陕西科技大学
SHAANXI UNIVERSITY OF SCIENCE & TECHNOLOGY

专业综合设计 I 课程设计 说明书

题目： 基于卷积神经网络的图像识别算法与实现

学生姓名： 马凌峰

学　　号： 202307020122

院 （系）： 电子信息与人工智能学院

专　　业： 计算机科学与技术

指导教师： 陈海丰

2025 年 6 月 30 日

摘要

随着人工智能技术的飞速发展，卷积神经网络（Convolutional Neural Network, CNN）已成为计算机视觉领域的核心技术，尤其在图像识别任务中展现出卓越的性能 。本次专业综合设计围绕“基于卷积神经网络的图像识别算法与实现”这一主题 ，旨在通过完整的项目实践，加深对 CNN 基本原理的理解，并提升工程实现与问题解决能力。

本报告详细阐述了从理论到实践的全过程。首先，介绍了卷积神经网络的基本构成，包括卷积层、池化层、激活函数、全连接层等核心模块的原理 。其次，描述了利用 PyTorch 深度学习框架进行模型设计、数据集加载、模型训练、验证与参数保存的完整流程 。随后，报告重点说明了如何脱离深度学习框架，仅使用 NumPy 库实现 CNN 模型的推理过程，通过复现底层计算逻辑，深化了对算法内部机制的理解 。最后，报告展示了将该模型推理逻辑从 Python 迁移至 C 语言环境的技术过程，重点解决了数据结构定义、跨语言参数加载和核心运算实现等问题，并实现了与 PyTorch 端误差小于10^{-5}的精确推理 。

通过本次设计，不仅成功实现了一个可用的图像识别模型，更在算法理解、编程实践、代码调试和跨语言开发方面得到了全面的锻炼 。

关键词： 卷积神经网络；图像识别；PyTorch；NumPy；C 语言实现

1. 卷积神经网络的基本原理

卷积神经网络（CNN）是一种特殊设计用于处理具有类似网格结构数据（如图像）的深度学习模型 。它通过一系列专门的层来模拟人类视觉皮层的处理方式，从而高效地提取和利用图像中的空间层级特征。一个典型的 CNN 模型主要由以下几种核心组件构成：

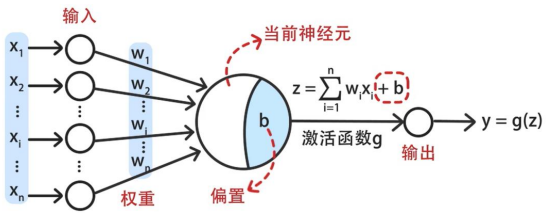


图 1 神经网络原理

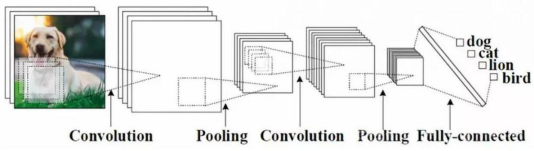


图 2 CNN 整体架构

1.1 卷积层 (Convolutional Layer)

卷积层是 CNN 的核心，其主要功能是提取输入数据的局部特征。它通过一个或多个称为“卷积核”（Kernel）或“滤波器”（Filter）的小型权重矩阵，在输入数据上进行滑动窗口式的点积运算。

- **工作原理**：卷积核在输入图像（或前一层的特征图）上从左到右、从上到下滑动。在每个位置，卷积核与其覆盖的图像区域进行逐元素相乘后

求和，再加上一个偏置项（Bias），从而生成输出特征图（Feature Map）中的一个像素。

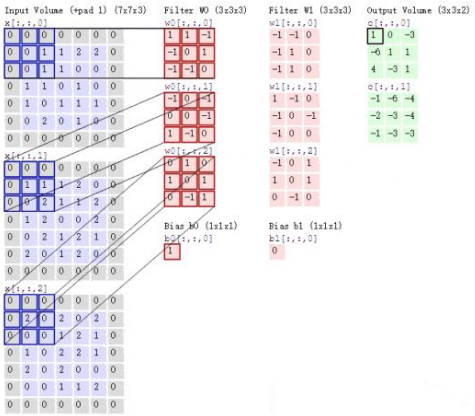


图 3 卷积操作示意图

- **关键参数：**
    - **卷积核大小 (Kernel Size)：**定义了感受野的大小，即一次运算能看到的区域范围。例如，一个 3 × 3 的卷积核。
    - **步长 (Stride)：**定义了卷积核每次滑动的距离。步长为 1 表示逐像素滑动，步长为 2 则表示每次跳过一个像素。
    - **填充 (Padding)：**在输入图像的边缘周围添加额外的像素（通常是 0），以控制输出特征图的空间尺寸。这有助于保留边缘信息，并使得输入和输出尺寸更容易匹配。
- **特征提取：**不同的卷积核可以学习到不同的图像特征，例如边缘、角点、纹理等。网络通过训练自动学习这些权重。

1.2 激活函数 (Activation Function)

激活函数为神经网络引入非线性因素，使其能够学习和表示更加复杂的函数关系。如果没有激活函数，多层神经网络本质上等同于一个单层的线性模型。

- **ReLU (Rectified Linear Unit)：**在本次设计中主要使用的激活函数是 ReLU。其数学表达式为 $f(x) = \max(0, x)$。

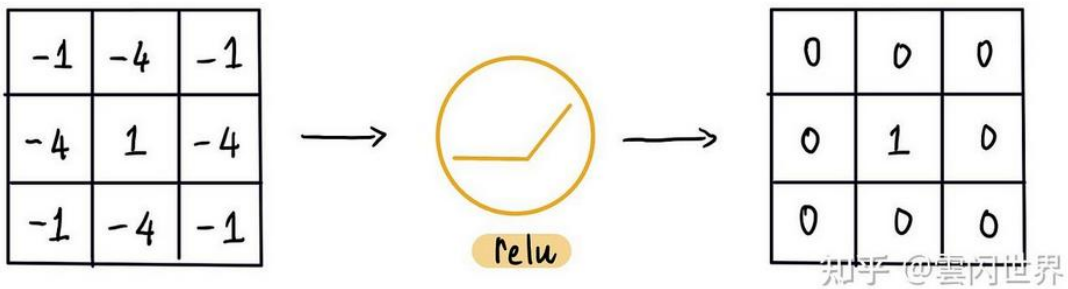


图 4 激活层原理示意图

- **优点**：计算简单、收敛速度快，并且在一定程度上缓解了梯度消失问题。

1.3 池化层 (Pooling Layer)

池化层（也称下采样层）通常紧跟在卷积层之后，其主要作用是降低特征图的空间维度（宽度和高度），从而减少网络中的参数数量和计算量，同时也能增强模型的鲁棒性（对微小位移不敏感）并防止过拟合。

- **最大池化 (Max Pooling)**：这是最常用的池化方法。它将特征图划分为若干个不重叠的矩形区域，并从每个区域中提取最大值作为输出。
- **工作原理**：与卷积类似，一个固定大小的窗口在特征图上滑动，但它不进行加权求和，而是取窗口内的最大值。

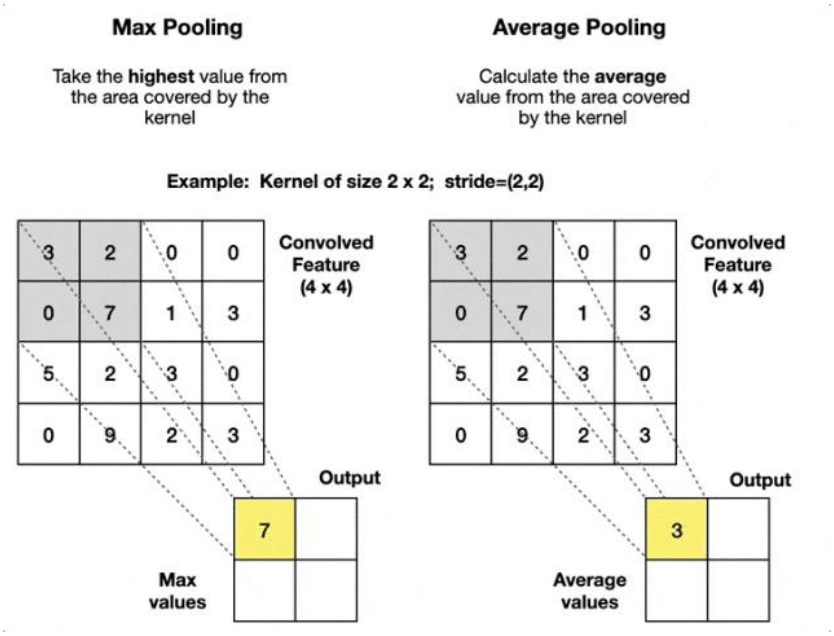


图 5 池化层原理示意图

1.4 批归一化层 (Batch Normalization Layer)

批归一化层是一种用于加速训练和提升模型稳定性的技术。它对每个批次（Batch）的数据在网络中每一层的输入进行归一化处理。

- **工作原理**：对于每个批次的输入，它会计算该批次数据的均值和方差，然后将输入数据标准化为均值为 0、方差为 1 的分布。接着，通过两个可学习的参数（缩放因子 γ 和平移因子 β）对标准化后的数据进行缩放和偏移，以保留网络的表达能力。
- **优点**：允许使用更高的学习率，加速模型收敛，并具有一定的正则化效果。

1.5 全连接层 (Fully Connected Layer)

在经过多个卷积和池化层提取了丰富的层次化特征后，全连接层负责将这些特征进行整合，并最终映射到样本的类别空间，用于分类任务。

- **工作原理**：全连接层的每个神经元都与前一层的所有神经元相连接。在进入全连接层之前，通常需要将多维的特征图“展平”（Flatten）成一个一维向量。

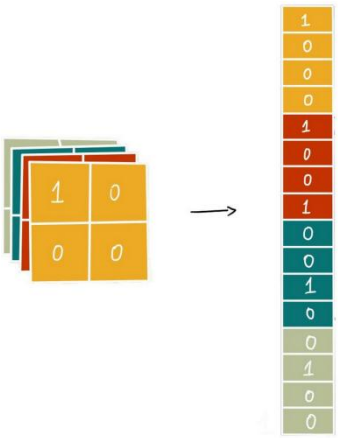
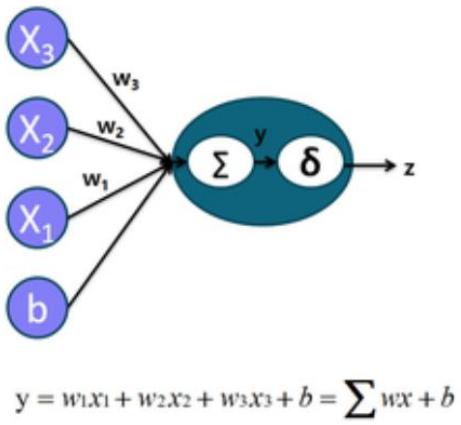


图 6 全连接层原理示意图



$$y = w_1x_1 + w_2x_2 + w_3x_3 + b = \sum wx + b$$

图 7 全连接运算原理示意图

- **功能**：它相当于一个传统的多层感知机，对提取到的特征进行高级别的推理和分类。

1.6 Dropout 层

Dropout 是一种在训练过程中使用的正则化技术，用于防止模型过拟合。

- **工作原理**：在训练的每次迭代中，Dropout 层会以一定的概率 p 随机地“丢弃”（即暂时使其输出为 0）一部分神经元。这样可以防止神经元之间产生复杂的共适应关系，迫使网络学习到更加鲁棒的特征。在测试阶段，所有神经元都会被使用，但其输出会按比例缩小以作补偿。

2. PyTorch 训练流程

本次设计首先利用 PyTorch 框架完成了 CNN 模型的搭建和训练 。整个流程遵循了标准的深度学习项目实践，涵盖了数据准备、模型构建、训练循环和验证评估等关键环节。

2.1 数据准备与加载

- **数据集**：实验使用的数据集包含多个情感类别的图像，存储在`./data/images`目录下，其对应的标签信息记录在`./data/labels.txt`文件中 。

- **`Dataset`类**：为了高效加载数据，我们定义了`CustomImageDataset`类，它继承自`torch.utils.data.Dataset`。该类负责读取图像路径和标签，

并在__getitem__方法中加载图像、应用预处理变换（如转换为 Tensor、归一化等）。

- **DataLoader**：使用 torch.utils.data.DataLoader 来创建数据加载器，它能够自动处理数据的批处理（batching）、打乱（shuffling）和并行加载，极大地提升了数据供给效率。

2.2 模型架构设计

本次实验采用的 CNN 模型（定义于 model.py 中的 LCNN 类）是一个轻量级的网络结构，具体由三个卷积块和三个全连接层组成。

- **模型结构**：
    1. **卷积块 1**: Conv2d (3→16) → BatchNorm2d → ReLU → Conv2d (16→16) → BatchNorm2d → ReLU → MaxPool2d → Dropout。
    2. **卷积块 2**: Conv2d (16→32) → BatchNorm2d → ReLU → Conv2d (32→32) → BatchNorm2d → ReLU → MaxPool2d → Dropout。
    3. **卷积块 3**: Conv2d (32→64) → BatchNorm2d → ReLU → Conv2d (64→64) → BatchNorm2d → ReLU → MaxPool2d → Dropout。
    4. **展平 (Flatten)**: 将最后一个池化层输出的特征图展平为一维向量。
    5. **全连接层**: Linear (1600→128) → ReLU → Dropout → Linear (128→64) → ReLU → Dropout → Linear (64→7)。

以下是 model.py 中定义的 LCNN 模型的核心代码：

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

class LCNN(nn.Module):
    def __init__(self, opts):
        super(LCNN, self).__init__()
        # 第一个卷积块
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 16, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(16)
        self.pool1 = nn.MaxPool2d(2, 2)
        # ... 其他卷积块和全连接层定义 ...
        self.fc3 = nn.Linear(64, opts.num_classes)

    def forward(self, x):
        # 前向传播逻辑
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool1(x)
        # ... 后续传播 ...
        x = x.view(x.size(0), -1) # 展平
```

```
        # ... 全连接层 ...
        x = self.fc3(x)
        return x
```

2.3 训练与验证

训练过程在 train.py 脚本中实现。

- **损失函数**：由于是多分类任务，选用了交叉熵损失函数 `nn.CrossEntropyLoss()`。
- **优化器**：选用 `Adam` 优化器，它是一种自适应学习率的优化算法，收敛速度快且表现稳定。
- **训练循环**：
    1. 遍历设定的 `num_epochs`。
    2. 在每个 epoch 中，将模型设置为训练模式 (`model.train()`)。
    3. 从 `train_loader` 中分批次获取图像和标签。
    4. 将数据迁移到指定设备（CPU 或 GPU）。
    5. 执行前向传播，计算模型输出。
    6. 计算损失。
    7. 执行反向传播 (`loss.backward()`) 和优化器更新 (`optimizer.step()`)。
- **验证与模型保存**：
    1. 每个 epoch 训练结束后，将模型设置为评估模式 (`model.eval()`)。
    2. 在验证集 `val_loader` 上进行推理，计算模型的分类准确率。
    3. 如果当前准确率高于历史最佳准确率，则保存当前模型的状态字典（权重参数）到 `best.ckpt` 文件中，以便后续使用 。

以下是 train.py 中训练循环的核心逻辑：

```
# train.py
# ... 省略了设置和加载部分 ...
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=opts.learning_rate)

for epoch in range(opts.num_epochs):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
```

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

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

        # 验证模型
        model.eval()
        with torch.no_grad():
            # ... 计算验证集准确率 ...
            acc = 100 * correct / total
            if acc > best_acc:
                best_acc = acc
                torch.save(checkpoint, os.path.join(opts.save_dir, 'best.
ckpt'))
```

完成训练后，使用提供的 save_parameters.py 脚本，将 best.ckpt 中的权重和偏置等可学习参数导出为二进制（.bin）文件，存放在 params/目录下，供后续的 NumPy 和 C 语言实现使用 。

3. NumPy 实现卷积神经网络的技术过程与关键代码说明

为了深入理解 CNN 的底层运算机制，本次设计要求使用 NumPy 库手动实现整个模型的推理过程 。这要求我们脱离 PyTorch 的自动微分和高度封装的层，从零开始构建每个模块的计算逻辑。此部分实现主要集中在 valid_python_many.py 和 valid_python_single.py 中。

3.1 技术过程

1. **参数加载**：首先，实现一个 `read_bin_as_floats` 函数，用于从`.bin` 文件中读取由 PyTorch 保存的模型参数（如 `conv1_weight.bin`, `conv1_bias.bin` 等）。读取后，根据模型结构将一维数组重塑（reshape）为正确的权重维度（例如，卷积核的形状为 `[OC, IC, KH, KW]`）。

2. **实现核心运算函数**：为 CNN 的每一个基本操作编写一个独立的 NumPy 函数。

    - **卷积 (`func_conv2d`)**：实现四重嵌套循环，模拟卷积核在输入特征图上的滑动计算。外层循环遍历输出通道，内层循环遍历输出特征图的高度和宽度，并计算输入窗口与相应卷积核的点积。

    - **批归一化 (`func_batch_norm`)**：根据批归一化的推理公式 $y = \gamma \frac{x-E[x]}{\sqrt{Var[x]+\epsilon}} + \beta$ 进行计算，其中 $E[x]$ 和 $Var[x]$ 是训练时得到的全局移动平均和移动方差。

    - **激活函数 (`func_relu`)**：利用 `np.maximum(0, input)`实现。

- **最大池化 (func_max_pooling)**：实现三重循环，在输入特征图的每个通道上，使用一个滑动窗口，并用 np.max()找到窗口内的最大值。
- **全连接 (func_fc)**：将展平后的输入向量与全连接层的权重矩阵进行矩阵乘法（np.dot），然后加上偏置。

3. **构建推理流程**：编写 complex_model_inference 函数，按照 LCNN 模型中定义的前向传播顺序，依次调用上述实现的各个运算函数，将输入图像逐层处理，最终得到分类的 logits 输出。
4. **精度验证**：将同一张输入图像分别送入 PyTorch 模型和我们用 NumPy 实现的模型，计算两者输出 logits 之间的平均绝对误差。根据任务要求，该误差小于10^{-5}，验证了 NumPy 实现的正确性 。

```
正在加载参数...
参数加载完成
PyTorch Logits: [[-27.659693 -10.738062  -8.621808  14.670205 -12.741111 -15.725271
  -15.983282]]
Numpy Logits: [[-27.65969583 -10.73806228  -8.62180856  14.67020568 -12.74111263
  -15.72527306 -15.98328162]]
PyTorch Prediction: happiness
Numpy Prediction: happiness
Mean Error: 0.0000012356
finish...
```

图 7 代码验证结果

3.2 关键代码说明

卷积操作 func_conv2d 的 NumPy 实现：

```python
# valid_python_many.py
def func_conv2d(input, kernel, bias, stride=1, padding=0):
    """卷积操作"""
    if padding > 0:
        input_padded = np_pad(input, padding) # 对输入进行填充
    else:
        input_padded = input

    C, H, W = input_padded.shape
    OC, _, KH, KW = kernel.shape

    # 计算输出尺寸
    OH = (H - KH) // stride + 1
    OW = (W - KW) // stride + 1
    output = np.zeros((OC, OH, OW))

    # 执行卷积运算
    for oc in range(OC): # 遍历每个输出通道
        for y in range(0, OH): # 遍历输出高度
            for x in range(0, OW): # 遍历输出宽度
                h_start, w_start = y * stride, x * stride
```

```
                h_end, w_end = h_start + KH, w_start + KW

                window = input_padded[:, h_start:h_end, w_start:w_en
d]
                # 卷积计算：对应元素相乘后求和
                conv_val = np.sum(window * kernel[oc])
                output[oc, y, x] = conv_val + bias[oc]
    return output
```

该函数清晰地展示了卷积的本质——加权求和。通过手动实现，可以直观地理解步长、填充等参数如何影响计算过程和输出尺寸。

完整推理流程 complex_model_inference：

```
# valid_python_many.py
def complex_model_inference(input_tensor, params):
    """复杂模型推理"""
    x = input_tensor.squeeze().numpy().astype(np.float32)

    # 第一个卷积块
    x = func_conv2d(x, params['conv1_weight'], params['conv1_bias'],
 stride=1, padding=1)
    x = func_batch_norm(x, params['bn1_weight'], params['bn1_bias'],

                        params['bn1_running_mean'], params['bn1_runni
ng_var'])
    x = func_relu(x)
    # ... 省略中间各层的调用 ...
    x = func_max_pooling(x, 2, 2)  # pool3

    # 展平
    x = x.reshape(1, -1)

    # 全连接层
    x = func_fc(x, params['fc1_weight'], params['fc1_bias'])
    x = func_relu(x)
    x = func_fc(x, params['fc2_weight'], params['fc2_bias'])
    x = func_relu(x)
    x = func_fc(x, params['fc3_weight'], params['fc3_bias'])

    return x
```

此函数是整个 NumPy 实现的核心，它将各个独立的运算模块串联起来，完整地复现了 PyTorch 模型的推理逻辑。

4. C 语言环境下实现卷积神经网络的技术过程与关键代码说明

为了追求更高的运行效率和更好的平台可移植性，课程设计要求将模型推理过程进一步用 C 语言实现 。这一步是理论知识向底层工程实践转化的关键环节，挑战在于手动管理内存和实现复杂的数值计算。实现代码主要在 example.cpp 中。

4.1 技术过程

1. **定义数据结构**：C 语言是强类型语言，首先需要为图像、向量和网络层定义清晰的结构体（`struct`）。例如，`Image` 结构体包含宽、高、通道数和指向浮点数数组的指针；`ConvolutionLayer` 结构体包含输入/输出通道、核大小、步长、填充以及指向权重和偏置的指针。

2. **图像与参数读取**：

    - **图像读取**：课程设计要求不能修改 `readBMP` 函数。该函数负责读取一个 24 位的 BMP 图像文件，将其像素数据解码、翻转（BMP 通常是倒置存储的），并进行与 PyTorch `transforms` 一致的归一化处理（`(x/255.0 - 0.5) / 0.5`），最终将 R, G, B 三个通道的数据分别存放在一块连续的内存中。

    - **参数加载**：实现 `readFloatBinary` 函数，用于从`.bin` 文件中读取浮点数参数并加载到内存中，供相应的网络层结构体使用。

3. **实现核心运算函数**：用 C 语言实现与 NumPy 版本功能相同的核心运算。

    - **填充 (pad)**：创建一个新的、更大的 `Image`，并将原始图像数据复制到其中央，四周用 0 填充。

    - **卷积 (convolve)**：这是 C 语言实现中最复杂的部分。它需要实现一个六重嵌套循环（遍历输出通道、输出高、输出宽、输入通道、卷积核高、卷积核宽），并精确地计算内存地址索引，以访问输入数据和权重。

    - **ReLU (`reluImage`)**：遍历图像数据中的每一个浮点数，执行 `x > 0 ? x : 0.0f` 操作。

    - **最大池化 (`max_pooling`)**：实现一个五重循环，在给定的窗口内遍历寻找最大值。

    - **展平 (`image2vector`)**：将三维的 `Image` 数据复制到一个一维的 `Vector` 结构中。

    - **全连接 (`fullyconnect`)**：通过两重循环实现矩阵向量乘法。

4. **内存管理**：C 语言没有自动垃圾回收机制，因此必须手动管理内存。在每个层产生新的输出（如卷积、池化后）时，使用 `malloc` 为其分配内存。在 `main` 函数的末尾，编写一个 `freeResources` 函数，统一使用 `free` 释放所有动态分配的内存，防止内存泄漏。

5. **结果验证**：将 C 语言程序最终计算出的 logits 数组通过 `savetxt` 函数保存为`.txt` 文件。然后使用提供的 `valid_c.py` 脚本读取此文件，并与 PyTorch 模型的输出进行比较，验证其最终误差小于10^{-5}。

```
复杂模型推理误差: 0.0000012356
PyTorch分类结果 (logits):
surprise  : -27.6597
fear      : -10.7381
disgust   : -8.6218
happiness : 14.6702
sadness   : -12.7411
anger     : -15.7253
neutral   : -15.9833
Numpy分类结果 (logits):
surprise  : -27.6597
fear      : -10.7381
disgust   : -8.6218
happiness : 14.6702
sadness   : -12.7411
anger     : -15.7253
neutral   : -15.9833
PyTorch预测: happiness
Numpy预测:  happiness
验证完成!
```

图 8 代码验证结果

4.2 关键代码说明

卷积操作 convolve 的 C 语言实现：

```
// example.cpp
void convolve(Image *input, Image *output, const ConvolutionLayer co
nv) {
    // ... 计算输出尺寸和分配内存 ...

    Image padded;
    pad(input, &padded, conv.padding); // 填充输入

    for (int oc = 0; oc < conv.out_channel; oc++) {
        for (int oh = 0; oh < out_height; oh++) {
            for (int ow = 0; ow < out_width; ow++) {
                float sum = 0.0f;
                for (int ic = 0; ic < conv.in_channel; ic++) {
                    for (int kh = 0; kh < conv.kernel_size; kh++) {
                        for (int kw = 0; kw < conv.kernel_size; kw++)
 {
                            // 计算输入和权重的内存索引
                            int input_h = oh * conv.stride + kh;
                            int input_w = ow * conv.stride + kw;
                            int input_idx = ic * padded.width * padd
ed.height +
                                            input_h * padded.width +
 input_w;
                            int weight_idx =
                                oc * conv.in_channel * conv.kernel_s
ize *
                                    conv.kernel_size +
```

```
                                ic * conv.kernel_size * conv.kernel_
size +
                                kh * conv.kernel_size + kw;
                            // 累加
                            sum += padded.data[input_idx] *
                                conv.weight[weight_idx];
                        }
                    }
                }
                sum += conv.bias[oc]; // 加偏置
                int output_idx =
                    oc * out_width * out_height + oh * out_width + o
w;
                output->data[output_idx] = sum;
            }
        }
    }
    if (conv.padding > 0) free(padded.data);
}
```

这段代码是整个 C 语言实现的心脏。与 Python 版本相比，它要求开发者对多维数组在内存中的线性布局有深刻理解，并能手动、准确地计算索引。这是对底层编程能力的一次重要考验。

main 函数中的完整执行流程：

```
// example.cpp
int main() {
    // ... 初始化和图像读取 ...

    // 初始化各层 (此处代码经过修改以反映复杂模型，原 example.cpp 仅为简单
示例)
    // initConv(&conv1, ...); initBn(&bn1, ...); initPool(&pool1,
 ...);

    // 按顺序执行模型各层
    // convolve(input, &conv1_out, conv1);
    // batch_norm(&conv1_out, &bn1_out, bn1);
    // reluImage(&bn1_out, &relu1_out);
    // max_pooling(&relu1_out, &pool1_out, pool1);
    // ... 重复执行后续卷积块 ...

    // 展平和全连接层
    // image2vector(&pool3_out, &fc_input);
    // fullyconnect(&fc_input, &fc1_out, fc1);
    // ...

    // 找到预测结果并打印
    // ...

    // 释放所有分配的内存
    freeResources(...);
```

```
    return 0;
}
```

通过以上步骤，我们成功地将一个由 PyTorch 训练的 CNN 模型，先后用 NumPy 和 C 语言进行了复现，并保证了计算结果的高度一致性，圆满完成了本次专业综合设计的全部任务要求。