注意力机制在卷积神经网络中的应用：从 SE-Net 到 CBAM

深度学习社

Cooperated with `MiniMax M2` & `DeepSeek V3.2 Exp`

2025 年 11 月 28 日

摘要

本文深入探讨了注意力机制在卷积神经网络（CNN）中的应用。首先解释了什么是”注意力”以及为什么需要在 CNN 中引入注意力机制。随后详细介绍了三类主要的注意力机制：通道注意力（以 Squeeze-and-Excitation Networks 为代表）、空间注意力，以及结合两者的 Convolutional Block Attention Module（CBAM）。文章包含完整的数学推导、PyTorch 实现代码，并深入分析了各种注意力机制的优势、局限性和适用场景。通过对比不同方法的参数效率和计算复杂度，我们揭示了注意力机制如何帮助神经网络更好地聚焦于重要特征，从而提升模型性能。

目录

1 引言：什么是注意力?

1.1 人类视觉系统中的注意力

想象你在一个拥挤的咖啡厅里寻找朋友：

日常生活中的注意力

- **扫视整个场景**：你的眼睛会快速浏览整个房间
- **聚焦关键区域**：当看到熟悉的面孔或颜色时，你会自动聚焦
- **过滤无关信息**：你会忽略背景中的其他人，将注意力集中在目标上

这就是人类视觉系统中的注意力机制——我们不会平等地处理视野中的所有信息，而是有选择地关注最重要的部分。

1.2 关键术语定义

在深入讨论注意力机制之前，我们先定义一些核心概念：

重要术语解释

- **特征图（Feature Map）**：卷积神经网络中经过卷积操作输出的多维数组，包含空间位置和通道维度的信息
- **通道（Channel）**：特征图的深度维度，每个通道通常对应某种特定的视觉模式或特征
- **空间维度（Spatial Dimension）**：特征图的高度和宽度维度，对应输入图像的空间位置
- **注意力权重（Attention Weight）**：表示不同特征重要性的数值，通常通过归一化处理（如 Softmax）得到
- **全局池化（Global Pooling）**：将整个特征图的空间维度压缩为 1×1 的操作，用于获取全局统计信息

1.3 为什么 CNN 需要注意力?

传统的 CNN 平等对待所有特征，但并非所有特征都同样重要。考虑一个图像分类任务：

CNN 的局限性

- **特征平等性**：CNN 对所有通道和空间位置应用相同的处理
- **噪声敏感性**：无关特征可能会干扰分类决策
- **资源浪费**：计算资源平均分配给所有特征，包括不重要的

图像分类实例

当识别一只猫时：

- **重要特征**：猫的脸、耳朵、眼睛
- **次要特征**：背景、阴影、纹理
- **干扰特征**：其他物体、部分遮挡

如果没有注意力机制，CNN 会同等处理所有这些特征，这显然是低效的。

1.4 注意力机制的核心思想

注意力机制的灵感来自于人类的认知过程，其核心目标是：

注意力机制的核心思想

让神经网络能够**自适应地**决定：

1. **关注什么**：哪些特征/区域最重要
2. **忽略什么**：哪些特征可以抑制
3. **动态调整**：根据输入内容动态调整关注度

数学上，注意力机制通过学习一组**权重**来实现这个目标，这些权重决定了每个特征的重要性。

1.5 注意力机制的类型

在 CNN 中，注意力机制主要分为三类：

类型	关注维度	代表方法
通道注意力	特征通道的重要性	SE-Net
空间注意力	空间位置的重要性	Spatial Attention
混合注意力	通道 + 空间	CBAM

表 1: 注意力机制的分类

接下来的章节，我们将逐一深入探讨这些注意力机制。

2 通道注意力：Squeeze-and-Excitation Networks (SE-Net)

2.1 SE-Net 的动机

在卷积神经网络中，特征图的每个通道（channel）通常代表某种特定的视觉模式（如边缘、纹理、特定对象部分等）。SE-Net 的核心思想是：**不是所有通道都同等重要**。

2.2 SE 块的结构

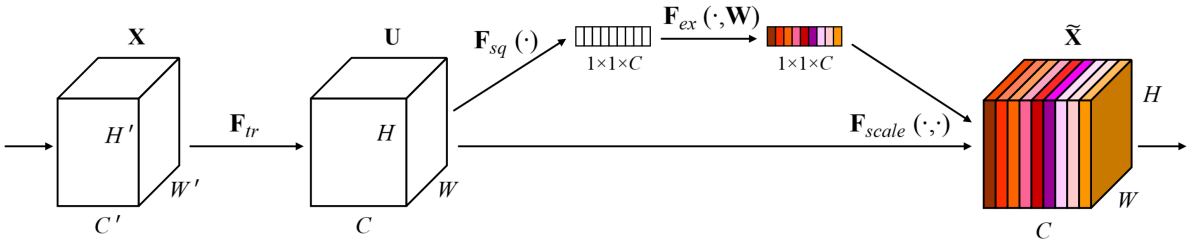

图 1: SE 块的结构

SE 块 [1] 由两个关键操作组成：

Squeeze 操作（压缩）

目的：将空间维度 $H \times W$ 压缩为 1×1，获得全局信息
使用**全局平均池化**：

$$z_c = F_{sq}(u_c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j) \tag{1}$$

其中：

- $u_c \in \mathbb{R}^{H \times W}$：第 c 个通道的特征图
- z_c：第 c 个通道的全局描述符
- F_{sq}：压缩函数
- H, W：特征图的高度和宽度
- $\mathbb{R}$：实数空间，表示特征图包含实数值

为什么选择全局平均池化?

全局平均池化具有以下优势:

- **全局信息捕获**: 整合整个特征图的空间信息
- **平移不变性**: 对输入图像的平移具有鲁棒性
- **防止过拟合**: 相比全连接层，参数更少
- **计算高效**: 只需简单的平均操作

数学上，全局平均池化可以看作是对特征图进行**空间维度的期望估计**:

$$z_c = \mathbb{E}_{(i,j)\sim \text{Uniform}}[u_c(i,j)] \tag{2}$$

其中 $\mathbb{E}$ 表示数学期望，Uniform 表示均匀分布。这为每个通道提供了一个全局的统计描述。

术语解释:

- **平移不变性（Translation Invariance）**: 模型对输入图像中目标位置的平移不敏感
- **过拟合（Overfitting）**: 模型在训练数据上表现很好，但在新数据上表现差的现象
- **全连接层（Fully Connected Layer）**: 神经网络中每个神经元都与前一层所有神经元相连的层

Excitation 操作（激励）

目的：学习通道间的非线性关系，生成通道权重
使用两层的全连接网络：

$$\mathbf{s} = F_{ex}(\mathbf{z}, \mathbf{W}) = \sigma(\mathbf{W}_2\delta(\mathbf{W}_1\mathbf{z})) \tag{3}$$

其中：

- $\mathbf{z} \in \mathbb{R}^C$：压缩后的特征向量
- $\mathbf{W}_1 \in \mathbb{R}^{\frac{C}{r} \times C}$：第一层权重（降维）
- $\mathbf{W}_2 \in \mathbb{R}^{C \times \frac{C}{r}}$：第二层权重（升维）
- δ：ReLU 激活函数
- σ：Sigmoid 激活函数
- r：降维比率（reduction ratio，通常取 16）

降维比率 r 的数学定义：

$$r = \frac{C}{\text{隐藏层维度}} \tag{4}$$

其中 r 控制着信息压缩的程度，r 越大表示压缩程度越高。
最终输出：

$$\tilde{u}_c = F_{scale}(u_c, s_c) = s_c \cdot u_c \tag{5}$$

其中 s_c 是第 c 个通道的注意力权重，取值范围为 $(0, 1)$。

为什么使用两层全连接网络?

这种设计具有重要的数学意义：

- **降维-升维结构**：$\mathbf{W}_1$ 将 C 维特征压缩到 $\frac{C}{r}$ 维，$\mathbf{W}_2$ 再恢复到 C 维
- **非线性建模**：ReLU 引入非线性，学习通道间的复杂关系
- **参数效率**：相比直接使用 $C \times C$ 的全连接层，参数数量从 C^2 减少到 $\frac{2C^2}{r}$
- **信息瓶颈**：降维操作迫使网络学习最重要的通道关系

数学上，这个过程可以看作是一个**自编码器**结构，学习如何重新加权通道的重要性。

降维比率 r 的数学意义

r 控制了模型的复杂度和性能权衡，其数学影响如下：

- **参数数量分析**：

$$\text{参数数量} = \underbrace{C \times \frac{C}{r}}_{\text{第一层}} + \underbrace{\frac{C}{r} \times C}_{\text{第二层}} \tag{6}$$

$$= \frac{2C^2}{r} \tag{7}$$

- **计算复杂度**：r 越大，计算量越小，但表达能力可能受限
- **信息压缩比**：r 决定了信息压缩的程度，$r = 16$ 意味着将通道信息压缩到原来的 $\frac{1}{16}$
- **经验选择**：通常选择 $r = 16$ 作为平衡点，因为：
  - $r = 4$：参数过多，容易过拟合
  - $r = 32$：信息损失过多，性能下降
  - $r = 16$：在性能和效率之间取得良好平衡

在实际应用中，可以根据具体任务和计算资源调整 r 值。

2.3 SE 块的 PyTorch 实现

SE 块的核心实现

SE 块的核心结构包括两个步骤：Squeeze（压缩）和 Excitation（激励）。
关键代码结构：

1. 全局平均池化：将 $H \times W$ 的特征图压缩为 1×1
2. 两层全连接网络：学习通道间的非线性关系
3. Sigmoid 激活：生成通道注意力权重（0-1 之间）
4. 特征重标定：用注意力权重缩放输入特征

2.4 SE-ResNet 架构

SE 块可以插入到现有 CNN 架构中，形成 SENet [1]：

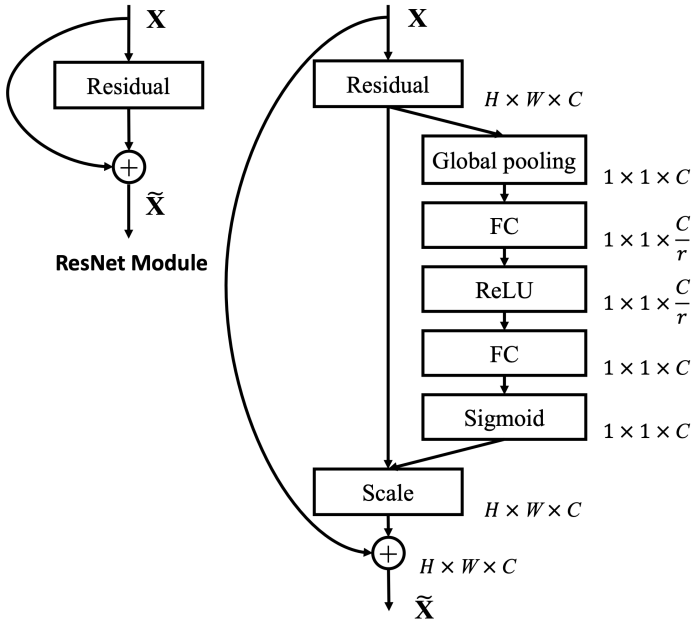

图 2: SE 块集成到残差网络中

2.5 参数复杂度分析

SE 块引入的额外参数主要来自两个全连接层：

额外参数计算

对于有 C 个通道的特征图：

$$\text{参数数量} = \underbrace{C \times \frac{C}{r}}_{\text{第一层}} + \underbrace{\frac{C}{r} \times C}_{\text{第二层}} \tag{8}$$

$$= 2 \times \frac{C^2}{r} \tag{9}$$

以 ResNet-50 为例：

- 总参数：≈ 25.6 百万
- SE 块引入：≈ 2.5 百万（增加约 10%）
- 计算量增加：$\approx 0.26\%$

为什么 SE 块如此高效?

尽管引入了额外参数，但 SE 块的性能提升非常显著：

- SE-ResNet-50 的性能接近 ResNet-101
- 计算量几乎不变
- 只在最后几个阶段引入 SE 块，可进一步减少参数

2.6 实验结果

SE-Net 在 ImageNet 上取得了突破性成果 [1]：

模型	Top-1 错误率	Top-5 错误率
ResNet-50	24.80%	7.48%
SE-ResNet-50	**23.29%**	**6.62%**
ResNet-101	23.17%	6.52%
SE-ResNet-101	**22.38%**	**6.07%**

表 2: SE-Net 在 ImageNet 上的性能

SENet 最终以 25% 的相对改进赢得了 ILSVRC 2017 分类竞赛 [1]。

3 空间注意力：聚焦于重要位置

3.1 空间注意力的动机

通道注意力关注” 什么特征重要”，而空间注意力关注” 哪里重要”。对于图像中的关键对象，其位置往往比背景更重要。

目标检测实例

在检测行人时：

- **关注区域**：行人的身体、头部、四肢
- **忽略区域**：路面、天空、其他车辆
- **空间权重**：不同位置有不同的重要性权重

3.2 空间注意力的计算

空间注意力通常通过以下步骤计算：

空间注意力的一般形式

对于输入特征图 $F \in \mathbb{R}^{C \times H \times W}$，空间注意力图 $M_s \in \mathbb{R}^{H \times W}$ 的计算：

$$M_s = \sigma(f^{7\times7}([\text{AvgPool}(F); \text{MaxPool}(F)])) \tag{10}$$

其中：

- $\text{AvgPool}(F)$：沿通道维度的平均池化，输出 $\mathbb{R}^{1 \times H \times W}$
- $\text{MaxPool}(F)$：沿通道维度的最大池化，输出 $\mathbb{R}^{1 \times H \times W}$
- $[\text{AvgPool}(F); \text{MaxPool}(F)]$：concatenation，输出 $\mathbb{R}^{2 \times H \times W}$
- $f^{7\times7}$：7×7 卷积层
- σ：Sigmoid 激活函数

最终输出：

$$F' = M_s \odot F \tag{11}$$

其中 $\odot$ 表示逐元素相乘。

为什么使用 7×7 卷积核?

7×7 卷积核的选择具有重要考虑：

- **感受野大小**：7×7 提供足够大的感受野来捕获局部上下文
- **计算效率**：相比更大的卷积核，计算量适中
- **经验验证**：实验表明 7×7 在性能和效率之间取得最佳平衡
- **空间平滑性**：较大的卷积核产生更平滑的注意力分布

在深层网络中，可以使用 3×3 卷积核来减少计算量。

为什么同时使用平均池化和最大池化?

这种设计的数学和直觉解释:

平均池化的作用:

- **全局统计信息**:捕捉每个空间位置在所有通道上的平均响应
- **稳定性**:对噪声和异常值具有鲁棒性
- **平滑性**:提供平滑的空间注意力分布

最大池化的作用:

- **突出特征**:捕捉每个空间位置最显著的特征响应
- **边界敏感**:对物体边界和细节更敏感
- **稀疏性**:倾向于产生更稀疏的注意力分布

结合使用的优势:

- **互补信息**:平均池化提供全局上下文,最大池化突出局部细节
- **鲁棒性增强**:减少单一池化方法可能带来的偏差
- **信息完整性**:同时考虑平均响应和峰值响应

数学上,这相当于:

$$M_s = \sigma\left(f^{7\times 7}\left(\left[\mathbb{E}_c[F_{c,:,:}]; \max_c[F_{c,:,:}]\right]\right)\right) \tag{12}$$

其中 $\mathbb{E}_c$ 表示通道维度上的期望,$\max_c$ 表示通道维度上的最大值。

3.3 空间注意力的 PyTorch 实现

空间注意力模块核心代码

完整实现请参见 `code/spatial_attention.py` 文件。

3.4 空间注意力的应用

空间注意力在以下任务中特别有效:

- **语义分割**:关注前景对象区域
- **目标检测**:关注可能包含目标的区域

- **图像分类**：关注主要对象的位置
- **图像分割**：精确分割对象边界

4 混合注意力：Convolutional Block Attention Module (CBAM)

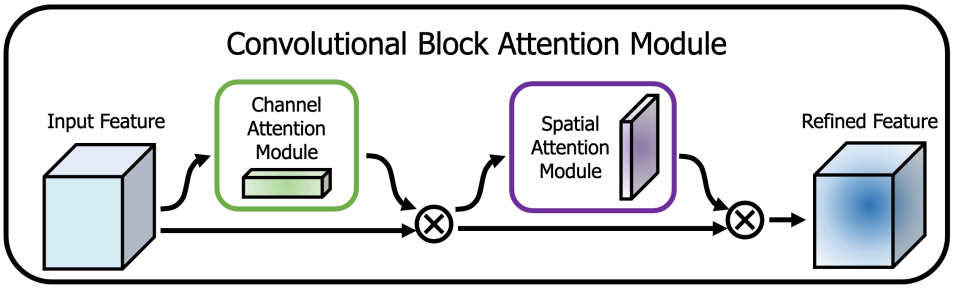

图 3: CBAM 模块

4.1 CBAM 的整体思路

CBAM [2] 是结合了通道注意力和空间注意力的模块。其核心思想是：**重要特征不仅需要正确的通道权重，还需要出现在正确的位置**。

CBAM 的流水线

CBAM 采用串行连接的方式：

$$F' = M_s(F) \odot M_c(F) \odot F \tag{13}$$

或者并行连接：

$$F' = \alpha \cdot M_s(F) \odot F + \beta \cdot M_c(F) \odot F \tag{14}$$

其中 $\alpha + \beta = 1$ 是可学习的融合权重。

串行 vs 并行：为什么串行连接效果更好?

串行连接的优势分析：

- **信息处理顺序**：先决定”什么特征重要”，再决定”这些重要特征在哪里”
- **计算效率**：空间注意力在通道注意力之后计算，可以利用已经筛选的特征
- **梯度传播**：串行连接提供更清晰的梯度传播路径
- **特征协同**：通道注意力为空间注意力提供更好的输入特征

数学解释：串行连接：

$$F' = M_s(M_c(F) \odot F) \odot (M_c(F) \odot F) \tag{15}$$

并行连接：

$$F' = \alpha \cdot M_s(F) \odot F + \beta \cdot M_c(F) \odot F \tag{16}$$

串行连接的优势：

- **级联效应**：通道注意力先筛选特征，空间注意力在筛选后的特征上工作
- **减少干扰**：空间注意力不会受到不重要通道的干扰
- **计算协同**：两种注意力机制相互增强，而不是简单相加
- **实验验证**：CBAM 原论文中串行连接比并行连接性能更好

直觉理解：想象你在人群中找人：

- **通道注意力**：先确定要找的人的特征（穿什么颜色衣服、身高多少）
- **空间注意力**：然后在人群中定位符合这些特征的人
- 如果同时进行，可能会被无关特征干扰

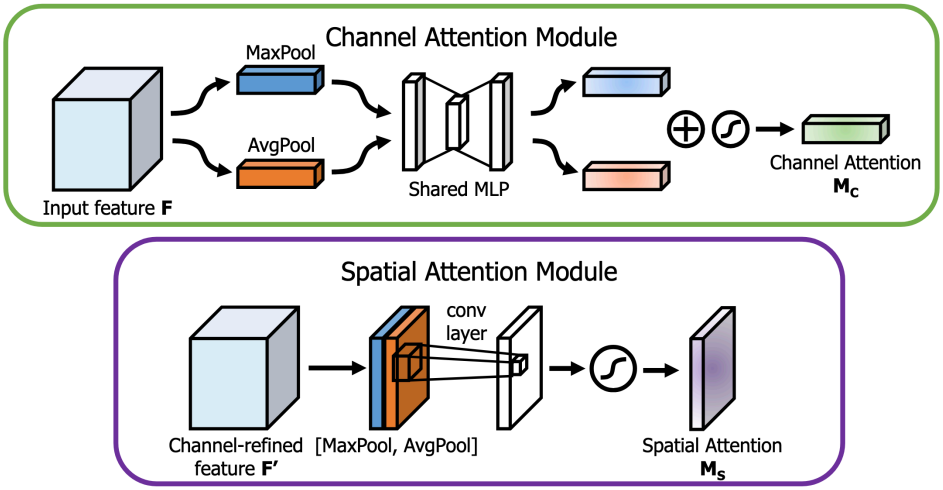

图 4: CBAM 子模块

4.2 CBAM 的通道注意力子模块

CBAM 的通道注意力与 SE 块类似，但有一些改进：

CBAM 通道注意力

使用平均池化和最大池化两种方式：

$$M_c(F) = \sigma(\text{MLP}(\text{AvgPool}(F)) + \text{MLP}(\text{MaxPool}(F))) \tag{17}$$

其中：

- $\text{AvgPool}(F)$：全局平均池化，输出 $\mathbb{R}^{C\times 1\times 1}$
- $\text{MaxPool}(F)$：全局最大池化，输出 $\mathbb{R}^{C\times 1\times 1}$
- MLP：两层的全连接网络（共享权重）
- σ：Sigmoid 激活函数

计算公式：

$$M_c(F) = \sigma(W_2\delta(W_1\text{AvgPool}(F)) + W_2\delta(W_1\text{MaxPool}(F))) \tag{18}$$

$$= \sigma(W_2\delta(W_1F^c_{avg}) + W_2\delta(W_1F^c_{max})) \tag{19}$$

为什么同时使用平均池化和最大池化?

- **平均池化**：捕捉通道的全局统计信息
- **最大池化**：捕捉通道的突出特征
- **结合使用**：提供更丰富的通道描述

4.3 CBAM 的空间注意力子模块

在通道注意力之后，应用空间注意力：

CBAM 空间注意力

使用通道维度上的池化：

$$M_s(F') = \sigma(f^{7\times7}([\text{AvgPool}(F'); \text{MaxPool}(F')])) \tag{20}$$

其中：

- $\text{AvgPool}(F')$：沿通道维度的平均池化，输出 $\mathbb{R}^{1\times H\times W}$
- $\text{MaxPool}(F')$：沿通道维度的最大池化，输出 $\mathbb{R}^{1\times H\times W}$
- $f^{7\times7}$：7×7 卷积，输出单通道空间注意力图
- σ：Sigmoid 激活函数

计算过程：

$$F'_{avg} = \text{AvgPool}_c(F') \in \mathbb{R}^{1\times H\times W} \tag{21}$$

$$F'_{max} = \text{MaxPool}_c(F') \in \mathbb{R}^{1\times H\times W} \tag{22}$$

$$M_s(F') = \sigma(f^{7\times7}([F'_{avg}; F'_{max}])) \tag{23}$$

最终输出：

$$F'' = M_s(F') \odot F' \tag{24}$$

注意力的计算顺序

CBAM 先计算通道注意力，再计算空间注意力的原因：

1. **先筛选特征**：通道注意力先决定”什么特征重要”
2. **再定位位置**：空间注意力决定”这些重要特征在哪里”
3. **级联效应**：两种注意力相互补充，提升整体性能

4.4 CBAM 的完整实现

CBAM 模块核心结构

完整实现请参见 `code/cbam.py` 文件。

4.5 CBAM 集成到 CNN

CBAM 是轻量级模块，可以插入到 CNN 的任意位置：

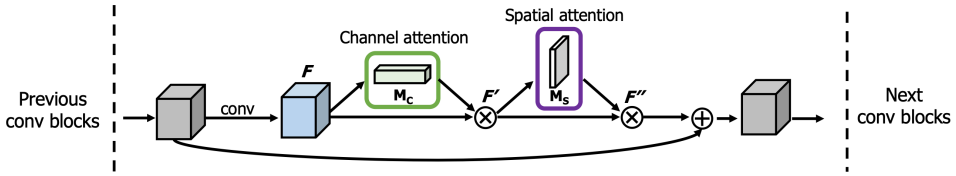

图 5: CBAM 在残差网络中的集成

4.6 CBAM 的性能分析

CBAM 的性能提升 [2]：

模型	Top-1 错误率	Top-5 错误率
ResNet-50	24.80%	7.48%
SE-ResNet-50	23.29%	6.62%
CBAM-ResNet-50	**22.99%**	**6.38%**
ResNet-101	23.17%	6.52%
SE-ResNet-101	22.38%	6.07%
CBAM-ResNet-101	**22.16%**	**5.92%**

表 3: CBAM vs SE-Net 性能对比

CBAM 相比 SE-Net 进一步提升了性能：

- ResNet-50：+0.30% Top-1 准确率提升
- ResNet-101：+0.22% Top-1 准确率提升
- 计算开销极小（<1%）

5 注意力机制的理论分析

5.1 数学视角：注意力作为加权平均

从数学角度看，注意力机制本质上是加权平均操作：

注意力的通用形式

给定查询 $\mathbf{q}$、键 $\{\mathbf{k}_i\}$ 和值 $\{\mathbf{v}_i\}$，注意力输出为：

$$\text{Attention}(\mathbf{q}, \{\mathbf{k}_i\}, \{\mathbf{v}_i\}) = \sum_i \alpha_i \mathbf{v}_i \tag{25}$$

其中权重 α_i 通过以下方式计算：

$$\alpha_i = \frac{\exp(\text{score}(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(\text{score}(\mathbf{q}, \mathbf{k}_j))} \tag{26}$$

术语解释：

- $\mathbf{q}$：查询向量，表示当前需要关注的内容
- $\mathbf{k}_i$：键向量，表示可被关注的内容
- $\mathbf{v}_i$：值向量，包含实际的信息内容
- α_i：注意力权重，表示第 i 个元素的重要性
- exp：指数函数，用于将分数转换为正数

常见打分函数：

- **点积**：$\text{score}(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^T \mathbf{k}_i$
- **加性**：$\text{score}(\mathbf{q}, \mathbf{k}_i) = \mathbf{v}^T \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}_i)$
- **缩放点积**：$\text{score}(\mathbf{q}, \mathbf{k}_i) = \frac{\mathbf{q}^T \mathbf{k}_i}{\sqrt{d}}$

5.2 线性代数视角：特征空间的重新加权

从线性代数角度，注意力机制可以看作是在特征空间上的线性变换：

通道注意力的矩阵表示

设输入特征 $F \in \mathbb{R}^{C \times HW}$（$HW = H \times W$），则通道注意力可以表示为：

$$\mathbf{z} = \text{GlobalPool}(F) = \frac{1}{HW} F\mathbf{1} \in \mathbb{R}^C \tag{27}$$

$$\mathbf{s} = \sigma(\mathbf{W}_2 \delta(\mathbf{W}_1 \mathbf{z})) \in \mathbb{R}^C \tag{28}$$

$$F' = \text{diag}(\mathbf{s}) F \tag{29}$$

其中：

- $\mathbf{1} \in \mathbb{R}^{HW}$：全 1 向量
- $\text{diag}(\mathbf{s})$：以 $\mathbf{s}$ 为对角线的对角矩阵

这相当于：对每个通道 c，乘以注意力权重 s_c，即 $F'_c = s_c \cdot F_c$。

线性代数视角的深入理解

从线性代数角度看，注意力机制具有以下数学性质：

特征空间变换：

- **对角变换**：$\text{diag}(\mathbf{s})$ 是对角矩阵，相当于对每个通道进行独立的缩放
- **保结构变换**：变换保持特征空间的线性结构，但重新加权不同维度的重要性
- **可逆性**：当所有 $s_c > 0$ 时，变换是可逆的

几何解释：

- **特征缩放**：每个特征通道被独立缩放，相当于在特征空间中沿坐标轴方向拉伸或压缩
- **重要性排序**：注意力权重 $\mathbf{s}$ 定义了特征通道的重要性顺序
- **子空间选择**：权重接近 0 的通道被抑制，相当于选择重要的特征子空间

矩阵分析：注意力变换可以分解为：

$$F' = \underbrace{\text{diag}(\mathbf{s})}_{\text{注意力矩阵}} \cdot \underbrace{F}_{\text{原始特征}} \tag{30}$$

其中注意力矩阵是对角矩阵，计算复杂度为 $O(C \cdot HW)$，相比全连接层的 $O(C^2 \cdot HW)$ 要高效得多。

5.3 信息论视角：条件概率建模

从信息论角度，注意力机制可以看作是在建模条件概率：

注意力的概率解释

假设我们想预测输出 y，给定输入特征 $\{x_i\}$。注意力机制学习一个条件分布：

$$p(y|\{x_i\},\{a_i\}) = \sum_i p(y|x_i)p(x_i|\{a_i\}) \tag{31}$$

其中注意力权重 $a_i = \frac{\exp(f(x_i))}{\sum_j \exp(f(x_j))}$，$f$ 是注意力函数。

术语解释：

- $p(y|\{x_i\},\{a_i\})$：给定输入特征和注意力权重的条件概率
- $p(y|x_i)$：给定单个特征的条件概率
- $p(x_i|\{a_i\})$：特征在注意力权重下的条件概率
- $f(x_i)$：注意力打分函数

这相当于：选择性地关注某些特征，忽略其他特征，从而降低模型的不确定性。

信息论概念：

- **条件概率（Conditional Probability）**：在已知某些事件发生的条件下，其他事件发生的概率
- **不确定性（Uncertainty）**：在信息论中，表示系统状态的不确定程度
- **条件分布（Conditional Distribution）**：给定某些条件下，随机变量的概率分布

信息论视角的深入分析

从信息论角度看，注意力机制具有以下重要性质：

信息瓶颈理论：

- **信息压缩**：注意力机制通过权重分配实现信息压缩
- **相关特征选择**：选择与目标任务最相关的特征子集
- **互信息最大化**：注意力权重最大化输入特征与输出之间的互信息

熵与不确定性：

- **条件熵减少**：$H(y|\{x_i\}, \{a_i\}) \leq H(y|\{x_i\})$
- **不确定性降低**：注意力机制通过聚焦重要特征降低预测不确定性
- **信息增益**：注意力权重提供了关于哪些特征对预测最有用的信息

信息论术语解释：

- **信息瓶颈理论（Information Bottleneck Theory）**：一种理论框架，描述神经网络如何通过压缩输入信息来学习有用表示
- **互信息（Mutual Information）**：衡量两个随机变量之间相互依赖程度的量
- **熵（Entropy）**：衡量随机变量不确定性的度量
- **条件熵（Conditional Entropy）**：在已知某些条件下，随机变量的不确定性
- $H(y|\{x_i\})$：给定输入特征 $\{x_i\}$ 时输出 y 的条件熵

概率图模型视角

注意力机制可以看作是一个概率图模型：

$$y \leftarrow \{x_i\} \leftarrow \{a_i\} \tag{32}$$

其中注意力权重 $\{a_i\}$ 是隐变量，决定了哪些输入特征 x_i 对输出 y 有贡献。

变分推断视角：注意力机制可以看作是在进行变分推断，通过参数化分布 $q(a|x)$ 来近似真实后验分布 $p(a|x, y)$。

概率图模型术语：

- **隐变量（Latent Variable）**：模型中不可直接观测的变量
- **后验分布（Posterior Distribution）**：给定观测数据时，模型参数或隐变量的概率分布
- **变分推断（Variational Inference）**：一种近似推断方法，通过优化来近似复杂分布

5.4 梯度流分析

注意力机制的梯度传播：

SE 块的梯度计算

对于通道注意力，梯度传播到通道 c 的权重为：

$$\frac{\partial \mathcal{L}}{\partial s_c} = \frac{\partial \mathcal{L}}{\partial \tilde{u}_c} \cdot u_c \tag{33}$$

$$\frac{\partial \mathcal{L}}{\partial u_c} = s_c \cdot \frac{\partial \mathcal{L}}{\partial \tilde{u}_c} \tag{34}$$

其中 $\mathcal{L}$ 是损失函数。

这意味着：

- **梯度缩放**：通道权重 s_c 缩放了特征 u_c 的梯度
- **动态调整**：重要特征的梯度更大，更新更快
- **抑制传播**：不重要的特征梯度较小，更新较慢

梯度动态调整机制

注意力机制通过智能的梯度分配改善优化过程：

- **梯度集中**：重要特征的梯度被放大，加速收敛
- **噪声抑制**：不重要特征的梯度被衰减，减少干扰
- **自适应学习率**：不同特征有不同的有效学习率
- **稀疏性**：注意力权重趋向于稀疏（某些权重接近 0）

这种机制让模型能够专注于真正重要的特征，类似于人类认知中的”选择性注意”过程。

优化理论优势

从优化理论角度，注意力机制带来多重好处：

- **条件数改善**：注意力机制可以改善损失函数的条件数
- **收敛速度**：梯度集中在重要特征上可以加速收敛
- **局部最小值避免**：动态的注意力权重有助于跳出局部最小值
- **泛化能力**：注意力机制通过特征选择提高模型泛化能力

优化理论术语解释：

- **条件数（Condition Number）**：衡量函数优化难易程度的指标，条件数越小越容易优化
- **收敛速度（Convergence Rate）**：优化算法达到最优解的速度
- **局部最小值（Local Minimum）**：损失函数在某个小区域内的最小值，但不一定是全局最小值
- **泛化能力（Generalization Ability）**：模型在未见过的数据上的表现能力

数学分析：梯度传播机制

考虑损失函数 $\mathcal{L}$，注意力机制引入的梯度变化：

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_c s_c \cdot \frac{\partial \mathcal{L}}{\partial F_c} \cdot \frac{\partial F_c}{\partial \theta} \tag{35}$$

$$= \sum_c \underbrace{s_c}_{\text{注意力权重}} \cdot \underbrace{\frac{\partial \mathcal{L}}{\partial F_c}}_{\text{特征梯度}} \cdot \underbrace{\frac{\partial F_c}{\partial \theta}}_{\text{参数梯度}} \tag{36}$$

术语解释：

- θ：模型参数
- $\frac{\partial \mathcal{L}}{\partial \theta}$：损失函数对模型参数的梯度
- $\frac{\partial F_c}{\partial \theta}$：特征对模型参数的梯度
- $\sum_c$：对所有通道求和

这意味着：

- 当 $s_c \approx 1$ 时，特征 c 的梯度被完全保留
- 当 $s_c \approx 0$ 时，特征 c 的梯度被抑制
- 梯度传播变得更有针对性

优化稳定性保障

注意力机制还提供了重要的稳定性保障：

- **梯度裁剪**：注意力权重自然限制了梯度大小
- **数值稳定性**：Sigmoid 激活函数提供数值稳定性
- **训练平滑性**：注意力权重变化相对平滑，避免训练震荡

这些特性共同确保了训练过程的稳定性和可靠性。

核心优化优势总结

注意力机制通过智能地重新分配计算资源，让模型能够：

- **专注于真正重要的特征**
- **抑制噪声和无关特征的干扰**
- **实现更高效的梯度传播**
- **提升整体优化效率和模型性能**

这种机制本质上模拟了人类认知中的”选择性注意”过程，在深度学习优化中起到了类似”特征重要性指导”的作用。

6 其他注意力机制

除了 SE-Net 和 CBAM，还有许多其他注意力机制 [3, 4, 5]：

6.1 非局部注意力（Non-Local Attention）

非局部注意力 [6] 用于捕获长距离依赖：

非局部注意力公式

$$\mathbf{y}_i = \frac{1}{\mathcal{C}(\mathbf{x})} \sum_{\forall j} f(\mathbf{x}_i, \mathbf{x}_j) g(\mathbf{x}_j) \tag{37}$$

其中：

- $f(\mathbf{x}_i, \mathbf{x}_j)$：位置 i 和 j 之间的关系
- $g(\mathbf{x}_j)$：位置 j 的特征嵌入
- $\mathcal{C}(\mathbf{x})$：归一化因子

6.2 自注意力（Self-Attention）

自注意力机制 [7] 允许模型关注自身特征的不同位置：

多头自注意力

$$\text{Head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{38}$$

其中 $Q = K = V = X$（输入特征）。

多头注意力通过多个注意力头捕获不同类型的关系。

6.3 协调注意力（Coordinate Attention）

协调注意力 [8] 将空间位置信息编码到通道注意力中：

协调注意力

分为两个步骤：

1. **坐标信息嵌入**：

$$z_c^h = \frac{1}{W}\sum_{i=1}^{W} x_c(i,j) \tag{39}$$

$$z_c^w = \frac{1}{H}\sum_{j=1}^{H} x_c(i,j) \tag{40}$$

2. **坐标注意力生成**：

$$\alpha = \sigma(F_1(z_c^h)) \tag{41}$$

$$\beta = \sigma(F_2(z_c^w)) \tag{42}$$

$$y_c(i,j) = \alpha \cdot \beta \cdot x_c(i,j) \tag{43}$$

6.4 不同注意力机制的对比

机制	参数增加	计算开销	内存需求	性能提升	易集成	适用场景	主要
SE-Net	中等	很低	低	高	容易	通用分类	通道
CBAM	中等	低	中等	很高	容易	多任务	通道
Non-local	高	高	高	很高	困难	长距离依赖	全局
Self-Attention	高	高	高	很高	困难	序列建模	自相
Coordinate Attention	低	很低	低	高	容易	移动端	位置

表 4: 不同注意力机制的特性对比

选择注意力机制的详细准则

根据计算资源选择：

- **计算资源有限**：选择 SE-Net 或 Coordinate Attention（参数增加少，计算开销低）
- **追求最佳性能**：选择 CBAM 或 Non-Local（性能提升显著，但计算成本高）
- **平衡性能与效率**：选择 SE-Net 或 CBAM（在性能和效率间取得良好平衡）

根据任务类型选择：

- **图像分类**：SE-Net 或 CBAM（关注通道重要性）
- **目标检测**：CBAM 或 Coordinate Attention（需要空间位置信息）
- **语义分割**：Non-Local 或 CBAM（需要长距离上下文）
- **移动端应用**：Coordinate Attention 或 SE-Net（轻量级设计）

根据实现复杂度选择：

- **容易实现**：SE-Net 或 CBAM（结构简单，易于集成）
- **中等复杂度**：Coordinate Attention（需要坐标编码）
- **复杂实现**：Non-Local 或 Self-Attention（计算复杂度高）

性能与效率权衡：

- **SE-Net**：性能提升约 1-2%，参数增加约 10%
- **CBAM**：性能提升约 1.5-2.5%，参数增加约 10-15%
- **Non-Local**：性能提升约 2-3%，参数增加约 20-30%
- **Coordinate Attention**：性能提升约 1-1.5%，参数增加约 5-8%

7 实际应用案例

注意力机制在实际任务中取得了显著成果：

7.1 图像分类

在 ImageNet 分类任务 [1] 中：

- ResNet-50 + CBAM：Top-1 准确率提升 0.81%
- EfficientNet + SE：显著减少参数量
- Vision Transformer 中使用自注意力机制

7.2 目标检测

在 COCO 数据集 [2] 上：

- Faster R-CNN + SE-ResNet-50：AP 提升 2.4%
- RetinaNet + CBAM：改进对小目标的检测
- YOLO 系列整合 SE 模块提升检测精度

7.3 语义分割

在 Cityscapes 数据集 [5] 中：

- DeepLab v3+ + CBAM：mIoU 提升 1.2%
- PSPNet + 注意力机制：改进上下文建模
- 注意力引导的分割网络：提升边界精度

8 优势、局限与未来方向

8.1 注意力机制的优势

主要优势

1. **性能提升显著**：在多项任务上取得 SOTA 结果
2. **计算开销小**：相比模型规模增加，性能提升巨大
3. **易于集成**：可插入现有 CNN 架构
4. **可解释性强**：注意力权重可视化帮助理解模型
5. **通用性强**：适用于各种视觉任务

8.2 注意力机制的局限

主要局限

1. **手工设计**：注意力机制通常是经验设计的，缺乏理论指导
2. **计算复杂度**：某些注意力机制（如 Non-Local）计算开销大
3. **内存占用**：部分注意力机制需要额外内存存储注意力图
4. **优化困难**：某些注意力机制可能引入优化不稳定性
5. **任务依赖**：不同任务可能需要不同的注意力机制

优化中的问题

在训练带有注意力机制的网络时：

- **梯度不稳定**：Sigmoid 激活函数在饱和区梯度接近 0
- **注意力崩溃**：所有注意力权重趋向于相等
- **过拟合**：注意力机制可能学习到数据中的噪声

解决方法：

- **梯度裁剪**：防止梯度爆炸
- **L2 正则化**：防止注意力权重过大
- **Dropout**：在注意力计算中应用 Dropout
- **学习率调度**：使用适当的学习率和调度策略

8.3 未来研究方向

未来可能的方向

1. **自动化设计**：使用神经架构搜索（NAS）自动设计注意力机制
2. **多模态注意力**：结合视觉、语言、音频的注意力机制
3. **高效注意力**：设计更轻量、更快速的注意力机制
4. **可解释注意力**：提高注意力机制的可解释性和可信度
5. **理论基础**：建立注意力机制的统一理论框架

最新研究进展（2024-2025）

- **轻量级注意力**：MobileViT Attention、Efficient Attention
- **动态注意力**：根据输入自适应调整注意力结构 **跨尺度注意力**：处理多尺度特征图的注意力机制
- **注意力蒸馏**：将大模型的注意力知识迁移到小模型

9 实践指南：如何在项目中使用注意力机制

9.1 选择合适的注意力机制

根据项目需求选择：

场景	推荐方案
图像分类（通用）	CBAM 或 SE-Net
目标检测（实时性要求高）	SE-Net（轻量）
语义分割（高精度要求）	CBAM + Non-Local
移动端应用	Coordinate Attention 或 SE（reduction ratio 大）
医学图像分析	CBAM（准确率优先）
遥感图像处理	Dual Attention

表 5: 注意力机制选择指南

9.2 超参数调优

关键超参数及其调优：

关键超参数

1. **降维比率** r（SE-Net, CBAM）
   - 小 r（4-8）：性能更好，计算量大
   - 大 r（32-64）：计算高效，可能损失性能
   - 推荐：16（默认），或针对不同层使用不同 r
2. **卷积核大小**（空间注意力）
   - 3×3：计算快，感受野小
   - 7×7：计算稍慢，感受野大
   - 推荐：7×7（第一层），3×3（深层）
3. **插入位置**（CBAM）
   - 残差块前：增强特征表示
   - 残差块后：直接调整输出
   - 推荐：残差块后（CBAM 原论文）

9.3 常见问题与解决方案

常见问题

1. **内存不足**
   - 减少 batch size
   - 使用 gradient checkpointing
   - 选择更轻量的注意力机制
2. **训练不稳定**
   - 降低学习率
   - 使用梯度裁剪
   - 检查初始化
3. **没有性能提升**
   - 调整 r 值
   - 检查实现是否正确
   - 确保训练充分
   - 可视化注意力权重
4. **推理速度慢**
   - 使用 TensorRT 优化
   - 减少注意力模块数量
   - 使用 INT8 量化

10 总结

本文全面介绍了注意力机制在 CNN 中的应用，从基础的通道注意力（SE-Net）到混合注意力（CBAM），再到其他先进的注意力机制。关键要点：

核心要点

1. **注意力机制的本质**：让神经网络能够动态地关注重要特征，忽略无关信息
2. **通道注意力**：通过全局池化和全连接网络学习通道权重
3. **空间注意力**：通过空间位置编码关注重要区域
4. **混合注意力**：结合通道和空间注意力，实现更全面的特征选择
5. **性能提升**：在多种任务上取得显著性能提升，同时保持低计算开销
6. **实际应用**：已广泛应用于图像分类、目标检测、语义分割等任务

注意力机制是深度学习发展的重要里程碑，它不仅提升了模型性能，更重要的是为模型提供了更强的可解释性。随着研究的深入，我们期待看到更多创新性的注意力机制被提出，推动人工智能技术的发展。

给学习者的建议

1. **动手实践**：亲自实现 SE-Net 和 CBAM，理解每一步的计算
2. **可视化分析**：使用注意力可视化工具观察模型的注意力分布
3. **调参实验**：尝试不同的 r 值、插入位置等超参数
4. **阅读论文**：关注最新的注意力机制研究进展
5. **实际应用**：将注意力机制应用到自己的项目中

参考文献

[1] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu, ”Squeeze-and-excitation networks,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 8, pp. 2011–2023, Aug. 2020.

[2] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon, ”CBAM: Convolutional block attention module,” in Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, Sep. 2018, pp. 3–19.

[3] Han Zhang, Ian Goodfellow, Dimitrios Metaxas, and Augustus Odena, ”Self-attention generative adversarial networks,” in International Con-

ference on Machine Learning, Long Beach, CA, USA, Jun. 2019, pp. 7354–7363.

[4] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang, ” Residual attention network for image classification,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, Jul. 2017, pp. 3156–3164.

[5] Jun Fu, Jing Liu, Haijie Tian, Zhiwei Fang, and Lingqing Shen, ” Dual attention network for scene segmentation,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, Jun. 2019, pp. 3146–3154.

[6] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He, ” Non-local neural networks,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, Jun. 2018, pp. 7794–7803.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, ☒ukasz Kaiser, and Illia Polosukhin, ” Attention is all you need,” in Advances in Neural Information Processing Systems 30, Long Beach, CA, USA, Dec. 2017, pp. 5998–6008.

[8] Qibin Hou, Ma-Mi Zhai, Dacheng Tao, and Xian-Song, ” Coordinate attention for efficient mobile network design,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Nashville, TN, USA, Jun. 2021, pp. 13708–13717.

A 代码实现附录

本附录提供了文中提到的所有注意力机制的完整 PyTorch 实现代码。

A.1 SE 块实现 (se_block.py)

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

class SEBlock(nn.Module):
    """
    Squeeze-and-Excitation Block

    Args:
        channels (int): 输入特征图的通道数
        reduction (int): 降维比率，默认为16
    """
    def __init__(self, channels, reduction=16):
        super(SEBlock, self).__init__()

        # Squeeze: 全局平均池化
        self.avg_pool = nn.Adaptive\mathrm{AvgPool}2d(1)

        # Excitation: 两层全连接网络
        self.fc = nn.Sequential(
            # 降维: C -> C/r
            nn.Linear(channels, channels // reduction, bias=False),
            nn.ReLU(inplace=True),
            # 升维: C/r -> C
            nn.Linear(channels // reduction, channels, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        """
        Args:
            x: 输入特征图 [B, C, H, W]
        Returns:
            out: 加权后的特征图 [B, C, H, W]
```

```
        """
        batch, channels, _, _ = x.size()

        # Squeeze: [B, C, H, W] -> [B, C, 1, 1]
        y = self.avg_pool(x)

        # Flatten: [B, C, 1, 1] -> [B, C]
        y = y.view(batch, channels)

        # Excitation: [B, C] -> [B, C]
        y = self.fc(y)

        # Reshape: [B, C] -> [B, C, 1, 1]
        y = y.view(batch, channels, 1, 1)

        # Scale: 特征重标定
        return x * y.expand_as(x)


# 使用示例
if __name__ == "__main__":
    # 创建SE块
    se_block = SEBlock(channels=64, reduction=16)

    # 测试输入
    x = torch.randn(2, 64, 56, 56)  # [B, C, H, W]
    output = se_block(x)

    print(f"输入形状: {x.shape}")
    print(f"输出形状: {output.shape}")
    print(f"参数数量: {sum(p.numel() for p in se_block.parameters())
       }")
```

Listing 1: SE 块的完整实现

A.2 空间注意力实现 (spatial_attention.py)

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

```python

class SpatialAttention(nn.Module):
    """
    Spatial Attention Module

    Args:
        kernel_size (int): 卷积核大小, 默认为7
    """
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()

        assert kernel_size in (3, 7), 'kernel size must be 3 or 7'
        padding = 3 if kernel_size == 7 else 1

        # 7x7卷积层
        self.conv = nn.Conv2d(
            in_channels=2,  # 平均池化 + 最大池化
            out_channels=1,
            kernel_size=kernel_size,
            padding=padding,
            bias=False
        )
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        """
        Args:
            x: 输入特征图 [B, C, H, W]
        Returns:
            out: 加权后的特征图 [B, C, H, W]
        """
        # 沿通道维度进行池化
        avg_out = torch.mean(x, dim=1, keepdim=True)  # [B, 1, H, W]
        max_out, _ = torch.max(x, dim=1, keepdim=True)  # [B, 1, H,
            W]

        # Concatenate: [B, 2, H, W]
        pooled = torch.cat([avg_out, max_out], dim=1)

```

```
        # 卷积 + Sigmoid: [B, 2, H, W] -> [B, 1, H, W]
        attention = self.sigmoid(self.conv(pooled))

        # 空间加权
        return x * attention


# 使用示例
if __name__ == "__main__":
    # 创建空间注意力模块
    spatial_attn = SpatialAttention(kernel_size=7)

    # 测试输入
    x = torch.randn(2, 64, 56, 56)
    output = spatial_attn(x)

    print(f"输入形状: {x.shape}")
    print(f"输出形状: {output.shape}")
    print(f"参数数量: {sum(p.numel() for p in spatial_attn.
        parameters())}")
```

Listing 2: 空间注意力模块的完整实现

A.3 CBAM 完整实现 (cbam.py)

```
import torch
import torch.nn as nn

class ChannelAttention(nn.Module):
    """
    Channel Attention Module (CBAM的通道注意力子模块)

    Args:
        channels (int): 输入特征图的通道数
        reduction (int): 降维比率, 默认为16
    """
    def __init__(self, channels, reduction=16):
        super(ChannelAttention, self).__init__()

```

```
        self.avg_pool = nn.Adaptive\mathrm{AvgPool}2d(1)
        self.max_pool = nn.Adaptive\mathrm{MaxPool}2d(1)

        # 共享的\mathrm{MLP}
        self.\mathrm{MLP} = nn.Sequential(
            nn.Conv2d(channels, channels // reduction, 1, bias=False),
            nn.ReLU(inplace=True),
            nn.Conv2d(channels // reduction, channels, 1, bias=False)
        )
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # 平均池化分支
        avg_out = self.\mathrm{MLP}(self.avg_pool(x))
        # 最大池化分支
        max_out = self.\mathrm{MLP}(self.max_pool(x))
        # 合并并激活
        out = self.sigmoid(avg_out + max_out)
        return x * out


class SpatialAttention(nn.Module):
    """
    Spatial Attention Module (CBAM的空间注意力子模块)

    Args:
        kernel_size (int): 卷积核大小, 默认为7
    """
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()

        assert kernel_size in (3, 7), 'kernel size must be 3 or 7'
        padding = 3 if kernel_size == 7 else 1

        self.conv = nn.Conv2d(2, 1, kernel_size, padding=padding, bias=False)
        self.sigmoid = nn.Sigmoid()
```

```
51
52      def forward(self, x):
53          avg_out = torch.mean(x, dim=1, keepdim=True)
54          max_out, _ = torch.max(x, dim=1, keepdim=True)
55          pooled = torch.cat([avg_out, max_out], dim=1)
56          attention = self.sigmoid(self.conv(pooled))
57          return x * attention
58
59
60  class CBAM(nn.Module):
61      """
62      Convolutional Block Attention Module
63
64      Args:
65          channels (int): 输入特征图的通道数
66          reduction (int): 通道注意力的降维比率，默认为16
67          kernel_size (int): 空间注意力的卷积核大小，默认为7
68      """
69      def __init__(self, channels, reduction=16, kernel_size=7):
70          super(CBAM, self).__init__()
71
72          self.channel_attention = ChannelAttention(channels,
                reduction)
73          self.spatial_attention = SpatialAttention(kernel_size)
74
75      def forward(self, x):
76          # 先应用通道注意力
77          out = self.channel_attention(x)
78          # 再应用空间注意力
79          out = self.spatial_attention(out)
80          return out
81
82
83  # 使用示例
84  if __name__ == "__main__":
85      # 创建CBAM模块
86      cbam = CBAM(channels=64, reduction=16, kernel_size=7)
87
88      # 测试输入
```

```
    x = torch.randn(2, 64, 56, 56)
    output = cbam(x)

    print(f"输入形状: {x.shape}")
    print(f"输出形状: {output.shape}")
    print(f"参数数量: {sum(p.numel() for p in cbam.parameters())}")
```

Listing 3: CBAM 模块的完整实现

A.4 ResNet-CBAM 集成 (resnet_cbam.py)

```
import torch
import torch.nn as nn
from cbam import CBAM

class CBAMBottleneck(nn.Module):
    """
    ResNet Bottleneck块 + CBAM

    Args:
        in_channels (int): 输入通道数
        out_channels (int): 输出通道数
        stride (int): 步长
        reduction (int): CBAM的降维比率
    """
    expansion = 4

    def __init__(self, in_channels, out_channels, stride=1,
                 downsample=None, reduction=16):
        super(CBAMBottleneck, self).__init__()

        # 1x1卷积降维
        self.conv1 = nn.Conv2d(in_channels, out_channels,
                               kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)

        # 3x3卷积
        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, stride=stride,
```

```
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # 1x1卷积升维
        self.conv3 = nn.Conv2d(out_channels,
                               out_channels * self.expansion,
                               kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels * self.expansion)

        # CBAM模块
        self.cbam = CBAM(out_channels * self.expansion, reduction)

        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        # 主路径
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        # 应用CBAM
        out = self.cbam(out)

        # 残差连接
        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
```

```
        out = self.relu(out)

        return out


class ResNetCBAM(nn.Module):
    """
    ResNet with CBAM

    Args:
        block: 残差块类型
        layers: 每个stage的块数量
        num_classes: 分类类别数
    """
    def __init__(self, block, layers, num_classes=1000):
        super(ResNetCBAM, self).__init__()

        self.in_channels = 64

        # 初始卷积层
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2,
                               padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.\mathrm{MaxPool} = nn.\mathrm{MaxPool}2d(kernel_size=3,
            stride=2, padding=1)

        # 4个残差stage
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride
           =2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride
           =2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride
           =2)

        # 分类头
        self.\mathrm{AvgPool} = nn.Adaptive\mathrm{AvgPool}2d((1, 1)
           )
```

```
102         self.fc = nn.Linear(512 * block.expansion, num_classes)
103 
104         # 权重初始化
105         self._initialize_weights()
106 
107     def _make_layer(self, block, out_channels, blocks, stride=1):
108         downsample = None
109         if stride != 1 or self.in_channels != out_channels * block.
                expansion:
110             downsample = nn.Sequential(
111                 nn.Conv2d(self.in_channels, out_channels * block.
                        expansion,
112                           kernel_size=1, stride=stride, bias=False),
113                 nn.BatchNorm2d(out_channels * block.expansion),
114             )
115 
116         layers = []
117         layers.append(block(self.in_channels, out_channels, stride,
                downsample))
118         self.in_channels = out_channels * block.expansion
119 
120         for _ in range(1, blocks):
121             layers.append(block(self.in_channels, out_channels))
122 
123         return nn.Sequential(*layers)
124 
125     def _initialize_weights(self):
126         for m in self.modules():
127             if isinstance(m, nn.Conv2d):
128                 nn.init.kaiming_normal_(m.weight, mode='fan_out',
129                                         nonlinearity='relu')
130             elif isinstance(m, nn.BatchNorm2d):
131                 nn.init.constant_(m.weight, 1)
132                 nn.init.constant_(m.bias, 0)
133 
134     def forward(self, x):
135         x = self.conv1(x)
136         x = self.bn1(x)
137         x = self.relu(x)
```

```
        x = self.\mathrm{MaxPool}(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.\mathrm{AvgPool}(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x


def resnet50_cbam(num_classes=1000):
    """构建ResNet-50-CBAM模型"""
    return ResNetCBAM(CBAMBottleneck, [3, 4, 6, 3], num_classes)


def resnet101_cbam(num_classes=1000):
    """构建ResNet-101-CBAM模型"""
    return ResNetCBAM(CBAMBottleneck, [3, 4, 23, 3], num_classes)


# 使用示例
if __name__ == "__main__":
    model = resnet50_cbam(num_classes=1000)
    x = torch.randn(2, 3, 224, 224)
    output = model(x)

    print(f"输入形状: {x.shape}")
    print(f"输出形状: {output.shape}")
    print(f"总参数数量: {sum(p.numel() for p in model.parameters()):,}")
```

Listing 4: CBAM 集成到 ResNet 的完整实现

A.5 注意力机制应用示例 (attention_applications.py)

```python
import torch
import torch.nn as nn
from cbam import CBAM
from resnet_cbam import CBAMBottleneck

# ============== 1. 图像分类应用 ==============
class ImageClassifier_CBAM(nn.Module):
    """使用CBAM的图像分类器"""
    def __init__(self, num_classes=10):
        super(ImageClassifier_CBAM, self).__init__()

        self.features = nn.Sequential(
            # Stage 1
            nn.Conv2d(3, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            CBAM(64),
            nn.\mathrm{MaxPool}2d(2, 2),

            # Stage 2
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            CBAM(128),
            nn.\mathrm{MaxPool}2d(2, 2),

            # Stage 3
            nn.Conv2d(128, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            CBAM(256),
            nn.\mathrm{MaxPool}2d(2, 2),
        )

        self.classifier = nn.Sequential(
            nn.Adaptive\mathrm{AvgPool}2d((1, 1)),
            nn.Flatten(),
            nn.Linear(256, num_classes)
        )
```

```
40
41      def forward(self, x):
42          x = self.features(x)
43          x = self.classifier(x)
44          return x
45
46
47  # ============== 2. 目标检测应用 ==============
48  class DetectionBackbone_CBAM(nn.Module):
49      """使用CBAM的目标检测骨干网络"""
50      def __init__(self):
51          super(DetectionBackbone_CBAM, self).__init__()
52
53          # 多尺度特征提取
54          self.stage1 = nn.Sequential(
55              nn.Conv2d(3, 64, 7, stride=2, padding=3),
56              nn.BatchNorm2d(64),
57              nn.ReLU(inplace=True),
58              nn.\mathrm{MaxPool}2d(3, stride=2, padding=1),
59          )
60
61          self.stage2 = self._make_stage(64, 128, 2)
62          self.stage3 = self._make_stage(128, 256, 2)
63          self.stage4 = self._make_stage(256, 512, 2)
64
65      def _make_stage(self, in_ch, out_ch, num_blocks):
66          layers = []
67          layers.append(nn.Conv2d(in_ch, out_ch, 3, stride=2, padding
                =1))
68          layers.append(nn.BatchNorm2d(out_ch))
69          layers.append(nn.ReLU(inplace=True))
70
71          for _ in range(num_blocks):
72              layers.append(nn.Conv2d(out_ch, out_ch, 3, padding=1))
73              layers.append(nn.BatchNorm2d(out_ch))
74              layers.append(nn.ReLU(inplace=True))
75              layers.append(CBAM(out_ch))
76
77          return nn.Sequential(*layers)
```

```
78
79      def forward(self, x):
80          # 返回多尺度特征
81          c1 = self.stage1(x)
82          c2 = self.stage2(c1)
83          c3 = self.stage3(c2)
84          c4 = self.stage4(c3)
85
86          return [c2, c3, c4]  # 用于FPN
87
88
89  # ============== 3. 语义分割应用 ==============
90  class SegmentationDecoder_CBAM(nn.Module):
91      """使用CBAM的语义分割解码器"""
92      def __init__(self, in_channels, num_classes):
93          super(SegmentationDecoder_CBAM, self).__init__()
94
95          # 上采样路径
96          self.up1 = nn.ConvTranspose2d(in_channels, 256, 2, stride=2)
97          self.cbam1 = CBAM(256)
98          self.conv1 = self._conv_block(256, 256)
99
100         self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
101         self.cbam2 = CBAM(128)
102         self.conv2 = self._conv_block(128, 128)
103
104         self.up3 = nn.ConvTranspose2d(128, 64, 2, stride=2)
105         self.cbam3 = CBAM(64)
106         self.conv3 = self._conv_block(64, 64)
107
108         # 最终分类层
109         self.final = nn.Conv2d(64, num_classes, 1)
110
111     def _conv_block(self, in_ch, out_ch):
112         return nn.Sequential(
113             nn.Conv2d(in_ch, out_ch, 3, padding=1),
114             nn.BatchNorm2d(out_ch),
115             nn.ReLU(inplace=True),
116             nn.Conv2d(out_ch, out_ch, 3, padding=1),
```

```
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        x = self.up1(x)
        x = self.cbam1(x)
        x = self.conv1(x)

        x = self.up2(x)
        x = self.cbam2(x)
        x = self.conv2(x)

        x = self.up3(x)
        x = self.cbam3(x)
        x = self.conv3(x)

        x = self.final(x)
        return x


# ============== 4. 注意力可视化工具 ==============
class AttentionVisualizer:
    """注意力权重可视化工具"""

    @staticmethod
    def visualize_channel_attention(model, input_tensor, layer_name):
        """可视化通道注意力权重"""
        activations = {}

        def hook_fn(module, input, output):
            if hasattr(module, 'channel_attention'):
                # 获取通道注意力权重
                attn = module.channel_attention(input[0])
                activations['channel_weights'] = attn.detach()

        # 注册hook
        for name, module in model.named_modules():
```

```
            if name == layer_name:
                module.register_forward_hook(hook_fn)

        # 前向传播
        with torch.no_grad():
            _ = model(input_tensor)

        return activations.get('channel_weights', None)

    @staticmethod
    def visualize_spatial_attention(model, input_tensor, layer_name):
        """可视化空间注意力图"""
        activations = {}

        def hook_fn(module, input, output):
            if hasattr(module, 'spatial_attention'):
                # 获取空间注意力图
                attn = module.spatial_attention(input[0])
                activations['spatial_map'] = attn.detach()

        # 注册hook
        for name, module in model.named_modules():
            if name == layer_name:
                module.register_forward_hook(hook_fn)

        # 前向传播
        with torch.no_grad():
            _ = model(input_tensor)

        return activations.get('spatial_map', None)


# 使用示例
if __name__ == "__main__":
    # 1. 图像分类
    print("=" * 50)
    print("图像分类应用")
    classifier = ImageClassifier_CBAM(num_classes=10)
```

```
    x = torch.randn(2, 3, 32, 32)
    out = classifier(x)
    print(f"输入: {x.shape}, 输出: {out.shape}")

    # 2. 目标检测
    print("\n" + "=" * 50)
    print("目标检测应用")
    detector_backbone = DetectionBackbone_CBAM()
    x = torch.randn(2, 3, 640, 640)
    features = detector_backbone(x)
    print(f"输入: {x.shape}")
    for i, feat in enumerate(features):
        print(f"特征{i+1}: {feat.shape}")

    # 3. 语义分割
    print("\n" + "=" * 50)
    print("语义分割应用")
    seg_decoder = SegmentationDecoder_CBAM(512, num_classes=21)
    x = torch.randn(2, 512, 28, 28)
    out = seg_decoder(x)
    print(f"输入: {x.shape}, 输出: {out.shape}")
```

Listing 5: 注意力机制在不同任务中的应用

A.6 性能基准测试 (benchmark.py)

```
import torch
import torch.nn as nn
import time
import numpy as np
from thop import profile, clever_format
from resnet_cbam import resnet50_cbam
from torchvision.models import resnet50

class AttentionBenchmark:
    """注意力机制性能基准测试工具"""

    @staticmethod
    def count_parameters(model):
```

```
        """统计模型参数数量"""
        return sum(p.numel() for p in model.parameters())

    @staticmethod
    def measure_inference_time(model, input_size=(1, 3, 224, 224),
                               num_iterations=100, warmup=10):
        """测量推理时间"""
        device = torch.device('cuda' if torch.cuda.is_available()
            else 'cpu')
        model = model.to(device)
        model.eval()

        # 创建测试输入
        x = torch.randn(input_size).to(device)

        # 预热
        with torch.no_grad():
            for _ in range(warmup):
                _ = model(x)

        # 同步GPU
        if torch.cuda.is_available():
            torch.cuda.synchronize()

        # 测量时间
        times = []
        with torch.no_grad():
            for _ in range(num_iterations):
                start = time.time()
                _ = model(x)

                if torch.cuda.is_available():
                    torch.cuda.synchronize()

                times.append(time.time() - start)

        times = np.array(times)
        return {
            'mean': times.mean() * 1000,  # ms
```

```
            'std': times.std() * 1000,
            'min': times.min() * 1000,
            'max': times.max() * 1000
        }

    @staticmethod
    def measure_flops(model, input_size=(1, 3, 224, 224)):
        """测量FLOPs"""
        x = torch.randn(input_size)
        flops, params = profile(model, inputs=(x,), verbose=False)
        flops, params = clever_format([flops, params], "%.3f")
        return {'FLOPs': flops, 'Params': params}

    @staticmethod
    def compare_models(models_dict, input_size=(1, 3, 224, 224)):
        """比较多个模型的性能"""
        results = {}

        for name, model in models_dict.items():
            print(f"\n测试模型: {name}")
            print("-" * 50)

            # 参数数量
            params = AttentionBenchmark.count_parameters(model)
            print(f"参数数量: {params:,}")

            # FLOPs
            flops_info = AttentionBenchmark.measure_flops(model,
                input_size)
            print(f"FLOPs: {flops_info['FLOPs']}")

            # 推理时间
            time_info = AttentionBenchmark.measure_inference_time(
                model, input_size
            )
            print(f"推理时间: {time_info['mean']:.2f} +/- "
                  f"{time_info['std']:.2f} ms")

            results[name] = {
```

```
                'params': params,
                'flops': flops_info['FLOPs'],
                'inference_time_ms': time_info['mean'],
                'inference_std_ms': time_info['std']
            }

        return results


# 使用示例
if __name__ == "__main__":
    print("=" * 60)
    print("注意力机制性能基准测试")
    print("=" * 60)

    # 准备模型
    models = {
        'ResNet-50': resnet50(pretrained=False),
        'ResNet-50-CBAM': resnet50_cbam(num_classes=1000),
    }

    # 运行基准测试
    results = AttentionBenchmark.compare_models(
        models,
        input_size=(1, 3, 224, 224)
    )

    # 打印对比结果
    print("\n" + "=" * 60)
    print("性能对比总结")
    print("=" * 60)

    baseline_params = results['ResNet-50']['params']
    baseline_time = results['ResNet-50']['inference_time_ms']

    for name, metrics in results.items():
        print(f"\n{name}:")
        print(f"  参数数量: {metrics['params']:,} "
              f"({metrics['params']/baseline_params:.2%})")
```

```
        print(f"  FLOPs: {metrics['flops']}")
        print(f"  推理时间: {metrics['inference_time_ms']:.2f} ms "
              f"({metrics['inference_time_ms']/baseline_time:.2%})")
```

Listing 6: 注意力机制性能基准测试

A.7 训练脚本示例 (train_cbam.py)

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from resnet_cbam import resnet50_cbam
import wandb  # 可选: 用于实验跟踪


def train_one_epoch(model, train_loader, criterion, optimizer,
                    device, epoch):
    """训练一个epoch"""
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for batch_idx, (inputs, targets) in enumerate(train_loader):
        inputs, targets = inputs.to(device), targets.to(device)

        # 前向传播
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)

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

        # 统计
        running_loss += loss.item()
```

```
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

        if batch_idx % 100 == 0:
            print(f'Epoch: {epoch} | Batch: {batch_idx}/{len(
                train_loader)} '
                  f'| Loss: {running_loss/(batch_idx+1):.3f} '
                  f'| Acc: {100.*correct/total:.2f}%')

    return running_loss / len(train_loader), 100. * correct / total


def validate(model, val_loader, criterion, device):
    """验证"""
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)

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

    return val_loss / len(val_loader), 100. * correct / total


def main():
    # 设置
    device = torch.device('cuda' if torch.cuda.is_available() else '
        cpu')
    num_epochs = 100
```

```
    batch_size = 128
    learning_rate = 0.1

    # 数据预处理
    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)),
    ])

    # 数据加载
    train_dataset = datasets.CIFAR10(root='./data', train=True,
                                     download=True,
                                     transform=transform_train)
    test_dataset = datasets.CIFAR10(root='./data', train=False,
                                    download=True,
                                    transform=transform_test)

    train_loader = DataLoader(train_dataset, batch_size=batch_size,
                              shuffle=True, num_workers=4)
    test_loader = DataLoader(test_dataset, batch_size=batch_size,
                             shuffle=False, num_workers=4)

    # 模型
    model = resnet50_cbam(num_classes=10).to(device)

    # 损失函数和优化器
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                          momentum=0.9, weight_decay=5e-4)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
```

```
                                                    T_max=
                                                        num_epochs)

    # 训练循环
    best_acc = 0.0
    for epoch in range(num_epochs):
        print(f'\nEpoch: {epoch+1}/{num_epochs}')

        # 训练
        train_loss, train_acc = train_one_epoch(
            model, train_loader, criterion, optimizer, device, epoch
        )

        # 验证
        val_loss, val_acc = validate(model, test_loader, criterion,
            device)

        # 学习率调度
        scheduler.step()

        print(f'Train Loss: {train_loss:.3f} | Train Acc: {train_acc
            :.2f}%')
        print(f'Val Loss: {val_loss:.3f} | Val Acc: {val_acc:.2f}%')

        # 保存最佳模型
        if val_acc > best_acc:
            best_acc = val_acc
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'accuracy': best_acc,
            }, 'best_cbam_model.pth')
            print(f'Saved best model with accuracy: {best_acc:.2f}%'
                )

    print(f'\nTraining completed! Best accuracy: {best_acc:.2f}%')
```

```
143 if __name__ == "__main__":
144     main()
```

Listing 7: 使用 CBAM 的完整训练脚本