Python与深度学习的基础

NumPy的学习

1.NumPy的介绍

- NumPy是Python中用于科学计算的核心库。它是 Python 语言的一个扩展程序库，支持大量的多维数组与矩阵运算，并提供丰富的数学函数库。
- NumPy 的核心是 `ndarray` 对象——一个功能强大的 N 维同构数组，封装了 n 维同类数据。大多数运算由底层编译代码（如 C/C++）执行，从而显著提升计算效率。该库包含多维数组和矩阵数据结构，提供了对 `ndarray` 进行高效操作的方法。

2.张量与数组的概念

- 在科学计算与深度学习中，“数组”和“张量”常常被用来描述数据结构，以下表格是对维度数据的介绍

维度	名称	特点描述	示例
0维	标量	只有一个数值，无方向	身高，体重
1维	向量	有大小有方向	数学中的坐标，通过Word2Vec后的词向量
2维	矩阵	有行和列组成的二维数组	灰度图像，二维表格
≥3维	张量	三维及以上的数组，在深度学习中同称为任意维度是数组	彩色图像（高度，宽度，通道数），文本批次（批次大小，序列长度，词向量维度）

3.NumPy的常见使用

1. numpy的下载

   `conda install numpy` or `pip install numpy`

2. numpy库的调用

   `import numpy as np`

3. 数组的创建

   > **一维数组的创建,并查看该数组的形状与类型**
   >
   > `a=np.array([1,2,3,4])`

> `a.shape`
>
> `a.dtype`
>
> 输出结果：a.shape=(4,);a.dtype=("int32")

> **二维数据的创建,并查看改数组的形状与类型**
>
> `a1=np.array([[1,1,1],[2,2,2],[3,3,3]])`
>
> 使用a1.shape查看新建数组的形状结果维（3，3）

> **其他类型数组的创建**
>
> > **创建一个由零组成的数组**
> >
> > `a=np.zeros(2，dtype=np.float64)` == `a=np.array([0,0],dtype=np.float64)`创建一个以0填充的形状为（2，）的数组,类型为float64，该类型根据想要的数据需求设置。
> >
> > `a.np.zeros((2,3))`创建一个形状为2x3并且以0填充的二维数组
>
> > **创建一个有1组成的数组**
> >
> > `a=np.ones(2,dtype=np.dtype64)`
> >
> > 输出结果:`array([1.,1.])`
> >
> > `a=np.ones((2,3))`
> >
> > 输出结果：`array([[1.,1.,1.],[1.,1.,1.]])`
>
> > **使用函数empty创建一个数组，其初始内容是随机的，并且取决于内存的状态**
> >
> > `a=np.empty(2)`
>
> > **使用arange函数生成数组**
> >
> > `a=np.arange(4)`
> >
> > 输出结果：`array([0,1,2,3])`
> >
> > `a=np.arange(1,9,2)`
> >
> > 表示：生成一个从1开始步长为2，到第9个数结束的一维数组
> >
> > 输出结果：`array([1,3,5,7])`

4. 数组的排序

   `a=np.array([2,6,1,4,8])`

   `np.sort(a)`

   输出结果：`array([1,2,4,6,8])`

5. 数组的连接

   `a=np.array([[1,2],[3,4],[5,6]])`

   `b=np.array([[7,8]])`

   `c=np.concatenate((a,b),axis=0)`

   输出结果：`array([[1,2],[3,4],[5,6],[7,8]])`

6. 数组的访问（数组访问都是从0开始的）

> **一维数组的访问**
>
> `a=np.array([2,6,1,4,8])`

> `a[0]` 结果为2
>
> `a[0:2]` 结果为：`array([2,6])`
>
> `a[-2:]` 结果为：`array([4,8])`

> **二维数组的访问**
>
> `a=np.array([[1,2],[3,4],[5,6]])`
>
> `a[0][1]` == `a[0,1]` 结果为2
>
> `a[0,]` 结果为`array([1,2])`
>
> `a[0:3,0]` 结果为 `array([1, 3, 5])`
>
> `b=a[a<5]` 结果为`array([1, 2, 3, 4])`

7. 数组的运算
   - 数组的最基本的用法就是,直接使用"-","+","*","/"这些符号直接对数据进行运算,也就是相对应位置的加减乘除.
   - 数组的其他求和运算方式

     > `a=np.array([[1,1],[2,2]])`
     >
     > 输出结果:array([[1, 2], [1, 2]])
     >
     > `a.sum(axis=0)` 按数组的行轴进行求和.输出结果`array([3, 3])`
     >
     > `a.sum(axis=1)` 按数组的列轴进行求和.输出结果`array([2, 4])`

   - 数组的最大值,最小值,总和

     `a.max()`,`a,min()`,`a.sum()`
   - 查看数组中的唯一值

     `np.unique(a)` 输出结果为:`array([1,2])`
   - 数组的转置

     `a.T` 输出结果:array([[1, 2], [1, 2]])
   - 翻转数组

     > 翻转一维数组
     >
     > `a=np.array([2,6,1,4,8])`
     >
     > `np.flip(a)` 输出结果: array([8,4,1,6,2])

     > 翻转二维数组
     >
     > `a=np.array([[1,1],[2,2]])` 输出结果: array([[1, 1],[2, 2]])
     >
     > `np.flip(a)` 输出结果: array([[1, 1],[2, 2]]

8. 保存与加载数组
   - `np.savetxt("filename.csv",a,delimiter=",")` 以",分隔"
   - `np.load('filename.csv')`
9. 其他常见命令

代码	用途
`a.shape`	查看数组a的形状
`a.dtype`	查看数组a的类型
`a.ndim`	查看数组的维数
`a.size`	数组元素的总数(也就是数组中有多少个数)
`a=c.reshape(x,y)`	将数组的形状重新更改为x行y列

PyTorch的学习

1.PyTorch的介绍

- PyTorch 是一个开源的 Python 机器学习库，基于 Torch 库，底层由 C++ 实现，应用于人工智能领域，如计算机视觉和自然语言处理。PyTorch 最初由 Meta Platforms 的人工智能研究团队开发，现在属 于Linux 基金会的一部分。
- 许多深度学习软件都是基于 PyTorch 构建的，包括特斯拉自动驾驶、Uber 的 Pyro、Hugging Face 的 Transformers、 PyTorch Lightning 和 Catalyst。
- PyTorch 主要有两大特征：类似于 NumPy 的张量计算，能在 GPU 或 MPS 等硬件加速器上加速。基于带自动微分系统的深度神经网络。PyTorch 包括 torch.autograd、torch.nn、torch.optim 等子模块。
  PyTorch 包含多种损失函数，包括 MSE（均方误差 = L2 范数）、交叉熵损失和负熵似然损失（对分类器有用）等。

2.Pytorch的常见用法

- torch的安装

  `conda install torch` or `pip insatll torch`

  安装不成功可以去网上找镜像

  查看是否安装成功

  `print("PyTorch 版本", torch._version_)`

  输出结果：`PyTorch 版本: 1.13.1+cpu`

- 创建张量

```
# 从列表创建
x = torch.tensor([1, 2, 3, 4])
print("从列表创建:", x)
```

输出结果：`从列表创建: tensor([1, 2, 3, 4])`这个与numpy的创建数组一样的语法

```
# 创建全一张量
ones = torch.ones(2, 3)
print("全一张量:\n", ones)
```

输出结果：`tensor([[1., 1., 1.],`

`[1., 1., 1.]])`

```
# 创建随机张量
rand_tensor = torch.rand(2, 3)
print("随机张量:\n", rand_tensor)
```

输出结果：`tensor([[0.6817, 0.5395, 0.3635],`

`[0.7416, 0.4544, 0.3653]])`

```
# 创建范围张量
arange = torch.arange(0, 10, 2)
print("范围张量:", arange)
```

输出结果：`tensor([0, 2, 4, 6, 8])`

- 张量的属性

```
tensor = torch.rand(3, 4)
print("张量形状:", tensor.shape)
print("张量维度:", tensor.dim())
print("张量数据类型:", tensor.dtype)
print("张量设备:", tensor.device)
```

输出结果：张量形状: torch.Size([3, 4])

张量维度: 2

张量数据类型: torch.float32

张量设备: cpu

- 张量是基本运算

```
#这些就像是对张量的相对于位置进行运算
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
print("a =", a)
print("b =", b)
print("a + b =", a + b)
print("a - b =", a - b)
print("a * b =", a * b)  # 逐元素乘法
print("a / b =", a / b)
print("a ** 2 =", a ** 2)
```

输出结果：`a = tensor([1, 2, 3])`,`b = tensor([4, 5, 6])`,`a + b = tensor([5, 7, 9])`

`a - b = tensor([-3, -3, -3])`,`a * b = tensor([ 4, 10, 18])`,

`a/ b = tensor([0.2500, 0.4000, 0.5000])`,`a ** 2 = tensor([1, 4, 9])`

- **矩阵乘法**

```
matrix_a = torch.rand(2, 3)
matrix_b = torch.rand(3, 2)
matrix_product = torch.matmul(matrix_a, matrix_b)
print(f"矩阵乘法: {matrix_a.shape} @ {matrix_b.shape} =
{matrix_product.shape}")
```

输出结果：`矩阵乘法: torch.Size([2, 3]) @ torch.Size([3, 2]) = torch.Size([2, 2])`

- 修改张量的形状

```
tensor = torch.arange(12)
reshaped = tensor.reshape(3, 4)
print("原始张量:", tensor)
print("reshape(3, 4):\n", reshaped)
```

输出结果：

`原始张量: tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])`

```
reshape(3, 4):
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

```
# 使用view方法也可以改变张量的形状
viewed = tensor.view(3, 4)
print("view(3, 4):\n", viewed)
```

输出结果：

```
view(3, 4):
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

- 张量的转置

```
transposed = reshaped.T
print("转置:\n", transposed)
```

输出结果：

```
转置:
 tensor([[ 0,  4,  8],
        [ 1,  5,  9],
        [ 2,  6, 10],
        [ 3,  7, 11]])
```

- 张量的索引与切片

```
tensor = torch.arange(24).reshape(4, 6)
print("原始张量:\n", tensor)
print("第一行:", tensor[0])
print("第一列:", tensor[:, 0])
print("子矩阵:\n", tensor[1:3, 2:4])
```

输出结果:

```
 tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17],
        [18, 19, 20, 21, 22, 23]])
```

`第一行: tensor([0, 1, 2, 3, 4, 5])`

`第一列: tensor([ 0,  6, 12, 18])`

子矩阵:`tensor([[ 8,  9],  [14, 15]])`

- 单变量的求导

```
x = torch.tensor(2.0, requires_grad=True)
y = x ** 2 + 3 * x + 1
y.backward()
print(f"x = {x}, y = {y}")
print(f"dy/dx = {x.grad}")
```

输出结果：`x = 2.0, y = 11.0`,`dy/dx = 7.0`

- **多变量的求导**

`requires_grad`是一个布尔标志，用于控制是否计算该张量的梯度:

`requires_grad=True`: **启用梯度计算**（用于训练）

`requires_grad=False`: **禁用梯度计算**（用于推理/预测）

```
x1 = torch.tensor(1.0, requires_grad=True)
x2 = torch.tensor(2.0, requires_grad=True)
z = x1 ** 2 + x1 * x2 + x2 ** 2
z.backward()
print(f"\n多变量函数: z = x1² + x1*x2 + x2²")
print(f"∂z/∂x1 = {x1.grad}")
print(f"∂z/∂x2 = {x2.grad}")
```

输出结果：`多变量函数: z = x1² + x1*x2 + x2²`,`∂z/∂x1 = 4.0`,`∂z/∂x2 = 5.0`

- 梯度清零

```
x1.grad.zero_()
x2.grad.zero_()
print(f"x1.grad = {x1.grad}")
print(f"x2.grad = {x2.grad}")
```

输出结果：`x1.grad = 0.0`,`x2.grad = 0.0`

- 神经网路的基础

```
import torch.nn as nn
import torch.nn.functional as F

# 定义简单的神经网络
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(10, 5)  # 输入10维，输出5维
        self.fc2 = nn.Linear(5, 2)   # 输入5维，输出2维
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 创建网络实例
model = SimpleNet()
print("网络结构:")
print(model)

#  查看参数
print("\n网络参数:")
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")

# 前向传播
input_data = torch.randn(1, 10)  # batch_size=1, input_size=10
output = model(input_data)
print(f"\n输入形状: {input_data.shape}")
print(f"输出形状: {output.shape}")
print(f"输出: {output}")
```

输出结果：

```
网络结构:
SimpleNet(
  (fc1): Linear(in_features=10, out_features=5, bias=True)
  (fc2): Linear(in_features=5, out_features=2, bias=True)
  (relu): ReLU()
)

网络参数:
fc1.weight: torch.Size([5, 10])
fc1.bias: torch.Size([5])
fc2.weight: torch.Size([2, 5])
fc2.bias: torch.Size([2])

输入形状: torch.Size([1, 10])
输出形状: torch.Size([1, 2])
输出: tensor([[-0.8680, -0.7156]], grad_fn=<AddmmBackward0>)
```

- 损失函数与优化器

```python
# 损失函数
criterion = nn.CrossEntropyLoss()

 优化器
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

 模拟训练步骤
print("模拟训练过程:")

#模拟数据
inputs = torch.randn(4, 10)  # batch_size=4, input_size=10
labels = torch.tensor([0, 1, 0, 1])  # 4个样本的标签

# 前向传播
outputs = model(inputs)
loss = criterion(outputs, labels)

print(f"初始损失: {loss.item():.4f}")

# 反向传播
optimizer.zero_grad()  # 梯度清零
loss.backward()        # 反向传播
optimizer.step()       # 更新参数

# 再次前向传播查看损失变化
outputs_after = model(inputs)
loss_after = criterion(outputs_after, labels)
print(f"一次更新后损失: {loss_after.item():.4f}")
```

输出结果：模拟训练过程:初始损失: 0.6784，一次更新后损失: 0.6751

- 加载数据集

```python
from torch.utils.data import Dataset, DataLoader

print("\n=== 数据集和数据加载器 ===")
```

```
4
5  # 自定义数据集
6  class CustomDataset(Dataset):
7      def __init__(self, data, labels):
8          self.data = data
9          self.labels = labels
10
11     def __len__(self):
12         return len(self.data)
13
14     def __getitem__(self, idx):
15         return self.data[idx], self.labels[idx]
16
17 # 创建模拟数据
18 data = torch.randn(100, 10)  # 100个样本，每个10维
19 labels = torch.randint(0, 2, (100,))  # 100个二分类标签
20
21 dataset = CustomDataset(data, labels)
22 dataloader = DataLoader(dataset, batch_size=8, shuffle=True)
23
24 print(f"数据集大小: {len(dataset)}")
25 print(f"数据加载器批次数: {len(dataloader)}")
26
27 #  遍历数据加载器
28 print("\n遍历前3个批次:")
29 for i, (batch_data, batch_labels) in enumerate(dataloader):
30     print(f"批次 {i+1}: 数据形状 {batch_data.shape}, 标签形状
   {batch_labels.shape}")
31     if i == 2:  # 只显示前3个批次
32         break
```

输出结果:

```
数据集大小: 100
数据加载器批次数: 13

遍历前3个批次:
批次 1: 数据形状 torch.Size([8, 10]), 标签形状 torch.Size([8])
批次 2: 数据形状 torch.Size([8, 10]), 标签形状 torch.Size([8])
批次 3: 数据形状 torch.Size([8, 10]), 标签形状 torch.Size([8])
```

- 模型训练

```
1  print("\n=== 完整训练示例 ===")
2
3  # 准备数据
4  def generate_data(n_samples=1000):
5      """生成简单的二分类数据"""
6      X = torch.randn(n_samples, 2)
7      # 根据到原点的距离创建标签
8      y = (X[:, 0]**2 + X[:, 1]**2 > 1).long()
9      return X, y
10
11 X, y = generate_data(1000)
12
13 # 分割训练集和测试集
14 train_size = int(0.8 * len(X))
```

```
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

print(f"训练集: {X_train.shape}, 测试集: {X_test.shape}")

# 定义模型
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.fc1 = nn.Linear(2, 10)
        self.fc2 = nn.Linear(10, 5)
        self.fc3 = nn.Linear(5, 2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = Classifier()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# 训练循环
def train_model(model, X_train, y_train, epochs=100):
    model.train()
    losses = []

    for epoch in range(epochs):
        # 前向传播
        outputs = model(X_train)
        loss = criterion(outputs, y_train)

        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

        if epoch % 20 == 0:
            print(f'Epoch [{epoch}/{epochs}], Loss: {loss.item():.4f}')

    return losses

print("开始训练...")
losses = train_model(model, X_train, y_train, epochs=100)

#  评估模型
def evaluate_model(model, X_test, y_test):
    model.eval()
    with torch.no_grad():
        outputs = model(X_test)
        _, predicted = torch.max(outputs, 1)
        accuracy = (predicted == y_test).float().mean()
    return accuracy.item()

```

```
accuracy = evaluate_model(model, X_test, y_test)
print(f"测试集准确率: {accuracy:.4f}")
```

输出结果：

```
训练集: torch.Size([800, 2]), 测试集: torch.Size([200, 2])
开始训练...
Epoch [0/100], Loss: 0.7553
Epoch [20/100], Loss: 0.6013
Epoch [40/100], Loss: 0.4866
Epoch [60/100], Loss: 0.2722
Epoch [80/100], Loss: 0.1243
测试集准确率: 0.9850
```

- 模型的保存和加载

```
#  保存模型
torch.save(model.state_dict(), 'model.pth')
print("模型已保存为 'model.pth'")

#  加载模型
new_model = Classifier()
new_model.load_state_dict(torch.load('model.pth'))
new_model.eval()
print("模型加载成功")

# 测试加载的模型
accuracy_loaded = evaluate_model(new_model, X_test, y_test)
print(f"加载模型的测试准确率: {accuracy_loaded:.4f}")
```

输出结果：

```
--- 模型保存和加载 ---
模型已保存为 'model.pth'
模型加载成功
加载模型的测试准确率: 0.9850
```

3.PyTorch 在MNIST和CIFAR上的使用

- PyTorch在MNIST数据集上的使用

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from tensorflow.keras.datasets import mnist
import numpy as np

# 设备配置
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 超参数
batch_size = 64
learning_rate = 0.01
num_epochs = 5

```

```
# 使用 Keras 加载 MNIST 数据
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# 数据预处理（归一化 + 增加通道维度 + 转换为 float32）
x_train = x_train.astype(np.float32) / 255.0  # [60000, 28, 28]
x_test = x_test.astype(np.float32) / 255.0    # [10000, 28, 28]

# 添加通道维度：[N, H, W] → [N, 1, H, W]
x_train = np.expand_dims(x_train, axis=1)  # 变成 [60000, 1, 28, 28]
x_test = np.expand_dims(x_test, axis=1)    # [10000, 1, 28, 28]

y_train = y_train.astype(np.int64)
y_test = y_test.astype(np.int64)

# 转换为 PyTorch Tensor
x_train_tensor = torch.from_numpy(x_train)
y_train_tensor = torch.from_numpy(y_train)
x_test_tensor = torch.from_numpy(x_test)
y_test_tensor = torch.from_numpy(y_test)

# 创建 Dataset 和 DataLoader
train_dataset = TensorDataset(x_train_tensor, y_train_tensor)
test_dataset = TensorDataset(x_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

# 定义简单
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.flatten(x)  # [B, 1, 28, 28] → [B, 784]
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = Net().to(device)

# 损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

# 训练函数
def train():
    model.train()
    for epoch in range(num_epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to(device), target.to(device)
```

```
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            if batch_idx % 100 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Step
[{batch_idx}/{len(train_loader)}], Loss: {loss.item():.4f}')

# 测试函数
def test():
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            pred = output.argmax(dim=1)
            correct += pred.eq(target).sum().item()
            total += target.size(0)
    print(f'Test Accuracy: {100. * correct / total:.2f}%')

# 执行
if __name__ == "__main__":
    train()
    test()
```

输出结果：

```
Epoch [5/5], Step [500/938], Loss: 0.2164
Epoch [5/5], Step [600/938], Loss: 0.3036
Epoch [5/5], Step [700/938], Loss: 0.1889
Epoch [5/5], Step [800/938], Loss: 0.3210
Epoch [5/5], Step [900/938], Loss: 0.3622
Test Accuracy: 91.75%
```

- PyTorch在CIFAR数据集上的使用

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10

#画图时将中文呈现出来
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus']=False

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
print(f"使用设备: {device}")

# 从Keras加载CIFAR-10数据集
print("从Keras加载CIFAR-10数据集...")
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# 数据预处理
# 归一化到0-1范围
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# 将标签从二维展平为一维
y_train = y_train.flatten()
y_test = y_test.flatten()

# 将numpy数组转换为PyTorch张量
# 注意：需要调整维度顺序 (H, W, C) -> (C, H, W)
x_train_tensor = torch.tensor(x_train).permute(0, 3, 1, 2)
x_test_tensor = torch.tensor(x_test).permute(0, 3, 1, 2)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

print(f"训练集形状: {x_train_tensor.shape}")
print(f"测试集形状: {x_test_tensor.shape}")
print(f"训练标签形状: {y_train_tensor.shape}")

# 创建PyTorch数据集
train_dataset = TensorDataset(x_train_tensor, y_train_tensor)
test_dataset = TensorDataset(x_test_tensor, y_test_tensor)

# 创建数据加载器
batch_size = 128
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

# CIFAR-10类别名称
classes = ('airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')

# 显示一些样本图像
def show_sample_images():
    # 获取一个batch的数据
    dataiter = iter(train_loader)
    images, labels = next(dataiter)

    # 显示前8张图像
    fig, axes = plt.subplots(2, 4, figsize=(12, 6))
    for i in range(8):
        ax = axes[i//4, i%4]
        # 调整维度顺序 (C, H, W) -> (H, W, C) 用于显示
        image = images[i].permute(1, 2, 0).numpy()
        ax.imshow(image)
        ax.set_title(classes[labels[i].item()])
        ax.axis('off')
    plt.tight_layout()
    plt.show()
```

```
72
73  show_sample_images()
74
75  # 定义一个简单的CNN模型
76  class SimpleCNN(nn.Module):
77      def __init__(self, num_classes=10):
78          super(SimpleCNN, self).__init__()
79          self.conv1 = nn.Conv2d(3, 32, 3, padding=1)  # 输入通道3，输出通
    道32
80          self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
81          self.pool = nn.MaxPool2d(2, 2)  # 2x2最大池化
82          self.fc1 = nn.Linear(64 * 8 * 8, 512)  # CIFAR-10经过两次池化后是
    8x8
83          self.fc2 = nn.Linear(512, num_classes)
84          self.dropout = nn.Dropout(0.25)
85
86      def forward(self, x):
87          x = self.pool(F.relu(self.conv1(x)))  # 32x32 -> 16x16
88          x = self.pool(F.relu(self.conv2(x)))  # 16x16 -> 8x8
89          x = x.view(-1, 64 * 8 * 8)  # 展平
90          x = F.relu(self.fc1(x))
91          x = self.dropout(x)
92          x = self.fc2(x)
93          return x
94
95  # 创建模型
96  model = SimpleCNN().to(device)
97  print("模型结构:")
98  print(model)
99
100 # 定义损失函数和优化器
101 criterion = nn.CrossEntropyLoss()
102 optimizer = optim.Adam(model.parameters(), lr=0.001)
103
104 # 训练函数
105 def train_epoch(model, train_loader, criterion, optimizer):
106     model.train()
107     running_loss = 0.0
108     correct = 0
109     total = 0
110
111     for batch_idx, (inputs, targets) in enumerate(train_loader):
112         inputs, targets = inputs.to(device), targets.to(device)
113
114         # 前向传播
115         optimizer.zero_grad()
116         outputs = model(inputs)
117         loss = criterion(outputs, targets)
118
119         # 反向传播
120         loss.backward()
121         optimizer.step()
122
123         # 统计
124         running_loss += loss.item()
125         _, predicted = outputs.max(1)
126         total += targets.size(0)
127         correct += predicted.eq(targets).sum().item()
```

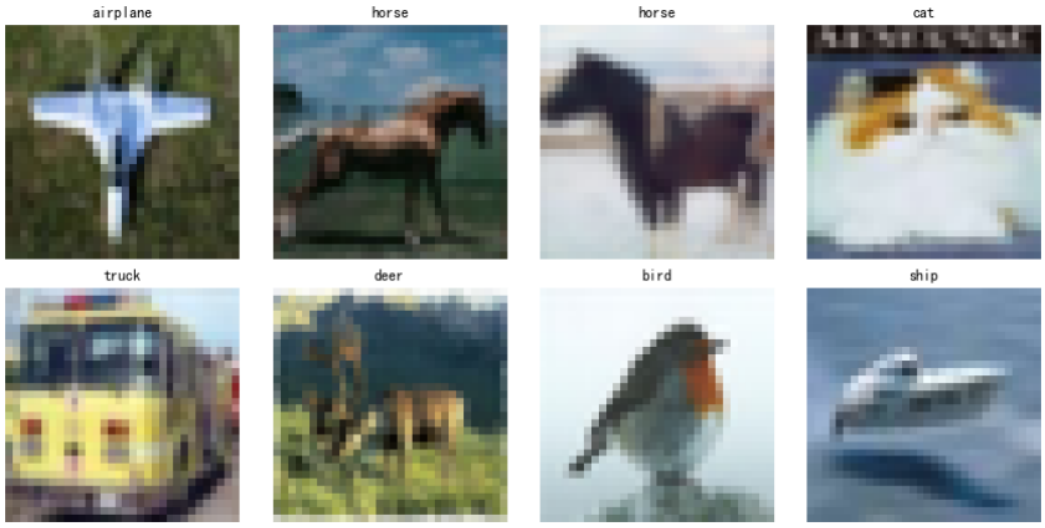
```
128
129         if batch_idx % 100 == 0:
130             print(f'Batch: {batch_idx}/{len(train_loader)}, Loss:
    {loss.item():.3f}')
131
132     epoch_loss = running_loss / len(train_loader)
133     epoch_acc = 100. * correct / total
134     return epoch_loss, epoch_acc
135
136 # 测试函数
137 def test_model(model, test_loader, criterion):
138     model.eval()
139     test_loss = 0.0
140     correct = 0
141     total = 0
142
143     with torch.no_grad():
144         for inputs, targets in test_loader:
145             inputs, targets = inputs.to(device), targets.to(device)
146             outputs = model(inputs)
147             loss = criterion(outputs, targets)
148
149             test_loss += loss.item()
150             _, predicted = outputs.max(1)
151             total += targets.size(0)
152             correct += predicted.eq(targets).sum().item()
153
154     test_loss = test_loss / len(test_loader)
155     test_acc = 100. * correct / total
156     return test_loss, test_acc
157
158 # 开始训练
159 num_epochs = 10
160 print("开始训练...")
161
162 train_losses = []
163 train_accs = []
164 test_losses = []
165 test_accs = []
166
167 for epoch in range(1, num_epochs + 1):
168     print(f'\nEpoch {epoch}/{num_epochs}')
169     print('-' * 40)
170
171     # 训练
172     train_loss, train_acc = train_epoch(model, train_loader,
    criterion, optimizer)
173     train_losses.append(train_loss)
174     train_accs.append(train_acc)
175
176     # 测试
177     test_loss, test_acc = test_model(model, test_loader, criterion)
178     test_losses.append(test_loss)
179     test_accs.append(test_acc)
180
181     print(f'训练损失: {train_loss:.4f}, 训练准确率: {train_acc:.2f}%')
182     print(f'测试损失: {test_loss:.4f}, 测试准确率: {test_acc:.2f}%')
183
```

```
# 绘制训练曲线
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(train_losses, label='训练损失')
plt.plot(test_losses, label='测试损失')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('损失曲线')

plt.subplot(1, 2, 2)
plt.plot(train_accs, label='训练准确率')
plt.plot(test_accs, label='测试准确率')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.title('准确率曲线')

plt.tight_layout()
plt.show()

# 显示一些测试结果
def show_test_results():
    model.eval()
    dataiter = iter(test_loader)
    images, labels = next(dataiter)
    images, labels = images.to(device), labels.to(device)

    with torch.no_grad():
        outputs = model(images)
        _, predicted = outputs.max(1)

    # 显示前12个测试结果
    images = images.cpu()
    fig, axes = plt.subplots(3, 4, figsize=(12, 9))

    for i in range(12):
        ax = axes[i//4, i%4]
        image = images[i].permute(1, 2, 0).numpy()
        ax.imshow(image)

        # 绿色表示正确，红色表示错误
        color = 'green' if predicted[i] == labels[i] else 'red'
        ax.set_title(f'True: {classes[labels[i]]}\nPred:
{classes[predicted[i]]}',
                     color=color, fontsize=10)
        ax.axis('off')

    plt.suptitle('测试结果（绿色:正确，红色:错误）')
    plt.tight_layout()
    plt.show()

show_test_results()

# 计算最终准确率
final_test_loss, final_test_acc = test_model(model, test_loader,
criterion)
```

```
print(f'\n最终测试准确率: {final_test_acc:.2f}%')

# 保存模型
torch.save(model.state_dict(), 'cifar10_simple_model.pth')
print("模型已保存为 'cifar10_simple_model.pth'")
```

输出结果:

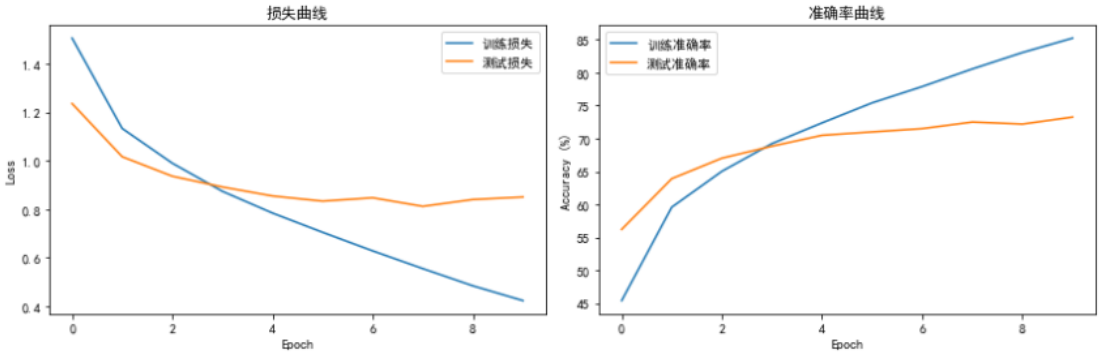
```
使用设备: cpu
从Keras加载CIFAR-10数据集...
训练集形状: torch.Size([50000, 3, 32, 32])
测试集形状: torch.Size([10000, 3, 32, 32])
训练标签形状: torch.Size([50000])
```

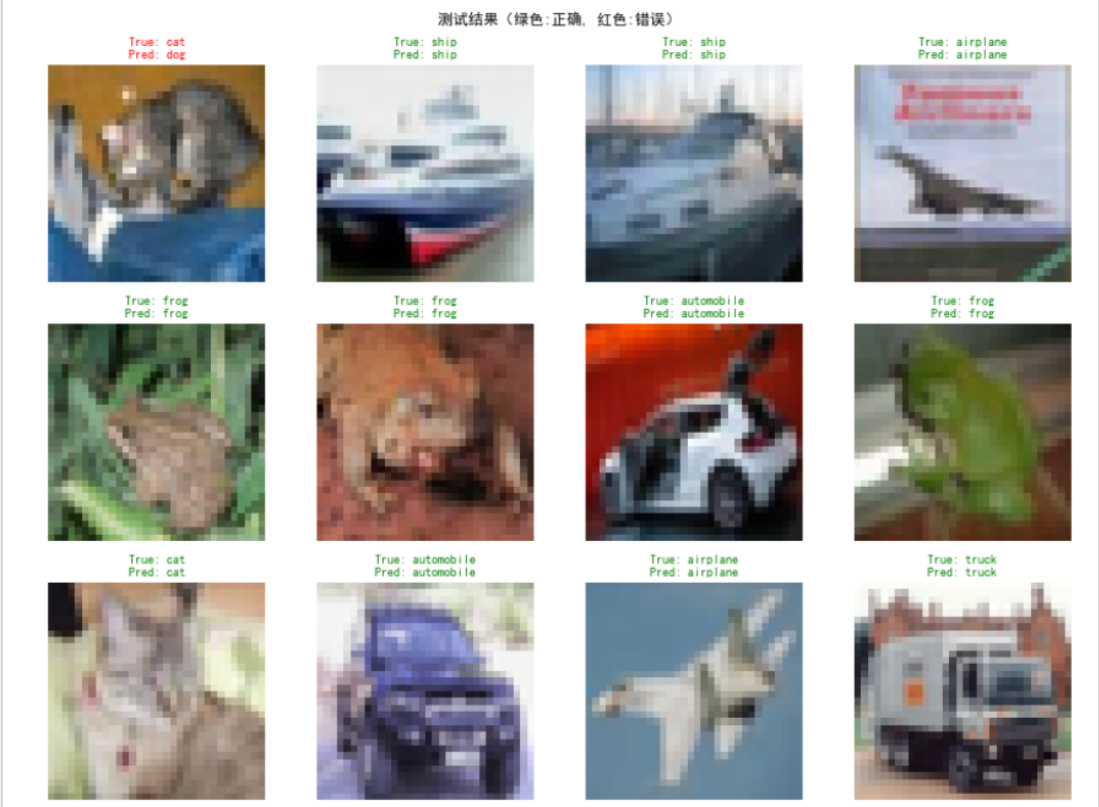



```
模型结构:
SimpleCNN(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
  (fc1): Linear(in_features=4096, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
)
```

训练数据结果:

轮次	训练准确率	训练准确率
1	45.45%	56.24%
2	59.61%	63.93%
3	67.03%	67.03%
4	69.24%	69.24%
5	72.39%	70.50%
6	75.42%	71.01%
7	77.89%	71.51%
8	80.56%	72.51%
9	83.04%	72.20%
10	85.22%	73.26%





最终测试准确率: 73.26%
模型已保存为 'cifar10_simple_model.pth'

Pandas的学习

1.Pandas的介绍

- 简单来说pandas是专门为数据分析而设计的，其提供了两种主要的数据结构Series（一维数组）和DataFrame数据框(2维表格)
- pandas的主要作用为进行数据的读取，对数据进行清洗，数据转换。安装方式为`pip install pandas` or `conda intall pandas`

2.Pandas的实际运用

- pandas的导入

  `import pandas as pd`

- 读取数据

  `data=pd.read_csv('data(2000-4000)_new_label(NoC01_minus1).csv')`

- 数据展示

	text_content	label_code	label_disease	new_label
0	evaluation of women with possible appendicitis...	C06	Digestive System Diseases	1
1	intermittent obstruction of an incarcerated hi...	C06	Digestive System Diseases	1
2	a retrospective analysis of therapy for acute ...	C06	Digestive System Diseases	1
3	rectal examination in general practice objecti...	C06	Digestive System Diseases	1
4	choice of emergency operative procedure for bl...	C06	Digestive System Diseases	1

  `a.shape`

  输出结果:(17996,4)

  `a.dtypes`

  输出结果:

```
text_content     object
label_code       object
label_disease    object
new_label         int64
dtype: object
```

2.1.读取学生成绩数据

- 加载数据

  `students = pd.read_csv("学生成绩.csv")`

  `students.head()`

	姓名	数学	语文	英语	总分
1	青	67	89	98.0	254
8	娜非	87	76	90.0	253
10	何佳	90	67	90.0	247
2	东湖	87	87	67.0	241
5	洪金宝	89	87	65.0	241
9	弹劾	89	95	56.0	240
6	搁金	87	54	98.0	239
3	航发	56	87	89.0	232
0	昂	67	97	67.0	231
11	哈苏	87	68	NaN	231
4	搭建	98	54	78.0	230
12	艾斯比	56	78	96.0	230
7	嘉佳	56	87	67.0	210

`students.info()#数据概况`

```
<class 'pandas.core.frame.DataFrame'>
Index: 13 entries, 1 to 7
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   姓名      13 non-null     object
 1   数学      13 non-null     int64
 2   语文      13 non-null     int64
 3   英语      12 non-null     float64
 4   总分      13 non-null     int64
dtypes: float64(1), int64(3), object(1)
memory usage: 624.0+ bytes
```

`student.describe()#统计摘要`

	数学	语文	英语	总分
count	13.000000	13.000000	12.000000	13.000000
mean	78.153846	78.923077	80.083333	236.846154
std	15.285321	14.320507	15.066268	11.596308
min	56.000000	54.000000	56.000000	210.000000
25%	67.000000	68.000000	67.000000	231.000000
50%	87.000000	87.000000	83.500000	239.000000
75%	89.000000	87.000000	91.500000	241.000000
max	98.000000	97.000000	98.000000	254.000000

2.2.数据预处理

- 查看是否有缺失值

`students.isnull().sum()`

```
姓名    0
数学    0
语文    0
英语    1
总分    0
dtype: int64
```

- 填充缺失值

  `students.fillna(value=69)`

	姓名	数学	语文	英语	总分
1	青	67	89	98.0	254
8	娜非	87	76	90.0	253
10	何佳	90	67	90.0	247
2	东湖	87	87	67.0	241
5	洪金宝	89	87	65.0	241
9	弹劾	89	95	56.0	240
6	搁金	87	54	98.0	239
3	航发	56	87	89.0	232
0	昂	67	97	67.0	231
11	哈苏	87	68	69.0	231
4	搭建	98	54	78.0	230
12	艾斯比	56	78	96.0	230
7	嘉佳	56	87	67.0	210

- 或者删除缺失值

  `students.dropna()`

- 查看是否有重复的值

  `student.duplicated()`

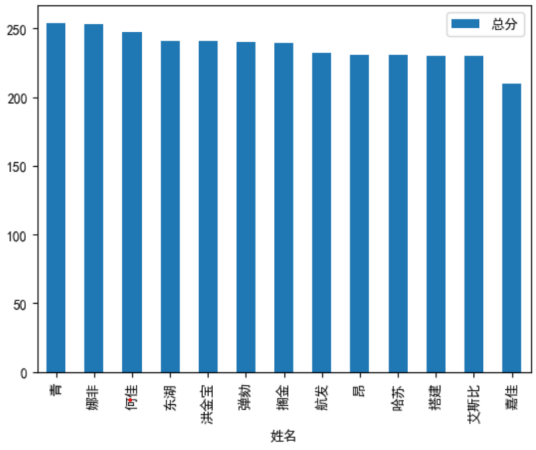
- 该数据没有重复的值如果有就用

  `student.drop_duplicates()`

2.3.数据可视化

- 简单的柱状图

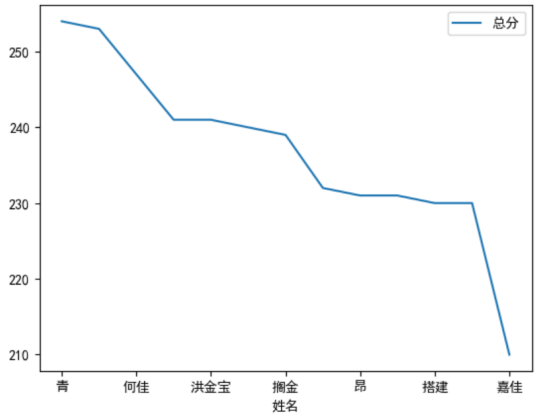
  ```
  import pandas as pd
  import matplotlib.pyplot as plt
  plt.rcParams['font.sans-serif']=['SimHei']
  student.plot(x='姓名',y='总分',kind='bar')
  plt.show()
  ```



- 简单折线图

```
student.plot(x='姓名',y='总分',kind='line')
```

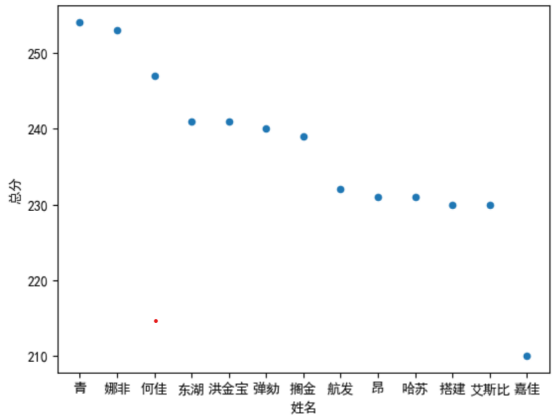
```
plt.show()
```



- 简单散点图

```
student.plot(x='姓名',y='总分',kind='scatter')
```

```
plt.show()
```



matplotlib的常见命令

1.以下表格展示了matplotlib的常用命令

示例	说明
`fig = plt.figure(figsize=None, dpi=None)`	一个没有坐标轴的空图形
`fig.suptitle("画板标题")`	画板添加大标题
`fig, ax = plt.subplots()`	**一个带有单个坐标轴的图形**
`fig, axs = plt.subplots(2, 2)`	一个带有2x2网格坐标轴的图形
`fig, axs = plt.subplots(2, 2,1)`	在2x2的网格坐标中画第一幅图
`axs.imshow(), ax.imshow(train_images[i],cmap='gray')`	将图像数据添加到坐标轴中，但不会立即显示 类似: 像在画布上作画，但还没有展示给观众看
`plt.tight_layout()`	**Matplotlib 中一个非常实用的自动布局调整函数，用于解决子图重叠、标签被截断等问题**
`ax=fig.add_subplot(7,7,i+1)`	第一个7是7行的意思，减小第一个数据输出图像的行距变大，增大第一个数据行距变小 当第一个数据大于第二个数据的时候图像整体会变小
`ax = fig.add_subplot(参数1，参数2，参数3)`	参数1 和参数2是用来对画板划分；参数3指的是 ax 指的是第几部分
`ax.set_title('示例图形')`	这个方法可以设置这条曲线叫什么，或者这个图形的名称
`ax.set_title("数字: {}".format(train_labels[i]))`	对某一幅图给一个“数字i”的标题，其中i是训练集的第i个标签
`ax.set_xlabel('X轴')`	对x轴命名，x轴标签
`ax.set_ylabel('Y轴')`	对y轴命名，y轴标签
`line, = ax.plot(x, y, label='正弦曲线')`	画折线图,label这个变量就是对这条线命名就像训练集和测试集
`ax.legend()`	**添加图例**
`ax.set_xlim(0, 10)`	x轴的范围从0到10
`ax.set_ylim(0, 10)`	y轴从0到10
`ax.set_xticks([0, 5, 10])`	显示x轴的刻度
`ax.set_yticks([-1, 0, 1])`	显示y轴的刻度
`ax.set_xticklabels(['起始', '中间', '结束'])`	给x轴的刻度显示三个标签
`ax.set_yticklabels(['低', '中', '高'])`	给y轴的刻度显示三个标签
`x_axis = ax.xaxis`	获取x轴对象

示例	说明
`y_axis = ay.xaxis`	获取y轴对象
`np.random.seed(19680801)`	设置随机数生成器的种子以确保随机数据可重现
`ax.plot(x, y, color='orange', linewidth=2, marker="^", linestyle="--")` 或者`ax.plot(x, y,"^--r")`	画折线图，color折线的颜色，linewidth线宽，marker="^"点形状，linestyle="--"线形状
`ax.scatter('a', 'b', c='c', s='d', data=data)`	画汽包图，x轴数据来自'a'键，y轴数据来自'b'键，点的颜色由'c'键的值决定，点的大小由'd'键的值决定
`plt.bar(X,Y)`	柱状图
`plt.hist(array)`	画直方图，array是一维数据
`plt.pie(data,lables=,autopct='%1.1f%%')`	饼图
`plt.boxplot(X,Y)`	箱状图
`plt.show()`	**显示图像**

神经网络在MNIST和CIFAR数据集上的实际运用

此次运行的代码是使用jupyter notebook软件运行的

神经网络对MNIST数据集的分类

1. 调用相关的库，以及加载数据

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.callbacks import EarlyStopping
import itertools
import seaborn as sns

#画图时将中文呈现出来
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus']=False
```

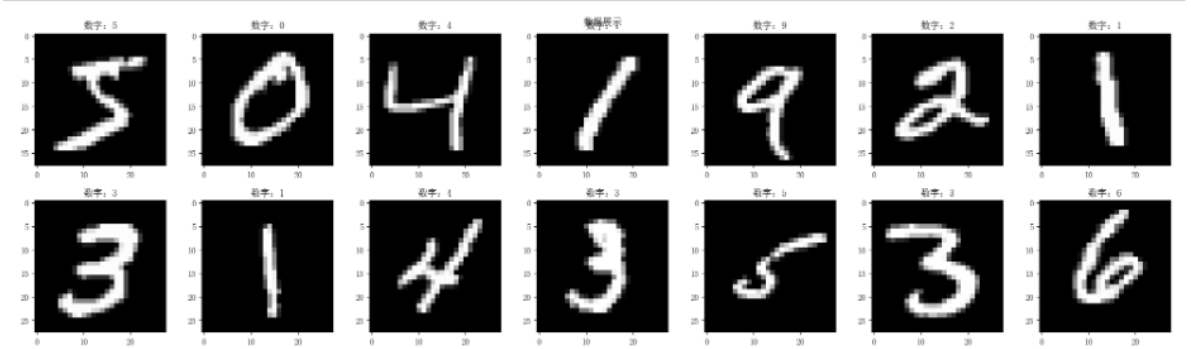
```
#MNIST数据集在keras这个库里面就可以加载出来
mnist=keras.datasets.mnist#读取数据

(train_images,train_labels),(test_images,test_labels)=mnist.load_data()#加载数据

print('train_images',train_images.shape)
print('train_label',train_labels.shape)
print('test_images',test_images.shape)
print('test_label',test_labels.shape)
```

> 输出结果

```
train_images (60000, 28, 28)
train_label (60000,)
test_images (10000, 28, 28)
test_label (10000,)
```

```
fig=plt.figure(figsize=(20,20))
fig.suptitle("数据展示")
for i in range(14):
    ax=fig.add_subplot(7,7,i+1)
    ax.imshow(train_images[i],cmap='gray')
    plt.tight_layout()
    ax.set_title("数字：{}".format(train_labels[i]))
plt.show()
```

输出结果：



2.数据预处理

```
train_x=train_images/255
test_x=test_images/255
print('train_x.shape',train_x.shape)
print("test_x.shape",test_x.shape)
```

输出结果：

```
train_x.shape (60000, 28, 28)
```

```
test_x.shape (10000, 28, 28)
```

```
#修改图片的形状
train_x1=tf.reshape(train_x,
[train_x.shape[0],train_x.shape[1]*train_x.shape[2]])
test_x1=tf.reshape(test_x,[test_x.shape[0],test_x.shape[1]*test_x.shape[2]])
print('train_x1.shape',train_x1.shape)
print('test_x1.shape',test_x1.shape)
```

输出结果：

```
train_x1.shape (60000, 784)
```

```
test_x1.shape (10000, 784)
```

```
#标签的数据预处理
train_Y=tf.one_hot(train_labels,depth=10)
test_Y=tf.one_hot(test_labels,depth=10)
print(train_labels[0],train_Y[0].numpy())
train_Y
```

输出结果：

```
5 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

3.模型的训练与评估

```
#网络的搭建
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Conv2D,MaxPooling2D,Flatten

model = keras.Sequential(
    [keras.layers.Dense(units=256,input_shape=(784,),activation='relu'),
     #units=256输入层神经元个数
     keras.layers.Dense(units=128,activation='relu'),
     keras.layers.Dropout(0.5),#改进
     keras.layers.BatchNormalization(),#改进
     keras.layers.Dense(units=10,activation='softmax')]
)
model.summary()
```

输出结果：

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_9 (Dense)             (None, 256)               200960

 dense_10 (Dense)            (None, 128)               32896

 dropout (Dropout)           (None, 128)               0

 batch_normalization (Batch  (None, 128)               512
 Normalization)

 dense_11 (Dense)            (None, 10)                1290

=================================================================
Total params: 235658 (920.54 KB)
Trainable params: 235402 (919.54 KB)
Non-trainable params: 256 (1.00 KB)
_________________________________________________________________
```

```
#编译模型与训练网路
from tensorflow.keras.callbacks import EarlyStopping

model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
stop = EarlyStopping(monitor="val_accuracy",min_delta=0.00001,patience = 3)
history=model.fit(train_x1,train_Y,epochs=20,verbose=2,validation_split=0.2,callbacks=[stop])
```

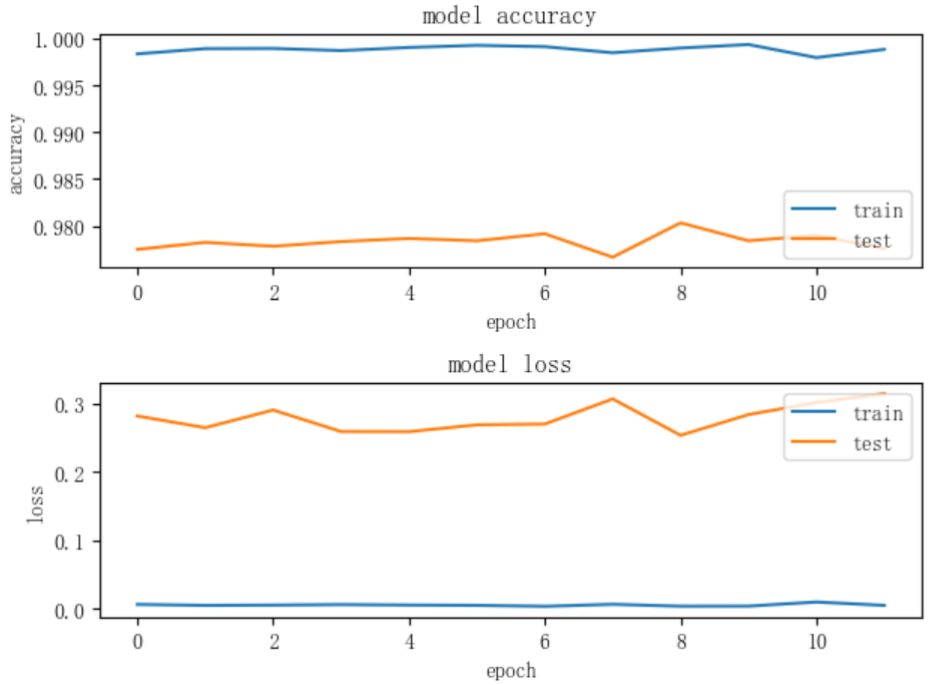
输出结果:

```
1500/1500 - 3s - loss: 0.0344 - accuracy: 0.9891 - val_loss: 0.0864 - val_accuracy: 0.9787 - 3s/epoch - 2ms/step
Epoch 12/20
1500/1500 - 3s - loss: 0.0317 - accuracy: 0.9900 - val_loss: 0.0831 - val_accuracy: 0.9797 - 3s/epoch - 2ms/step
Epoch 13/20
1500/1500 - 3s - loss: 0.0273 - accuracy: 0.9918 - val_loss: 0.0891 - val_accuracy: 0.9773 - 3s/epoch - 2ms/step
Epoch 14/20
1500/1500 - 3s - loss: 0.0249 - accuracy: 0.9919 - val_loss: 0.0934 - val_accuracy: 0.9778 - 3s/epoch - 2ms/step
Epoch 15/20
1500/1500 - 3s - loss: 0.0224 - accuracy: 0.9930 - val_loss: 0.0970 - val_accuracy: 0.9792 - 3s/epoch - 2ms/step
```

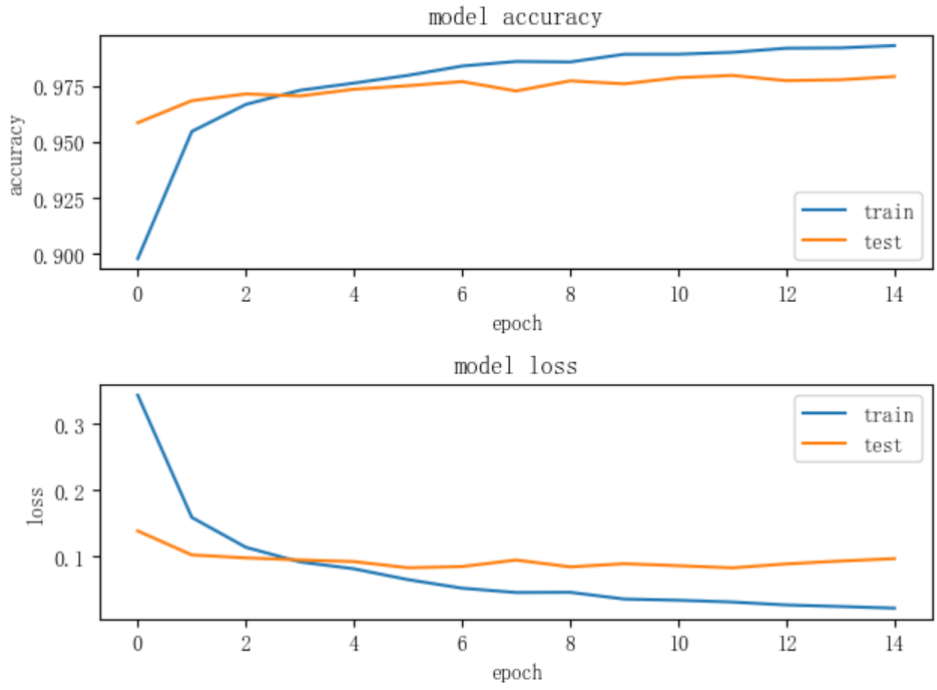
4.结果可视化

```
def drow_model(training):
    plt.figure()
    plt.subplot(2,1,1)
    plt.plot(training.history['accuracy'])
    plt.plot(training.history['val_accuracy'])
    plt.title("model accuracy")
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train','test'],loc='lower right')
    plt.subplot(2,1,2)
    plt.plot(training.history['loss'])
    plt.plot(training.history['val_loss'])
    plt.title("model loss")
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train','test'],loc='upper right')
    plt.tight_layout()
    plt.show()
drow_model(history)
```

输出结果:



结果显示模型过拟合，所以加入Dropout层和批次归一化之后的输出结果为



神经网络在CIFAR数据上的分类

1.库的调用与数据的加载

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.callbacks import EarlyStopping
```

```
import itertools
import seaborn as sns

plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus']=False

cifar10=keras.datasets.cifar10#读取数据
(train_images,train_labels),(test_images,test_labels)=cifar10.load_data()#加载数据

print('train_images',train_images.shape)
print('train_label',train_labels.shape)
print('test_images',test_images.shape)
print('test_label',test_labels.shape)
```

输出结果：

```
train_images (50000, 32, 32, 3)
```

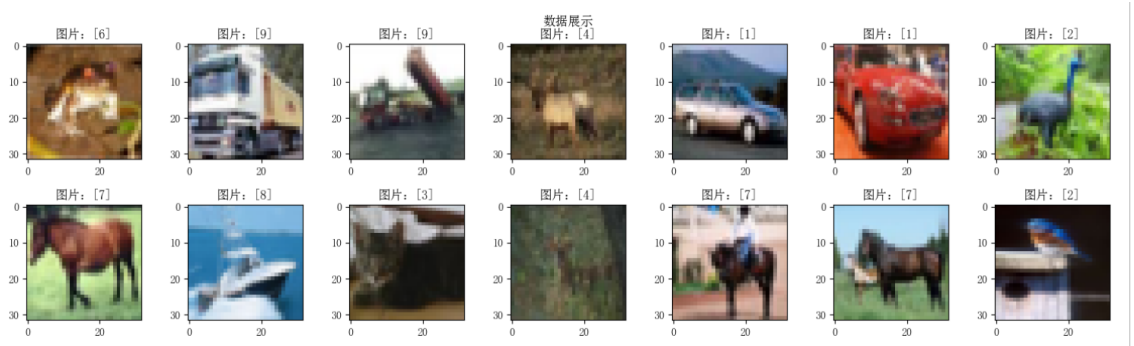
```
train_label (50000, 1)
```

```
test_images (10000, 32, 32, 3)
```

```
test_label (10000, 1)
```

```
#对数据图像进行可视化
fig=plt.figure(figsize=(20,20))
fig.suptitle("数据展示")
for i in range(14):
    ax=fig.add_subplot(7,7,i+1)
    fig.set_figheight(15)
    fig.set_figwidth(15)
    ax.imshow(train_images[i],)
    plt.tight_layout()
    ax.set_title("图片：{}".format(train_labels[i]))
plt.show()
```

输出结果：



2.数据的预处理

```
#对图像进行预处理
train_x=train_images/255
test_x=test_images/255
print('train_x.shape',train_x.shape)
print("test_x.shape",test_x.shape)
```

输出结果：

```
train_x.shape (50000, 32, 32, 3)
```

```
test_x.shape (10000, 32, 32, 3)
```

```
#修改数据的形状
train_x1=tf.reshape(train_x,
[train_x.shape[0],train_x.shape[1]*train_x.shape[2]*train_x.shape[3]])
test_x1=tf.reshape(test_x,
[test_x.shape[0],test_x.shape[1]*test_x.shape[2]*test_x.shape[3]])
print('train_x1.shape',train_x1.shape)
print('test_x1.shape',test_x1.shape)
```

输出结果：

```
train_x1.shape (50000, 3072)
```

```
test_x1.shape (10000, 3072)
```

```
#对数据标签进行数据预处理
train_Y=tf.one_hot(tf.squeeze(train_labels),depth=10)
test_Y=tf.one_hot(tf.squeeze(test_labels),depth=10)
print(train_labels[0],train_Y[0].numpy())
train_Y
```

输出结果：

```
[6] [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

3.模型的训练与评估

```
#使用最基本的网络去训练数据
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Conv2D,MaxPooling2D,Flatten,
Dropout, BatchNormalization

model = keras.Sequential(
    [Dense(units=256,input_shape=(3072,),activation='relu'),#units=256输入层
神经元个数
     Dense(units=128,activation='relu'),
     Dense(units=10,activation='softmax')]
)
model.summary()
```

输出结果：

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_26 (Dense)            (None, 256)               786688

 dense_27 (Dense)            (None, 128)               32896

 dense_28 (Dense)            (None, 10)                1290

=================================================================
Total params: 820874 (3.13 MB)
Trainable params: 820874 (3.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
#编译模型和训练网路
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
from tensorflow.keras.callbacks import EarlyStopping
stop = EarlyStopping(monitor="val_accuracy",min_delta=0.00001,patience = 3)
history=model.fit(train_x1,train_Y,epochs=5,verbose=2,validation_split=0.2,callbacks=[stop])
```

输出结果：

```
Epoch 1/5
1250/1250 - 9s - loss: 1.8809 - accuracy: 0.3234 - val_loss: 1.7483 - val_accuracy: 0.3752 - 9s/epoch - 7ms/step
Epoch 2/5
1250/1250 - 9s - loss: 1.6962 - accuracy: 0.3903 - val_loss: 1.6607 - val_accuracy: 0.4038 - 9s/epoch - 7ms/step
Epoch 3/5
1250/1250 - 9s - loss: 1.6203 - accuracy: 0.4188 - val_loss: 1.6701 - val_accuracy: 0.4062 - 9s/epoch - 7ms/step
Epoch 4/5
1250/1250 - 9s - loss: 1.5712 - accuracy: 0.4369 - val_loss: 1.5876 - val_accuracy: 0.4322 - 9s/epoch - 7ms/step
Epoch 5/5
1250/1250 - 9s - loss: 1.5337 - accuracy: 0.4508 - val_loss: 1.6434 - val_accuracy: 0.4176 - 9s/epoch - 7ms/step
```

由于普通的网络结构无法训练数据，所以就采用CNN网络来训练数据

```
#卷积神经网络的搭建与训练
from tensorflow.keras.layers import Conv2D,MaxPooling2D,BatchNormalization,Dropout,Dense,GlobalAveragePooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow import keras
from tensorflow.keras.models import Sequential
def create_enhanced_model(input_shape=(32,32,3)):
    model = Sequential([
        Conv2D(64,(3,3),activation='relu',padding='same',input_shape=input_shape),
        BatchNormalization(),
        Conv2D(64,(3,3),activation='relu',padding='same'),
        MaxPooling2D((2,2)),
        Dropout(0.25),

        Conv2D(128,(3,3),activation='relu',padding='same'),
        BatchNormalization(),
        Conv2D(128,(3,3),activation='relu',padding='same'),
        MaxPooling2D((2,2)),
        Dropout(0.25),

```

```
        Conv2D(128,(3,3),activation='relu',padding='same'),
        BatchNormalization(),
        Conv2D(128,(3,3),activation='relu',padding='same'),
        MaxPooling2D((2,2)),
        Dropout(0.25),

        GlobalAveragePooling2D(),
        Dense(256,activation='relu'),
        Dropout(0.5),
        Dense(10,activation='softmax')
    ])
    return model
# 数据增强生成器
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    zoom_range=0.1,
    fill_mode='nearest'
)
# 创建模型
model1 = create_enhanced_model(input_shape=(32, 32, 3))
model1.compile(optimizer='adam',loss='categorical_crossentropy',metrics=
['accuracy'])
model1.summary()
stop = EarlyStopping(monitor="val_accuracy",min_delta=0.001,patience = 3)
history=model1.fit(train_x,train_Y,batch_size=128,epochs=100,verbose=2,valid
ation_split=0.2,callbacks=[stop])
```

输出结果：

```
1250/1250 - 258s - loss: 0.3138 - accuracy: 0.8921 - val_loss: 0.5499 - val_accuracy: 0.8253 - 258s/epoch - 206ms/step
Epoch 16/100
1250/1250 - 250s - loss: 0.3066 - accuracy: 0.8947 - val_loss: 0.5455 - val_accuracy: 0.8411 - 250s/epoch - 200ms/step
Epoch 17/100
1250/1250 - 252s - loss: 0.2937 - accuracy: 0.8997 - val_loss: 0.5323 - val_accuracy: 0.8396 - 252s/epoch - 202ms/step
Epoch 18/100
1250/1250 - 249s - loss: 0.2739 - accuracy: 0.9057 - val_loss: 0.5620 - val_accuracy: 0.8355 - 249s/epoch - 199ms/step
Epoch 19/100
1250/1250 - 250s - loss: 0.2627 - accuracy: 0.9091 - val_loss: 0.6401 - val_accuracy: 0.8228 - 250s/epoch - 200ms/step
```