uniform_(-bound, bound)
143     else:
144         raise ValueError(f"invalid distribution {distribution}")
145 
146 def lecun_normal_(tensor):
147     variance_scaling_(tensor, mode='fan_in', distribution='truncated_normal'
          )
148 
149 def init_vit_weights(module, name='', head_bias=0.):
150     if  isinstance(module, nn.Linear):
151         if name.startswith('head'):
152             nn.init.zeros_(module.weight)
153             nn.init.constant_(module.bias, head_bias)
154         elif name.startswith('pre_logits'):
155             lecun_normal_(module.weight)
156             nn.init.zeros_(module.bias)
157         else:
158             trunc_normal_(module.weight, std=.02)
159             if module.bias is not None:
160                 nn.init.zeros_(module.bias)
161     elif  isinstance(module, (nn.LayerNorm, nn.GroupNorm, nn.BatchNorm2d)):
162         nn.init.zeros_(module.bias)
163         nn.init.ones_(module.weight)
164 
165 
166 class MyVisionTransformer(nn.Module):
167     def __init__(self,
```

```
168                 img_size=224,
169                 patch_size=16,
170                 in_c=3,
171                 n_classes=1000,
172                 embed_dim=768,
173                 depth=12,
174                 n_heads=12,
175                 mlp_ratio=4.,
176                 qkv_bias=True,
177                 representation_size=None,
178                 proj_dropout=0.,
179                 attn_dropout=0.,
180                 dropout=0.,
181                 norm_layer=None):
182         super().__init__()
183 
184         self.n_classes = n_classes
185         self.n_features = self.embed_dim = embed_dim
186         self.n_tokens = 1
187 
188         norm_layer = norm_layer or partial(nn.LayerNorm, eps=1e-6)
189         self.patch_embed = PatchEmbed(img_size, patch_size, in_c, embed_dim)
190         n_patches = self.patch_embed.n_patches
191 
192         self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
193         self.pos_embed = nn.Parameter(torch.zeros(
194             1, n_patches+self.n_tokens, embed_dim))
195         self.pos_dropout = nn.Dropout(dropout)
196 
197         self.blocks = nn.Sequential(*[
198             EncoderBlock(embed_dim, n_heads, mlp_ratio, qkv_bias,
199                          dropout, attn_dropout, proj_dropout, norm_layer)
200             for _ in  range(depth)
201         ])
202 
203         self.norm = norm_layer(embed_dim)
204         self.head = nn.Linear(self.n_features, n_classes)
205 
206         nn.init.trunc_normal_(self.pos_embed, std=.02)
```

```
207         nn.init.trunc_normal_(self.cls_token, std=.02)
208         self. apply(init_vit_weights)
209 
210     def forward(self, x):
211         x = self.patch_embed(x)
212         cls_token = self.cls_token.expand(x.shape[0], -1, -1)
213         x = torch.cat((cls_token, x), dim=1)
214 
215         x = self.pos_dropout(x + self.pos_embed)
216         x = self.blocks(x)
217         x = self.norm(x)
218         x = self.head(x[:, 0])
219         return x
```

上述代码实现了若干模块,包括 MLP、Attention、EncoderBlock、PatchEmbed：

- MLP：多层感知机模块，包含两个全连接层和激活函数；
- Attention：自注意力模块，用于计算序列数据的注意力权重；
- EncoderBlock：编码器模块，包含注意力和 MLP 两个子模块；
- PatchEmbed：图像块嵌入模块，用于将图像分块并嵌入到向量空间中。

通过连接上述模块，可以构建一个简单的 Transformer 模型。在 MyVisionTransformer 类中，首先通过 PatchEmbed 模块将图像块嵌入到向量空间中，然后通过多个 EncoderBlock 模块进行特征提取，最后通过全连接层得到输出结果。Transformer 模型的结构如图 4所示。

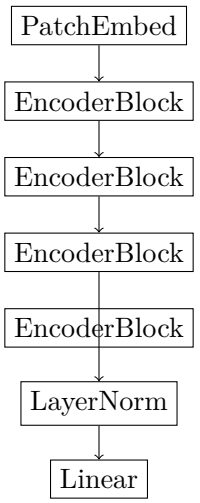


图 4: Transformer 网络结构

4.6 运行脚本

通过运行如下命令启动 jupyter notebook，然后在浏览器中打开链接：

```
jupyter notebook
```

之后打开 `proj4.ipynb` 文件，点击如图 5按钮运行脚本，即可开始实验。

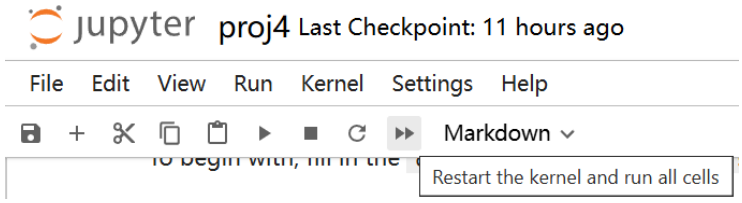


图 5: Jupyter Notebook 界面

运行结果如图 6所示，可以看到所有单元测试均通过，说明实验代码正确。

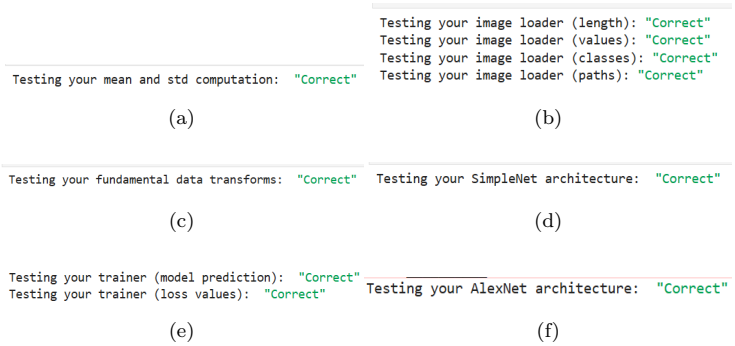


图 6: 单元测试结果

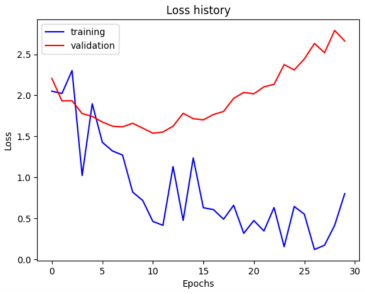
5 实验结果与分析

本章节将介绍本实验的结果与分析，包括实验环境、实验结果、消融实验和实验分析。

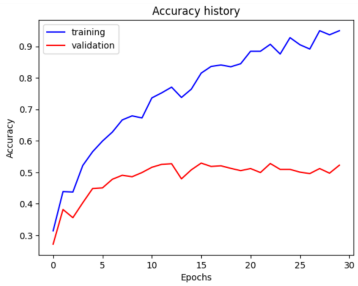
5.1 实验结果

本实验对 SimpleNet、SimpleNet with Dropout（并使用数据增强）、AlexNet 3 种模型进行了训练和测试，得到了如图 7所示的实验结果。如图所示，几种模型呈现出不同的效果，其中：

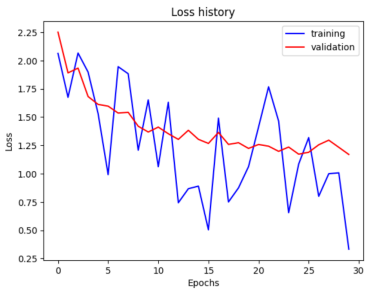
- 如子图 (a)(b) 所示，SimpleNet 在训练集上损失下降较快，但是在测试集上损失先下降后上升；同时 SimpleNet 在训练集上准确率很高，但是在测试集上准确率较低，出现了过拟合现象。
- 如子图 (c)(d) 所示，SimpleNet with Dropout（并使用数据增强）在训练集和测试集上的损失都不断下降，同时在训练集和测试集上的准确率都不断上升。虽然在测试集上的表现低于训练集，但是过拟合现象有所缓解。
- 如子图 (e)(f) 所示，AlexNet 在训练集和测试集上的损失都不断下降，同时在训练集和测试集上的准确率都不断上升。AlexNet 在测试集上的表现显著优于 SimpleNet 和 SimpleNet with Dropout，表明 AlexNet 的泛化能力更强。



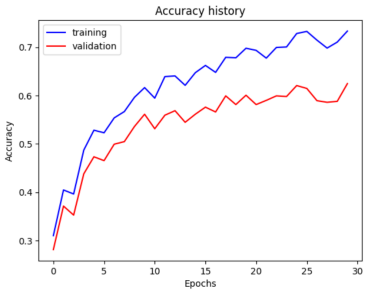
(a) SimpleNet 损失变化



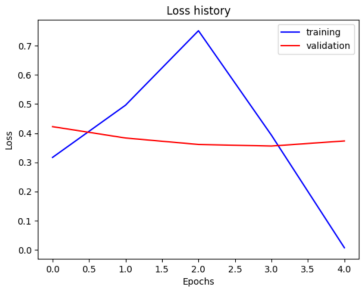
(b) SimpleNet 准确率变化



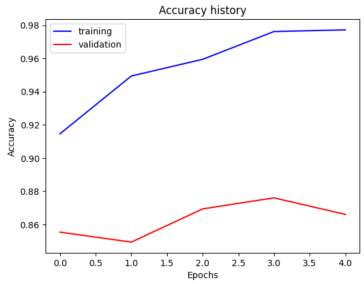
(c) SimpleNet with Dropout 损失变化



(d) SimpleNet with Dropout 准确率变化

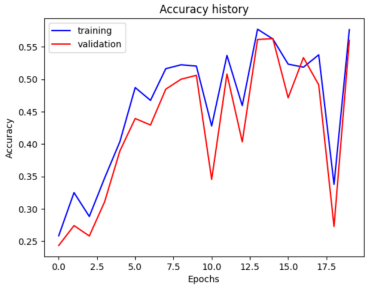


(e) AlexNet 损失变化



(f) AlexNet 准确率变化

(g) Transformer 损失变化



(h) Transformer 准确率变化

图 7: 实验结果

各个模型的性能如表 1所示，可以看出：

- SimpleNet 在测试集上的准确率最低，且出现了严重的过拟合现象。
- SimpleNet with Dropout 在测试集上的准确率次之，且过拟合现象有所缓解。
- AlexNet 在测试集上的准确率相对来说很高，且泛化能力较强。
- 未经预训练的 Transformer 模型在测试集上的准确率较低。
- 经过预训练的 Transformer 模型在测试集上的准确率最高，且泛化能力最强。

表 1: 各模型性能

模型	数据增强	预训练	训练集准确率	测试集准确率
SimpleNet	×	×	0.950	0.523
SimpleNet with Dropout	√	×	0.733	0.625
AlexNet	√	√	0.977	0.866
Transformer	√	×	0.576	0.560
Transformer	√	√	0.898	0.875

5.2 消融实验

为了探究 Dropout 模块，以及数据增强操作对模型性能的影响，本实验进行了消融实验。消融实验的结果如表 2所示，可以看出：

- 在 SimpleNet 基础上加入 Dropout 模块，测试集准确率上升，说明 Dropout 模块可以缓解过拟合现象。
- 在 SimpleNet with Dropout 基础上加入水平翻转操作，测试集准确率上升，说明水平翻转操作可以增加数据多样性。
- 在 SimpleNet with Dropout 基础上加入颜色抖动操作，测试集准确率上升，说明颜色抖动操作可以增加数据多样性。
- 在 SimpleNet with Dropout 基础上加入垂直翻转操作，测试集准确率下降，说明水平、垂直翻转操作不利于模型训练。

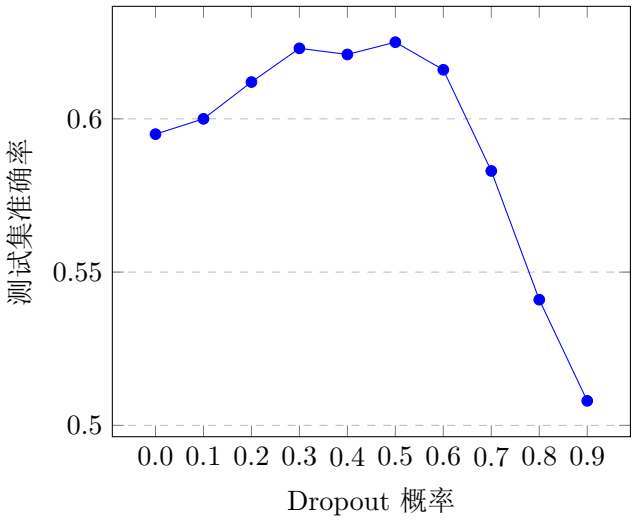


图 8: Dropout 概率与测试集准确率

- 在 SimpleNet with Dropout 基础上加入随机裁剪操作，测试集准确率上升，说明随机裁剪操作可以增加数据多样性。

表 2: 消融实验结果

模型	训练集准确率	测试集准确率
SimpleNet	**0.950**	0.523
+ Dropout	0.896	0.537
+ Dropout、水平翻转	0.755	0.596
+ Dropout、水平翻转、颜色抖动	0.783	0.603
+ Dropout、水平、垂直翻转、颜色抖动	0.650	0.519
+ Dropout、水平翻转、随机裁剪	0.733	**0.625**

此外，本实验还探究了 Dropout 概率对模型性能的影响，如图 8所示，测试集准确率随着 Dropout 概率的增加先上升后下降，在 Dropout 概率为 0.5 时达到最高值，之后快速下降。这说明 Dropout 取 0.5 时最佳，而 Dropout 概率过大会影响模型性能。

5.3 实验分析

本实验通过对 SimpleNet、SimpleNet with Dropout 和 AlexNet 三种模型的训练和测试，探究了 Dropout 模块和数据增强操作对模型性能的影响。实验结果表明：

- Dropout 模块可以缓解过拟合现象，提高模型泛化能力。
- 数据增强操作可以增加数据多样性，提高模型性能。然而，不合理的数据增强操作可能会降低模型性能。例如，垂直翻转操作可能会产生不符合现实情况的图像，从而降低模型性能。
- AlexNet 模型在测试集上的准确率更高，表明设计合理的网络结构可以提高模型性能。
- 未经预训练的 Transformer 模型在测试集上的准确率较低，而预训练之后的 Transformer 模型在测试集上的准确率最高，这是因为 Transformer 模型结构复杂、参数众多，需要大量数据进行训练，因此预训练可以极大提高模型性能。

训练集和测试集上的准确率可以用来评价模型的性能和泛化能力，当训练集上的准确率远高于测试集时，说明模型出现了过拟合现象。本实验中，SimpleNet 模型出现了严重的过拟合现象，而 Dropout 和数据增强操作可以缓解这一现象。此外，预训练可以提高模型性能，增强模型泛化能力。

6 实验结论

本实验通过深度学习对图像数据进行分类，实现了一个简单的图像分类器 SimpleNet，并通过数据增强和 Dropout 操作对其进行了改进，缓解过拟合的问题。此外，本实验还实现了一个经典的卷积神经网络 AlexNet，并对比了这两种模型的性能。从实验结果中可以看出，Dropout 和数据增强可以缓解过拟合现象，此外设计合理的网络结构和预训练有利于提升模型的性能和泛化能力。通过本次实验，我对深度学习的图像分类任务有了更深入的理解，对深度学习的网络结构和训练技巧有了更多的实践经验。

对于本次实验，我有如下建议：

- 实验中只使用了简单的 Dropout 操作，可以尝试更多的正则化方法，如 L1 正则化、L2 正则化等，以缓解过拟合现象。

- 实验中只使用了简单的网络结构，可以尝试更复杂的网络结构，如 ResNet、VGG 等，以提升模型的性能和泛化能力。

A 完整代码

data_transform.py

```
'''
Contains functions with different data transforms
'''

import numpy as np
import torchvision.transforms as transforms

from typing import Tuple


def get_fundamental_transforms(inp_size: Tuple[ int,  int],
                               pixel_mean: np.array,
                               pixel_std: np.array) -> transforms.Compose:
    '''
    Returns the core transforms needed to feed the images to our model

    Args:
    - inp_size: tuple denoting the dimensions for input to the model
    - pixel_mean: the mean  of the raw dataset
    - pixel_std: the standard deviation of the raw dataset
    Returns:
    - fundamental_transforms: transforms.Compose with the fundamental
        transforms
    '''
    return transforms.Compose([
        transforms.Resize(inp_size),
        transforms.ToTensor(),
        transforms.Normalize(mean=pixel_mean, std=pixel_std),
    ])


def get_data_augmentation_transforms(inp_size: Tuple[ int,  int],
                                     pixel_mean: np.array,
                                     pixel_std: np.array) -> transforms.
                                          Compose:
```

```
    '''
    Returns the data augmentation + core transforms needed to be applied on the
    train set

    Args:
    - inp_size: tuple denoting the dimensions for input to the model
    - pixel_mean: the mean  of the raw dataset
    - pixel_std: the standard deviation of the raw dataset
    Returns:
    - aug_transforms: transforms.Compose with all the transforms
    '''
    return transforms.Compose([
        transforms.RandomResizedCrop(inp_size, scale=(0.8, 1.0)),
        # transforms.Resize(inp_size),
        transforms.RandomHorizontalFlip(),
        # transforms.RandomVerticalFlip(),
        transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=pixel_mean, std=pixel_std),
    ])
```

simple_net.py

```
import torch
import torch.nn as nn


class SimpleNet(nn.Module):
    def __init__(self):
        '''
        Init function to define the layers and loss function

        Note: Use 'sum' reduction in the loss_criterion. Read Pytorch documention
        to understand what it means
        '''
         super().__init__()

```

```
15          self.cnn_layers = nn.Sequential(
16              nn.Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1), padding=(0,
                    0)),
17              nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3), padding=0),
18              nn.ReLU(),
19              nn.Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1), padding=(0,
                     0)),
20              nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3), padding=0),
21              nn.ReLU(),
22          )
23          self.fc_layers = nn.Sequential(
24              nn.Linear(500, 100),
25              nn.Linear(100, 15),
26          )
27
28      def forward(self, x: torch.tensor) -> torch.tensor:
29          '''
30          Perform the forward pass with the net
31
32          Args:
33          -   x: the input image [Dim: (N,C,H,W)]
34          Returns:
35          -   y: the output (raw scores) of the net [Dim: (N,15)]
36          '''
37          x = self.cnn_layers(x)
38          x = x.view(x.size(0), -1)
39          return self.fc_layers(x)
```

dl_utils.py

```
1   '''
2   Utilities to be used along with the deep model
3   '''
4
5   import torch
6
7
8   def predict_labels(model: torch.nn.Module, x: torch.tensor) -> torch.tensor:
9       '''
10      Perform the forward pass and extract the labels from the model output
```

```
11
12      Args:
13      -   model: a model (which inherits from nn.Module)
14      -   x: the input image [Dim: (N,C,H,W)]
15      Returns:
16      -   predicted_labels: the output labels [Dim: (N,)]
17      '''
18      logits = model(x)
19      _, predicted_labels = torch. max(logits, 1)
20      return predicted_labels
21
22
23  def compute_loss(model: torch.nn.Module,
24                   model_output: torch.tensor,
25                   target_labels: torch.tensor,
26                   is_normalize:  bool = True) -> torch.tensor:
27      '''
28      Computes the loss between the model output and the target labels
29
30      Args:
31      -   model: a model (which inherits from nn.Module)
32      -   model_output: the raw scores output by the net
33      -   target_labels: the ground truth class labels
34      -   is_normalize: bool flag indicating that loss should be divided by
            the batch size
35      Returns:
36      -   the loss value
37      '''
38      # loss = torch.nn.functional.cross_entropy(model_output, target_labels,
            reduction='sum')
39      targets = torch.nn.functional.one_hot(target_labels, num_classes=
            model_output.size(1))
40      loss = - torch.
            sum(targets * torch.log_softmax(model_output, dim=1), dim=1). sum()
41
42      if is_normalize:
43          loss /= model_output.size(0)
44
45      return loss
```

simple_net_dropout.py

```python
import torch
import torch.nn as nn


class SimpleNetDropout(nn.Module):
    def __init__(self):
        '''
        Init function to define the layers and loss function

        Note: Use 'sum' reduction in the loss_criterion. Read Pytorch documention
        to understand what it means
        '''
         super().__init__()
        self.cnn_layers = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1), padding=(0, 0)),
            nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3), padding=0),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1), padding=(0, 0)),
            nn.MaxPool2d(kernel_size=(3, 3), stride=(3, 3), padding=0),
            nn.ReLU(),
        )
        self.fc_layers = nn.Sequential(
            nn.Linear(500, 100),
            nn.Dropout(p=0.0),
            nn.Linear(100, 15),
        )

    def forward(self, x: torch.tensor) -> torch.tensor:
        '''
        Perform the forward pass with the net

        Args:
        -   x: the input image [Dim: (N,C,H,W)]
        Returns:
        -   y: the output (raw scores) of the net [Dim: (N,15)]
```

```
36          '''
37          x = self.cnn_layers(x)
38          x = x.view(x.size(0), -1)
39          return self.fc_layers(x)
```