模式识别与机器学习大作业
PRML

王滑 尹超 伍昱衡 陈志强 崔祯徐 郑子辰
(按照姓氏笔画排序)
中国科学院大学，北京 10004
University of Chinese Academy of Sciences, Beijing 100049, China

2025.6.3 - 2025.6.30

序言

本文为笔者模式识别与机器学习的大作业。

望老师批评指正。

目录

Chapter 1 神经网络部分

1.1 神经网络简介

1.1.1 关键要点

- 神经网络是一种模拟人类大脑的计算模型，用于模式识别和预测。
- 不同结构如 CNN、RNN、Transformer 各有专长，适合不同任务。
- 研究表明，选择结构取决于数据类型和任务复杂性。

1.1.2 神经网络简介

神经网络（Neural Networks）是一种受生物神经系统启发的机器学习模型，广泛用于分类、回归和生成任务。它由多个节点（神经元）组成，这些节点通过加权连接传递信息，通过训练调整权重以学习数据模式。训练过程包括前向传播、损失计算和反向传播。

1.1.3 不同结构概述

神经网络的结构多样化，每种结构针对特定问题设计。以下是主要类型及其适用场景：

- **前向神经网络（FNN）**：适合静态数据，如基本分类。
- **卷积神经网络（CNN）**：专为图像处理设计，擅长提取空间特征。
- **循环神经网络（RNN）和 LSTM**：处理序列数据，如语言和时间序列。
- **Transformer**：用于自然语言处理，处理长距离依赖。
- **生成对抗网络（GAN）**：生成新数据，如图像生成。

1.1.4 结构差异

不同结构在数据处理方式和复杂性上存在显著差异。例如，CNN 通过卷积层提取图像特征，而 RNN 通过循环捕捉时间依赖。Transformer 则依赖注意力机制，适合并行计算。

1.1.5 详细调研笔记

神经网络的定义与基本原理

神经网络是一种计算模型，模仿人类大脑神经系统的结构和功能，由多个层组成，包括输入层、隐藏层和输出层。每个神经元通过加权连接接收输入，应用激活函数（如 ReLU、Sigmoid）引入非线性，并传递信号。训练过程通过前向传播计算输出，反向传播调整权重以最小化损失函数（如均方误差或交叉熵）。

根据 GeeksforGeeks 的《Neural Networks: A Beginner's Guide》（https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/），神经网络的学习过程包括输入计算、输出生成和参数迭代优化，广泛应用于模式识别和复杂问题解决。

不同神经网络结构的分类与差异

神经网络的结构多样化，以下是主要类型及其特点，基于 V7Labs 的《The Essential Guide to Neural Network Architectures》（https://www.v7labs.com/blog/neural-network-architectures-guide/）和 Wikipedia 的《Neural Network (Machine Learning)》(https://en.wikipedia.org/wiki/Neural_network_(machine_learning)) 的综合分析：

表 1.1: 神经网络结构对比

结构	描述	关键特点	局限性	适用场景
FNN	数据单向流动，无循环	无反馈机制，适合静态数据	无法处理序列数据	基本分类、回归
MLP	FNN 扩展，含隐藏层	处理非线性，学习复杂特征	计算量较大	图像分类、语音识别
CNN	使用卷积和池化层	参数共享，提取空间特征	池化丢失空间关系	图像分析、物体检测、NLP
RNN	处理序列，循环连接	记忆功能，捕捉时间依赖	梯度消失，训练慢	NLP、时间序列预测
LSTM	RNN 增强，记忆单元	解决长序列梯度消失	训练速度慢	语音识别、机器翻译
GAN	生成器与判别器对抗	生成新数据，如图像、文本	训练不稳定	图像生成、数据增强
Transformer	基于注意力机制	处理长距离依赖，适合并行计算	计算复杂度高	NLP、机器翻译
ResNet	深层网络，跳跃连接	解决梯度消失，深层训练	高计算资源	图像分类、目标检测
Hopfield 网络	基于 Hebbian 学习	能量函数驱动，模式检索	不适合训练	模式识别、记忆任务
Boltzmann 机	无监督，生成式模型	随机能量函数，生成任务	训练复杂	深度生成模型
RBF 网络	功能近似，2013 年引入	最佳近似，非线性识别	结构与 MLP 不同	分类、非线性系统
Highway 网络	2015 年，开放门控	训练超深网络，解决退化	与 ResNet 类似	深层网络训练
Capsule 网络	改进 CNN，保留层次	本地胶囊，旋转鲁棒性	实现复杂	空间关系处理
MobileNet	轻量级，适合移动设备	深度可分离卷积	性能受限	移动设备、机器人

结构之间的关键差异

- **数据类型**：CNN 适合空间数据（如图像），RNN/LSTM 适合序列数据（如文本、时间序列），Transformer 适合长文本，GAN 专注于生成数据。
- **处理方式**：FNN 和 MLP 是静态的，RNN/LSTM 有记忆，Transformer 使用自注意力机制，CNN 通过卷积提取特征。
- **复杂性**：FNN 简单，ResNet 和 Transformer 更复杂，适合更深的网络和复杂任务。
- **训练难度**：RNN 存在梯度消失，LSTM 和 ResNet 通过设计解决此问题，Transformer 依赖大规模数据和计算资源。

根据 MyGreatLearning 的《Types of Neural Networks and Definition of Neural Network》（https://www.mygreatlearning.com/blog/types-of-neural-networks/），不同结构的生物启发设计（如 ANN 模仿神经元）决定了其在复杂应用中的表现。

应用场景与局限性

- **CNN**：如 V7Labs 的《Convolutional Neural Networks Guide》（链接）所示，广泛用于图像分类和物体检测，但池化可能丢失空间信息。
- **RNN 和 LSTM**：如 V7Labs 的《Recurrent Neural Networks Guide》（链接）所述，适合 NLP 和时间序列，但训练慢，LSTM 缓解了长序列问题。
- **Transformer**：如《Attention Is All You Need》（链接）所示，主导 NLP 领域，但计算成本高。
- **GAN**：如《Generative Adversarial Networks》（链接）所述，生成高质量图像，但训练不稳定。

综合分析

神经网络的多样性使其能够适应各种任务，从简单的 FNN 到复杂的 Transformer，每种结构都有其独特优势。选择合适结构需考虑数据类型、任务复杂性和计算资源。根据 UpGrad 的《Neural Network Architecture: Types, Components & Key Algorithms》（链接），未来的研究可能进一步优化轻量级网络（如 MobileNet）以适应移动设备。

关键引用

- GeeksforGeeks Neural Networks Beginner's Guide:
  https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/
- V7Labs Essential Guide to Neural Network Architectures:
  https://www.v7labs.com/blog/neural-network-architectures-guide/
- Wikipedia Neural Network Machine Learning:
  https://en.wikipedia.org/wiki/Neural_network_(machine_learning)
- MyGreatLearning Types of Neural Networks Definition:
  https://www.mygreatlearning.com/blog/types-of-neural-networks/
- UpGrad Neural Network Architecture Components Algorithms:
  https://www.upgrad.com/blog/neural-network-architecture-components-algorithms/
- V7Labs Convolutional Neural Networks Guide:
  https://www.v7labs.com/blog/convolutional-neural-networks-guide/
- V7Labs Recurrent Neural Networks Guide:
  https://www.v7labs.com/blog/recurrent-neural-networks-guide/
- Attention Is All You Need Transformer Paper:
  https://arxiv.org/abs/1706.03762
- Generative Adversarial Networks NIPS Paper:
  https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

1.1.6 ResNet 与 Vision Transformer (ViT) 的结构对比

表 1.2: ResNet 与 Vision Transformer (ViT) 的详细结构对比

比较维度	ResNet	Vision Transformer (ViT)
架构类型	卷积神经网络（CNN）	Transformer 架构
提出年份	2015	2020
提出机构	微软研究院	Google Brain
基本单元	卷积层 + 残差连接（Residual Block）	自注意力模块（Multi-head Attention）+ MLP
参数量（Base 模型）	较少（如 ResNet-50 约 25M）	较多（如 ViT-B 约 86M）
计算复杂度	较低，主要是卷积操作	高，自注意力为 $O(n^2)$ 时间复杂度
输入处理	原始图像直接进入卷积网络	图像切成 Patch，再投影为序列
位置建模方式	隐式建模（卷积天然包含位置信息）	显式位置编码（Positional Encoding）
空间建模能力	局部为主，靠堆叠层数扩大全局感受野	全局建模能力强（自注意力机制）
可解释性	较强，可通过卷积特征图分析	较弱，注意力机制不易解释
收敛速度	快速，适合从头训练	慢，对初始化敏感
是否需要预训练	可以从头训练，也支持预训练	强烈依赖预训练（无预训练效果差）
数据规模依赖	中小规模数据也能表现良好	需要大规模数据（如 ImageNet-21k）
训练资源需求	普通 GPU 即可训练（如单卡）	需多卡/TPU，大内存显卡更佳
推理速度	快（卷积并行度高）	慢（序列操作限制并行度）
适合任务	图像分类、目标检测、语义分割等经典视觉任务	大规模视觉任务、跨模态学习、多任务联合建模
代表模型	ResNet-18/34/50/101/152	ViT-B/16, ViT-L/32, DeiT, Swin Transformer

1.2 神经网络 CNN 训练（纯手写第一版）

1.2.1 CNN 训练代码一

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, SubsetRandomSampler
import numpy as np

# 数据预处理 - 增强数据增强
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
```

```python
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

# 加载 CIFAR-10 数据集
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
    transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
    transform=transform_test)

# 划分训练集和验证集
validation_split = 0.2  # 20% 用于验证集
dataset_size = len(trainset)
indices = list(range(dataset_size))
np.random.seed(42)  # 固定随机种子以确保可重复性
np.random.shuffle(indices)
split = int(np.floor(validation_split * dataset_size))
train_indices, val_indices = indices[split:], indices[:split]

# 创建 DataLoader
train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)

trainloader = DataLoader(trainset, batch_size=128, sampler=train_sampler, num_workers
    =2)
valloader = DataLoader(trainset, batch_size=128, sampler=val_sampler, num_workers=2)
testloader = DataLoader(testset, batch_size=128, shuffle=False, num_workers=2)

# 定义改进的 CNN 模型
class ImprovedCNN(nn.Module):
    def __init__(self):
        super(ImprovedCNN, self).__init__()

        # 第一个卷积块
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2)
        )

        # 第二个卷积块
```

```
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2)
        )

        # 第三个卷积块
        self.conv3 = nn.Sequential(
            nn.Conv2d(128, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2)
        )

        # 全连接层
        self.fc = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(256 * 4 * 4, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

# 主程序
if __name__ == '__main__':
    # 实例化模型
    model = ImprovedCNN()

    # 定义损失函数和优化器
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
    # 学习率调度器
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', factor=0.1,
```

```
    patience=5, verbose=True)

    # 如果有 GPU，将模型移动到 GPU
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)
    print(f"使用设备: {device}")

    # 训练模型
    best_val_acc = 0.0
    for epoch in range(100):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(trainloader):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

            if i % 100 == 99:
                print(f'[{epoch + 1}, {i + 1}] loss: {running_loss / 100:.3f} | acc:
    {100.*correct/total:.2f}%')
                running_loss = 0.0

        # 每个epoch结束后验证
        model.eval()
        val_loss = 0
        correct = 0
        total = 0
        with torch.no_grad():
            for data in valloader:
                images, labels = data
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
```

```

        val_acc = 100. * correct / total
        print(f'Epoch {epoch+1}: 验证准确率: {val_acc:.2f}%')

        # 更新学习率
        scheduler.step(val_loss)

        # 保存验证集上最佳模型
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), 'best_cifar10_model.pth')
            print(f'保存最佳模型, 验证准确率: {best_val_acc:.2f}%')

    print('训练完成')

    # 加载最佳模型进行测试
    model.load_state_dict(torch.load('best_cifar10_model.pth'))
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'最佳模型在测试集上的准确率: {100 * correct / total:.2f}%')
```

Listing 1.1: 神经网络 CNN 训练（纯手写第一版）

1.2.2 CNN 训练结果一

```
PS C:\Users\78003\Desktop\prml_re> conda activate yclearning
PS C:\Users\78003\Desktop\prml_re> python cifar10_classifier.py

使用设备: cuda
[1, 100] loss: 1.809 | acc: 33.21%
[1, 200] loss: 1.504 | acc: 38.73%
[1, 300] loss: 1.352 | acc: 42.79%
Epoch 1: 验证准确率: 56.47%
保存最佳模型, 验证准确率: 56.47%
[2, 100] loss: 1.199 | acc: 56.58%
[2, 200] loss: 1.103 | acc: 58.38%
[2, 300] loss: 1.011 | acc: 60.15%
Epoch 2: 验证准确率: 64.90%
保存最佳模型, 验证准确率: 64.90%
```

```
[3, 100] loss: 0.954 | acc: 66.12%
[3, 200] loss: 0.917 | acc: 66.95%
[3, 300] loss: 0.871 | acc: 67.58%
Epoch 3: 验证准确率: 70.78%
保存最佳模型，验证准确率: 70.78%
[4, 100] loss: 0.831 | acc: 70.93%
[4, 200] loss: 0.798 | acc: 71.69%
[4, 300] loss: 0.782 | acc: 71.95%
Epoch 4: 验证准确率: 73.30%
保存最佳模型，验证准确率: 73.30%
[5, 100] loss: 0.739 | acc: 73.97%
[5, 200] loss: 0.731 | acc: 74.01%
[5, 300] loss: 0.714 | acc: 74.43%
Epoch 5: 验证准确率: 76.20%
保存最佳模型，验证准确率: 76.20%
[6, 100] loss: 0.680 | acc: 76.30%
[6, 200] loss: 0.683 | acc: 76.29%
[6, 300] loss: 0.668 | acc: 76.58%
Epoch 6: 验证准确率: 77.85%
保存最佳模型，验证准确率: 77.85%
[7, 100] loss: 0.635 | acc: 77.83%
[7, 200] loss: 0.629 | acc: 78.08%
[7, 300] loss: 0.614 | acc: 78.16%
Epoch 7: 验证准确率: 77.90%
保存最佳模型，验证准确率: 77.90%
[8, 100] loss: 0.570 | acc: 80.14%
[8, 200] loss: 0.592 | acc: 79.82%
[8, 300] loss: 0.578 | acc: 79.85%
Epoch 8: 验证准确率: 74.53%
[9, 100] loss: 0.560 | acc: 80.34%
[9, 200] loss: 0.539 | acc: 80.91%
[9, 300] loss: 0.563 | acc: 80.74%
Epoch 9: 验证准确率: 79.64%
保存最佳模型，验证准确率: 79.64%
[10, 100] loss: 0.531 | acc: 81.48%
[10, 200] loss: 0.518 | acc: 81.74%
[10, 300] loss: 0.520 | acc: 81.86%
Epoch 10: 验证准确率: 79.66%
保存最佳模型，验证准确率: 79.66%
[11, 100] loss: 0.496 | acc: 82.55%
[11, 200] loss: 0.489 | acc: 82.66%
[11, 300] loss: 0.508 | acc: 82.54%
Epoch 11: 验证准确率: 81.02%
保存最佳模型，验证准确率: 81.02%
[12, 100] loss: 0.476 | acc: 83.37%
[12, 200] loss: 0.473 | acc: 83.50%
[12, 300] loss: 0.481 | acc: 83.46%
Epoch 12: 验证准确率: 81.05%
保存最佳模型，验证准确率: 81.05%
```

```
[13, 100] loss: 0.444 | acc: 84.39%
[13, 200] loss: 0.450 | acc: 84.26%
[13, 300] loss: 0.462 | acc: 84.24%
Epoch 13: 验证准确率: 82.22%
保存最佳模型, 验证准确率: 82.22%
[14, 100] loss: 0.439 | acc: 85.30%
[14, 200] loss: 0.439 | acc: 84.95%
[14, 300] loss: 0.431 | acc: 85.12%
Epoch 14: 验证准确率: 82.90%
保存最佳模型, 验证准确率: 82.90%
[15, 100] loss: 0.404 | acc: 86.21%
[15, 200] loss: 0.430 | acc: 85.62%
[15, 300] loss: 0.428 | acc: 85.62%
Epoch 15: 验证准确率: 82.29%
[16, 100] loss: 0.404 | acc: 86.64%
[16, 200] loss: 0.399 | acc: 86.36%
[16, 300] loss: 0.410 | acc: 86.12%
Epoch 16: 验证准确率: 81.61%
[17, 100] loss: 0.367 | acc: 87.30%
[17, 200] loss: 0.396 | acc: 86.84%
[17, 300] loss: 0.394 | acc: 86.74%
Epoch 17: 验证准确率: 84.65%
保存最佳模型, 验证准确率: 84.65%
[18, 100] loss: 0.378 | acc: 86.97%
[18, 200] loss: 0.361 | acc: 87.34%
[18, 300] loss: 0.385 | acc: 87.26%
Epoch 18: 验证准确率: 84.91%
保存最佳模型, 验证准确率: 84.91%
[19, 100] loss: 0.355 | acc: 88.05%
[19, 200] loss: 0.356 | acc: 87.84%
[19, 300] loss: 0.356 | acc: 87.79%
Epoch 19: 验证准确率: 84.83%
[20, 100] loss: 0.349 | acc: 87.92%
[20, 200] loss: 0.351 | acc: 87.86%
[20, 300] loss: 0.345 | acc: 87.98%
Epoch 20: 验证准确率: 83.97%
[21, 100] loss: 0.327 | acc: 88.64%
[21, 200] loss: 0.334 | acc: 88.61%
[21, 300] loss: 0.350 | acc: 88.48%
Epoch 21: 验证准确率: 85.73%
保存最佳模型, 验证准确率: 85.73%
[22, 100] loss: 0.314 | acc: 89.00%
[22, 200] loss: 0.327 | acc: 88.87%
[22, 300] loss: 0.329 | acc: 88.83%
Epoch 22: 验证准确率: 86.06%
保存最佳模型, 验证准确率: 86.06%
[23, 100] loss: 0.303 | acc: 89.61%
[23, 200] loss: 0.317 | acc: 89.28%
[23, 300] loss: 0.313 | acc: 89.32%
```

```
Epoch 23: 验证准确率: 84.69%
[24, 100] loss: 0.296 | acc: 89.80%
[24, 200] loss: 0.296 | acc: 89.88%
[24, 300] loss: 0.305 | acc: 89.72%
Epoch 24: 验证准确率: 86.34%
保存最佳模型, 验证准确率: 86.34%
[25, 100] loss: 0.276 | acc: 90.61%
[25, 200] loss: 0.299 | acc: 90.16%
[25, 300] loss: 0.295 | acc: 90.01%
Epoch 25: 验证准确率: 85.66%
[26, 100] loss: 0.279 | acc: 90.29%
[26, 200] loss: 0.284 | acc: 90.25%
[26, 300] loss: 0.290 | acc: 90.18%
Epoch 26: 验证准确率: 85.39%
[27, 100] loss: 0.264 | acc: 90.84%
[27, 200] loss: 0.276 | acc: 90.66%
[27, 300] loss: 0.285 | acc: 90.42%
Epoch 27: 验证准确率: 86.06%
[28, 100] loss: 0.259 | acc: 91.16%
[28, 200] loss: 0.274 | acc: 90.86%
[28, 300] loss: 0.273 | acc: 90.73%
Epoch 28: 验证准确率: 87.06%
保存最佳模型, 验证准确率: 87.06%
[29, 100] loss: 0.252 | acc: 91.41%
[29, 200] loss: 0.251 | acc: 91.33%
[29, 300] loss: 0.276 | acc: 91.04%
Epoch 29: 验证准确率: 85.27%
[30, 100] loss: 0.247 | acc: 91.32%
[30, 200] loss: 0.248 | acc: 91.41%
[30, 300] loss: 0.254 | acc: 91.36%
Epoch 30: 验证准确率: 85.89%
[31, 100] loss: 0.231 | acc: 91.73%
[31, 200] loss: 0.241 | acc: 91.70%
[31, 300] loss: 0.260 | acc: 91.41%
Epoch 31: 验证准确率: 86.26%
[32, 100] loss: 0.225 | acc: 92.10%
[32, 200] loss: 0.241 | acc: 91.93%
[32, 300] loss: 0.239 | acc: 91.88%
Epoch 32: 验证准确率: 85.79%
[33, 100] loss: 0.223 | acc: 92.39%
[33, 200] loss: 0.227 | acc: 92.32%
[33, 300] loss: 0.240 | acc: 92.12%
Epoch 33: 验证准确率: 86.09%
[34, 100] loss: 0.216 | acc: 92.52%
[34, 200] loss: 0.224 | acc: 92.39%
[34, 300] loss: 0.224 | acc: 92.26%
Epoch 34: 验证准确率: 86.81%
[35, 100] loss: 0.174 | acc: 94.02%
[35, 200] loss: 0.163 | acc: 94.31%
```

```
[35, 300] loss: 0.153 | acc: 94.62%
Epoch 35: 验证准确率: 89.76%
保存最佳模型, 验证准确率: 89.76%
[36, 100] loss: 0.137 | acc: 95.43%
[36, 200] loss: 0.143 | acc: 95.30%
[36, 300] loss: 0.147 | acc: 95.28%
Epoch 36: 验证准确率: 90.03%
保存最佳模型, 验证准确率: 90.03%
[37, 100] loss: 0.127 | acc: 95.79%
[37, 200] loss: 0.133 | acc: 95.71%
[37, 300] loss: 0.134 | acc: 95.64%
Epoch 37: 验证准确率: 89.60%
[38, 100] loss: 0.128 | acc: 95.79%
[38, 200] loss: 0.130 | acc: 95.79%
[38, 300] loss: 0.127 | acc: 95.82%
Epoch 38: 验证准确率: 90.13%
保存最佳模型, 验证准确率: 90.13%
[39, 100] loss: 0.116 | acc: 96.22%
[39, 200] loss: 0.121 | acc: 96.20%
[39, 300] loss: 0.125 | acc: 96.17%
Epoch 39: 验证准确率: 90.29%
保存最佳模型, 验证准确率: 90.29%
[40, 100] loss: 0.113 | acc: 96.48%
[40, 200] loss: 0.115 | acc: 96.36%
[40, 300] loss: 0.120 | acc: 96.22%
Epoch 40: 验证准确率: 90.32%
保存最佳模型, 验证准确率: 90.32%
[41, 100] loss: 0.113 | acc: 96.33%
[41, 200] loss: 0.112 | acc: 96.38%
[41, 300] loss: 0.110 | acc: 96.39%
Epoch 41: 验证准确率: 90.15%
[42, 100] loss: 0.108 | acc: 96.53%
[42, 200] loss: 0.119 | acc: 96.36%
[42, 300] loss: 0.111 | acc: 96.41%
Epoch 42: 验证准确率: 90.12%
[43, 100] loss: 0.103 | acc: 96.80%
[43, 200] loss: 0.113 | acc: 96.61%
[43, 300] loss: 0.114 | acc: 96.49%
Epoch 43: 验证准确率: 90.38%
保存最佳模型, 验证准确率: 90.38%
[44, 100] loss: 0.108 | acc: 96.48%
[44, 200] loss: 0.102 | acc: 96.60%
[44, 300] loss: 0.111 | acc: 96.54%
Epoch 44: 验证准确率: 90.61%
保存最佳模型, 验证准确率: 90.61%
[45, 100] loss: 0.100 | acc: 96.78%
[45, 200] loss: 0.106 | acc: 96.77%
[45, 300] loss: 0.101 | acc: 96.79%
Epoch 45: 验证准确率: 90.46%
```

```
[46, 100] loss: 0.099 | acc: 96.74%
[46, 200] loss: 0.105 | acc: 96.65%
[46, 300] loss: 0.101 | acc: 96.71%
Epoch 46: 验证准确率: 90.20%
[47, 100] loss: 0.096 | acc: 96.93%
[47, 200] loss: 0.102 | acc: 96.77%
[47, 300] loss: 0.100 | acc: 96.77%
Epoch 47: 验证准确率: 90.17%
[48, 100] loss: 0.101 | acc: 96.80%
[48, 200] loss: 0.097 | acc: 96.82%
[48, 300] loss: 0.099 | acc: 96.81%
Epoch 48: 验证准确率: 90.44%
[49, 100] loss: 0.094 | acc: 96.95%
[49, 200] loss: 0.090 | acc: 97.04%
[49, 300] loss: 0.091 | acc: 97.10%
Epoch 49: 验证准确率: 90.40%
[50, 100] loss: 0.089 | acc: 97.23%
[50, 200] loss: 0.092 | acc: 97.12%
[50, 300] loss: 0.090 | acc: 97.15%
Epoch 50: 验证准确率: 90.52%
[51, 100] loss: 0.087 | acc: 97.48%
[51, 200] loss: 0.087 | acc: 97.37%
[51, 300] loss: 0.093 | acc: 97.24%
Epoch 51: 验证准确率: 90.48%
[52, 100] loss: 0.088 | acc: 97.29%
[52, 200] loss: 0.089 | acc: 97.29%
[52, 300] loss: 0.086 | acc: 97.26%
Epoch 52: 验证准确率: 90.55%
[53, 100] loss: 0.090 | acc: 97.21%
[53, 200] loss: 0.089 | acc: 97.25%
[53, 300] loss: 0.087 | acc: 97.23%
Epoch 53: 验证准确率: 90.48%
[54, 100] loss: 0.091 | acc: 97.01%
[54, 200] loss: 0.088 | acc: 97.12%
[54, 300] loss: 0.087 | acc: 97.20%
Epoch 54: 验证准确率: 90.57%
[55, 100] loss: 0.088 | acc: 97.13%
[55, 200] loss: 0.088 | acc: 97.15%
[55, 300] loss: 0.087 | acc: 97.16%
Epoch 55: 验证准确率: 90.77%
保存最佳模型, 验证准确率: 90.77%
[56, 100] loss: 0.086 | acc: 97.30%
[56, 200] loss: 0.084 | acc: 97.47%
[56, 300] loss: 0.083 | acc: 97.46%
Epoch 56: 验证准确率: 90.63%
[57, 100] loss: 0.089 | acc: 97.21%
[57, 200] loss: 0.086 | acc: 97.24%
[57, 300] loss: 0.087 | acc: 97.18%
Epoch 57: 验证准确率: 90.71%
```

```
[58, 100] loss: 0.086 | acc: 97.09%
[58, 200] loss: 0.083 | acc: 97.28%
[58, 300] loss: 0.085 | acc: 97.27%
Epoch 58: 验证准确率: 90.69%
[59, 100] loss: 0.083 | acc: 97.47%
[59, 200] loss: 0.077 | acc: 97.57%
[59, 300] loss: 0.089 | acc: 97.42%
Epoch 59: 验证准确率: 90.40%
[60, 100] loss: 0.081 | acc: 97.55%
[60, 200] loss: 0.085 | acc: 97.43%
[60, 300] loss: 0.084 | acc: 97.40%
Epoch 60: 验证准确率: 90.52%
[61, 100] loss: 0.084 | acc: 97.29%
[61, 200] loss: 0.080 | acc: 97.38%
[61, 300] loss: 0.082 | acc: 97.40%
Epoch 61: 验证准确率: 90.67%
[62, 100] loss: 0.084 | acc: 97.35%
[62, 200] loss: 0.084 | acc: 97.37%
[62, 300] loss: 0.088 | acc: 97.34%
Epoch 62: 验证准确率: 90.44%
[63, 100] loss: 0.090 | acc: 97.15%
[63, 200] loss: 0.079 | acc: 97.42%
[63, 300] loss: 0.085 | acc: 97.39%
Epoch 63: 验证准确率: 90.39%
[64, 100] loss: 0.083 | acc: 97.39%
[64, 200] loss: 0.086 | acc: 97.25%
[64, 300] loss: 0.084 | acc: 97.24%
Epoch 64: 验证准确率: 90.70%
[65, 100] loss: 0.093 | acc: 97.04%
[65, 200] loss: 0.086 | acc: 97.08%
[65, 300] loss: 0.081 | acc: 97.21%
Epoch 65: 验证准确率: 90.24%
[66, 100] loss: 0.088 | acc: 97.16%
[66, 200] loss: 0.082 | acc: 97.29%
[66, 300] loss: 0.083 | acc: 97.32%
Epoch 66: 验证准确率: 90.48%
[67, 100] loss: 0.084 | acc: 97.45%
[67, 200] loss: 0.086 | acc: 97.33%
[67, 300] loss: 0.078 | acc: 97.41%
Epoch 67: 验证准确率: 90.39%
[68, 100] loss: 0.083 | acc: 97.55%
[68, 200] loss: 0.085 | acc: 97.44%
[68, 300] loss: 0.079 | acc: 97.50%
Epoch 68: 验证准确率: 90.56%
[69, 100] loss: 0.082 | acc: 97.55%
[69, 200] loss: 0.085 | acc: 97.42%
[69, 300] loss: 0.083 | acc: 97.43%
Epoch 69: 验证准确率: 90.32%
[70, 100] loss: 0.084 | acc: 97.33%
```

```
[70, 200] loss: 0.087 | acc: 97.27%
[70, 300] loss: 0.088 | acc: 97.21%
Epoch 70: 验证准确率: 90.07%
[71, 100] loss: 0.089 | acc: 97.13%
[71, 200] loss: 0.085 | acc: 97.19%
[71, 300] loss: 0.089 | acc: 97.18%
Epoch 71: 验证准确率: 90.63%
[72, 100] loss: 0.085 | acc: 97.14%
[72, 200] loss: 0.082 | acc: 97.31%
[72, 300] loss: 0.084 | acc: 97.33%
Epoch 72: 验证准确率: 90.39%
[73, 100] loss: 0.082 | acc: 97.50%
[73, 200] loss: 0.085 | acc: 97.50%
[73, 300] loss: 0.086 | acc: 97.45%
Epoch 73: 验证准确率: 90.45%
[74, 100] loss: 0.083 | acc: 97.33%
[74, 200] loss: 0.086 | acc: 97.33%
[74, 300] loss: 0.082 | acc: 97.33%
Epoch 74: 验证准确率: 90.37%
[75, 100] loss: 0.083 | acc: 97.50%
[75, 200] loss: 0.085 | acc: 97.44%
[75, 300] loss: 0.086 | acc: 97.33%
Epoch 75: 验证准确率: 90.35%
[76, 100] loss: 0.083 | acc: 97.48%
[76, 200] loss: 0.080 | acc: 97.46%
[76, 300] loss: 0.089 | acc: 97.41%
Epoch 76: 验证准确率: 90.45%
[77, 100] loss: 0.079 | acc: 97.59%
[77, 200] loss: 0.085 | acc: 97.39%
[77, 300] loss: 0.088 | acc: 97.33%
Epoch 77: 验证准确率: 90.40%
[78, 100] loss: 0.079 | acc: 97.53%
[78, 200] loss: 0.082 | acc: 97.51%
[78, 300] loss: 0.091 | acc: 97.34%
Epoch 78: 验证准确率: 90.32%
[79, 100] loss: 0.084 | acc: 97.15%
[79, 200] loss: 0.084 | acc: 97.23%
[79, 300] loss: 0.081 | acc: 97.31%
Epoch 79: 验证准确率: 90.33%
[80, 100] loss: 0.088 | acc: 97.33%
[80, 200] loss: 0.083 | acc: 97.33%
[80, 300] loss: 0.080 | acc: 97.46%
Epoch 80: 验证准确率: 90.48%
[81, 100] loss: 0.085 | acc: 97.38%
[81, 200] loss: 0.084 | acc: 97.32%
[81, 300] loss: 0.085 | acc: 97.24%
Epoch 81: 验证准确率: 90.67%
[82, 100] loss: 0.081 | acc: 97.54%
[82, 200] loss: 0.082 | acc: 97.45%
```

```
[82, 300] loss: 0.082 | acc: 97.48%
Epoch 82: 验证准确率: 90.43%
[83, 100] loss: 0.079 | acc: 97.72%
[83, 200] loss: 0.082 | acc: 97.52%
[83, 300] loss: 0.087 | acc: 97.45%
Epoch 83: 验证准确率: 90.40%
[84, 100] loss: 0.083 | acc: 97.50%
[84, 200] loss: 0.082 | acc: 97.45%
[84, 300] loss: 0.084 | acc: 97.41%
Epoch 84: 验证准确率: 90.07%
[85, 100] loss: 0.083 | acc: 97.30%
[85, 200] loss: 0.084 | acc: 97.27%
[85, 300] loss: 0.085 | acc: 97.25%
Epoch 85: 验证准确率: 90.24%
[86, 100] loss: 0.085 | acc: 97.32%
[86, 200] loss: 0.081 | acc: 97.38%
[86, 300] loss: 0.083 | acc: 97.38%
Epoch 86: 验证准确率: 90.34%
[87, 100] loss: 0.087 | acc: 97.26%
[87, 200] loss: 0.085 | acc: 97.33%
[87, 300] loss: 0.088 | acc: 97.31%
Epoch 87: 验证准确率: 90.45%
[88, 100] loss: 0.085 | acc: 97.38%
[88, 200] loss: 0.084 | acc: 97.32%
[88, 300] loss: 0.085 | acc: 97.33%
Epoch 88: 验证准确率: 90.38%
[89, 100] loss: 0.084 | acc: 97.45%
[89, 200] loss: 0.080 | acc: 97.48%
[89, 300] loss: 0.085 | acc: 97.39%
Epoch 89: 验证准确率: 90.62%
[90, 100] loss: 0.087 | acc: 97.17%
[90, 200] loss: 0.085 | acc: 97.19%
[90, 300] loss: 0.082 | acc: 97.32%
Epoch 90: 验证准确率: 90.41%
[91, 100] loss: 0.085 | acc: 97.27%
[91, 200] loss: 0.086 | acc: 97.31%
[91, 300] loss: 0.086 | acc: 97.30%
Epoch 91: 验证准确率: 90.59%
[92, 100] loss: 0.085 | acc: 97.30%
[92, 200] loss: 0.087 | acc: 97.32%
[92, 300] loss: 0.082 | acc: 97.39%
Epoch 92: 验证准确率: 90.71%
[93, 100] loss: 0.089 | acc: 97.17%
[93, 200] loss: 0.085 | acc: 97.23%
[93, 300] loss: 0.086 | acc: 97.25%
Epoch 93: 验证准确率: 90.59%
[94, 100] loss: 0.084 | acc: 97.45%
[94, 200] loss: 0.088 | acc: 97.39%
[94, 300] loss: 0.087 | acc: 97.31%
```

```
Epoch 94: 验证准确率: 90.52%
[95, 100] loss: 0.089 | acc: 97.12%
[95, 200] loss: 0.083 | acc: 97.21%
[95, 300] loss: 0.083 | acc: 97.21%
Epoch 95: 验证准确率: 90.66%
[96, 100] loss: 0.085 | acc: 97.41%
[96, 200] loss: 0.082 | acc: 97.48%
[96, 300] loss: 0.080 | acc: 97.49%
Epoch 96: 验证准确率: 90.63%
[97, 100] loss: 0.086 | acc: 97.29%
[97, 200] loss: 0.081 | acc: 97.41%
[97, 300] loss: 0.088 | acc: 97.37%
Epoch 97: 验证准确率: 90.34%
[98, 100] loss: 0.083 | acc: 97.54%
[98, 200] loss: 0.086 | acc: 97.45%
[98, 300] loss: 0.079 | acc: 97.49%
Epoch 98: 验证准确率: 90.28%
[99, 100] loss: 0.081 | acc: 97.44%
[99, 200] loss: 0.087 | acc: 97.30%
[99, 300] loss: 0.084 | acc: 97.33%
Epoch 99: 验证准确率: 90.43%
[100, 100] loss: 0.083 | acc: 97.47%
[100, 200] loss: 0.084 | acc: 97.43%
[100, 300] loss: 0.085 | acc: 97.39%
Epoch 100: 验证准确率: 90.59%
训练完成
最佳模型在测试集上的准确率: 90.60%
```

Listing 1.2: CNN 训练结果一

1.3 Summary

Introduction

The CIFAR-10 dataset, comprising 60,000 32x32 color images across 10 classes (e.g., airplane, automobile, bird), is a benchmark for image classification, with 50,000 training and 10,000 test images. This report compares two neural network models—ResNet and Vision Transformer (ViT)—on CIFAR-10, evaluating training from scratch and fine-tuning pretrained models. We analyze performance, computational resources, and experimental outcomes, supported by code implementations.

Dataset and Models

CIFAR-10 Dataset

CIFAR-10 includes 10 classes with 6,000 images each, split into 50,000 training and 10,000 test images, evenly distributed. Its low resolution (32x32) tests model performance on small datasets.

ResNet

Residual Networks (ResNet) use skip connections to ease deep network training. ResNet-110 (1.7M parameters) and ResNet50 (25M parameters) are evaluated.

Vision Transformer (ViT)

ViT splits images into patches and applies Transformer architecture. ViT-B/16 (86M parameters) relies on large-scale pretraining for optimal performance.

Data Splitting and Preprocessing

The dataset is split into 50,000 training and 10,000 test images. Preprocessing includes:

- **Training**: Random cropping (32x32, 4-pixel padding), random horizontal flipping, normalization (mean [0.4914, 0.4822, 0.4465], std [0.2023, 0.1994, 0.2010]).
- **Testing**: Normalization only.

Training from Scratch

Training from scratch uses no pretrained weights. ResNet outperforms ViT:

- **ResNet-110**: Achieves 93.57% accuracy (1.7M parameters), efficient training [1].
- **ViT**: Standard ViT reaches 77–88% accuracy; optimized versions hit 90.92% (6.3M parameters) [3, 4].

Fine-Tuning Pretrained Models

Pretrained models are fine-tuned on CIFAR-10 after training on large datasets:

- **ResNet50 (ImageNet-1k)**: 92.34–92.63% accuracy [5].
- **ViT base (ImageNet-1k)**: 98.5% accuracy [4].

表 1.3: Accuracy and Parameters for Training from Scratch

Model	Accuracy (%)	Parameters (M)	Notes
ResNet-110	93.57	1.7	[1]
ViT (Standard)	77–88	6.3	Varies by configuration [4]
ViT (Optimized)	90.92	6.3	[3]

- **ViT-H/14 (JFT-300M)**: 99.50% accuracy; ViT-L/16 (ImageNet-21k): 99.15% [2].
- **BiT-L (ResNet152x4, JFT-300M)**: 99.37% [2].

表 1.4: Accuracy for Fine-Tuned Pretrained Models

Model	Pretraining Dataset	Accuracy (%)	Notes
ResNet50	ImageNet-1k	92.34–92.63	[5]
ViT base	ImageNet-1k	98.5	[4]
ViT-H/14	JFT-300M	99.50	[2]
ViT-L/16	ImageNet-21k	99.15	[2]
BiT-L (ResNet152x4)	JFT-300M	99.37	[2]

Computational Resources and Performance

- **ResNet**: Fewer parameters (1.7M for ResNet-110, 25M for ResNet50), fast convergence (90% accuracy in 5 epochs) [7].
- **ViT**: More parameters (86M for ViT-B/16), slower initial learning (10,000 iterations to stabilize), but efficient fine-tuning [6, 2].

Experimental Analysis

Error Analysis

Misclassified images should be analyzed. ViT may excel in context-heavy classes (e.g., cat vs. dog) due to global attention, while ResNet performs better on texture details (e.g., airplane) [4].

Performance Gap Causes

CNNs (ResNet) leverage inductive biases (e.g., translation invariance), suiting small datasets. ViT requires more data to learn patterns [6].

Algorithmic Trade-Offs

- **ResNet**: Efficient, fewer parameters, ideal for small datasets; less scalable on large datasets.
- **ViT**: Superior with large-scale pretraining, flexible, but resource-intensive and prone to overfitting on small datasets.

Visualization

Accuracy can be visualized using a bar plot. Below is a Python snippet for comparison:

```python
import matplotlib.pyplot as plt
models = ['ResNet-110', 'ViT (Standard)', 'ViT (Optimized)']
accuracies = [93.57, 88, 90.92]
plt.bar(models, accuracies, color=['#1f77b4', '#ff7f0e', '#2ca02c'])
plt.xlabel('Model')
plt.ylabel('Accuracy (\%)')
plt.title('CIFAR-10 Accuracy (Training from Scratch)')
plt.show()
```

Quantitative Metrics

- **Accuracy**: See Tables 1 and 2.
- **Parameters**: ResNet-110 (1.7M), ResNet50 (25M), ViT-B/16 (86M), ViT (Optimized, 6.3M).
- **Training Time**: ResNet-110 reaches 90%+ in hours; ViT requires longer (10,000 iterations).
- **Resources**: ViT pretraining needs high-end GPUs (e.g., V100); ResNet trains efficiently on standard GPUs.

Code Implementations

ResNet Training

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
    transform=transform_train)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
    transform=transform_test)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True)
```

```python
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=False)

model = torchvision.models.resnet18(pretrained=False, num_classes=10)
model.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
model.maxpool = nn.Identity()
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9, weight_decay=1e-4)

num_epochs = 50
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}'
    )

model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
print(f'Test Accuracy: {100 * correct / total}\%')
```

ViT Fine-Tuning

```python
import torch
from transformers import ViTForImageClassification, ViTFeatureExtractor
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
```

```
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

train_dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=
    transform)
test_dataset = datasets.CIFAR10(root='./data', train=False, download=True, transform=
    transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

model = ViTForImageClassification.from_pretrained('google/vit-base-patch16-224-in21k',
    num_labels=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)
criterion = torch.nn.CrossEntropyLoss()

num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images).logits
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}'
    )

model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images).logits
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
print(f'Test Accuracy: {100 * correct / total}\%')
```

Conclusion

ResNet-110 excels in training from scratch on CIFAR-10 (93.57% accuracy, low computational cost). Pretrained ViT models, especially ViT-H/14 fine-tuned from JFT-300M (99.50%), outperform ResNet (BiT-L, 99.37%). Model choice depends on data scale and resources: ResNet suits limited data/resources, while ViT excels with large-scale pretraining. Code implementations are provided for reproducibility.

参考文献

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385*, 2016.

[2] A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *arXiv:2010.11929*, 2020.

[3] ViT-CIFAR Repository, https://github.com/omihub777/ViT-CIFAR.

[4] vision-transformers-cifar10 Repository, https://github.com/kentaroy47/vision-transformers-cifar10.

[5] Fine-Tuning ResNet50 Pretrained on ImageNet for CIFAR-10, https://github.com/sidthoviti/Fine-Tuning-ResNet50-Pretrained-on-ImageNet-for-CIFAR-10.

[6] PyTorch Lightning, "Fine-tuning Vision Transformer on CIFAR-10," https://colab.research.google.com/github/NielsRogge/Transformers-Tutorials/blob/master/VisionTransformer/Fine_tuning_the_Vision_Transformer_on_CIFAR_10_with_PyTorch_Lightning.ipynb.

[7] PyTorch Forum, "ResNet with CIFAR10 only reaches 86% accuracy," https://discuss.pytorch.org/t/resnet-with-cifar10-only-reaches-86-accuracy-expecting-90/135051.

附录 A. 中英文对照表

A.1 中英文对照表

表 A.5: 中英文对照表

英文	中文
Bayes classification	贝叶斯分类
decision rule	决策规则
minimum error rate	最小错误率
minimum risk	最小风险
rejection option	拒识选项
Gaussian distribution	高斯分布
covariance matrix	协方差矩阵
discriminant function	判别函数
decision boundary	决策边界
Hidden Markov Model	隐马尔可夫模型
hidden state	隐状态
observation sequence	观测序列
maximum likelihood estimation	最大似然估计
Bayesian estimation	贝叶斯估计
k-Nearest Neighbor	k 近邻
Parzen window	Parzen 窗
linear discriminant	线性判别
quadratic discriminant	二次判别
prior probability	先验概率
posterior probability	后验概率