

# CS231n：从线性分类器到扩散模型

Lin · 2026 年 2 月

# 目录

1 为什么要从零手写? .....	4
1.1 挑战背景 .....	4
1.2 为什么不直接用 PyTorch? .....	4
1.3 学习路线总览 .....	4
2 线性分类器：当一切还很简单 .....	5
2.1 数据预处理：为什么输入不能“裸”进网络? .....	5
2.2 kNN——理解距离的直觉 .....	5
2.3 SVM——Margin 的几何意义 .....	6
2.3.1 Hinge Loss 推导 .....	6
2.3.2 梯度推导 .....	6
2.3.3 向量化实现 .....	6
2.4 Softmax——从打分到概率 .....	7
2.4.1 公式体系 .....	7
2.4.2 数值稳定性：exp 溢出是 Softmax 的经典数值灾难 .....	7
2.4.3 梯度推导 .....	7
3 两层神经网络：第一次真正理解反向传播 .....	9
3.1 网络结构 .....	9
3.2 计算图与链式法则 .....	9
3.3 梯度检查 .....	9
3.4 权重初始化 .....	10
4 深层网络与正则化技术 .....	11
4.0.1 梯度消失/爆炸的结构性根源 .....	11
4.1 激活函数的演进 .....	11
4.2 BatchNorm——最难的反向传播 .....	12
4.2.1 前向传播 .....	12
4.2.2 反向传播推导 .....	12
4.3 LayerNorm .....	13
4.4 Dropout——随机正则化 .....	13
4.4.1 为什么 Dropout 有效？——防止神经元“共谋” .....	13
4.4.2 过拟合与正则化的全景 .....	13
4.5 任意深度全连接网络 .....	14
5 卷积神经网络：从局部感受野到全局语义 .....	15
5.1 卷积层 .....	15
5.1.1 前向传播 .....	15
5.1.2 Padding 与 Stride 的工程权衡 .....	15
5.1.3 感受野（Receptive Field）与网络深度 .....	15
5.1.4 im2col 原理 .....	16
5.1.5 反向传播 .....	16
5.2 池化层 .....	17
5.3 三层卷积网络 .....	17

6 优化器：让网络真正学起来 .....	19
6.1 SGD .....	19
6.2 SGD with Momentum .....	19
6.3 RMSProp .....	19
6.4 Adam .....	19
6.4.1 优化器对比 .....	19
6.5 训练技巧：避坑指南 .....	20
6.5.1 学习率衰减 (Learning Rate Decay) .....	20
6.5.2 Batch Size 对梯度噪声的影响 .....	20
7 序列模型：从 RNN 到 Transformer .....	21
7.1 Vanilla RNN .....	21
7.2 LSTM .....	21
7.3 Image Captioning .....	21
7.4 Transformer .....	21
7.4.1 位置编码 .....	22
7.4.2 Multi-Head Attention .....	22
7.4.3 Decoder Layer .....	22
7.4.4 Vision Transformer (ViT) .....	23
7.4.5 Scaling Law：为什么“简单架构 + 海量数据”能产生智能 .....	23
8 迁移学习：站在巨人的肩膀上 .....	24
8.1 为什么要用预训练模型？ .....	24
9 自监督学习与基础模型 .....	25
9.1 SimCLR 对比学习 .....	25
9.2 CLIP .....	25
9.3 DINO 视频分割 .....	25
10 扩散模型：从噪声中生成图像 .....	26
10.1 DDPM 理论框架 .....	26
10.2 U-Net 架构 .....	26
10.3 噪声调度 .....	27

# 1 为什么要从零手写?

## 1.1 挑战背景

CS231n 是斯坦福大学计算机视觉方向最经典的课程之一。它的作业设计有一个显著特点：要求从 NumPy 级别手写前向传播和反向传播开始，而不是调用 PyTorch 的 `autograd`。

这意味着你不能把反向传播当作黑箱——你必须亲手推导每一层的梯度公式，亲手处理 shape 对齐、数值稳定性、广播机制等工程细节。

我用两周时间完成了全部三个 Assignment 的核心代码：

- **Assignment 1:** kNN、SVM、Softmax、两层神经网络
- **Assignment 2:** 全连接网络（任意深度）、BatchNorm、Dropout、卷积网络、RNN/LSTM Captioning
- **Assignment 3:** Transformer Captioning、Vision Transformer、SimCLR 自监督学习、CLIP/DINO、DDPM 扩散模型

## 1.2 为什么不直接用 PyTorch?

PyTorch 的 `autograd` 是一个精心设计的计算图引擎。笔者之前也是一直调包，但正因为它太好用了，很容易陷入一种“会调 API 但不懂原理”的状态。手写实现的意义在于：

1. 理解梯度流动：当你手写 `batchnorm_backward` 时，你才会真正理解为什么 BatchNorm 的反向传播公式如此复杂——它涉及对均值和方差的链式求导，而这两者本身是输入的函数。
2. 理解数值稳定性：Softmax 的 log-sum-exp 技巧、BatchNorm 中除以  $\sqrt{\sigma^2 + \epsilon}$  而不是  $\sigma$ ——这些细节在框架中被隐藏，但在手写时会自然暴露。
3. 理解向量化：从 two-loop 到 no-loop 的 NumPy 实现，本质是理解矩阵运算如何替代逐元素操作。

## 1.3 学习路线总览

整个实现路径可以概括为一条从简单到复杂的递进链：

阶段	核心内容
线性分类器	kNN → SVM → Softmax → 梯度推导
浅层网络	两层网络 → 反向传播 → 梯度检查
深层网络	任意深度全连接 → BN / LN → Dropout
卷积网络	Conv → Pool → Spatial BN → 三层 CNN
序列模型	RNN → LSTM → Image Captioning
注意力机制	Transformer Decoder → ViT
自监督学习	SimCLR 对比学习 → CLIP → DINO
生成模型	DDPM → U-Net → Classifier-Free Guidance

## 2 线性分类器：当一切还很简单

### 2.1 数据预处理：为什么输入不能“裸”进网络？

在进入任何分类器之前，有一个看似平凡却极为关键的步骤：数据预处理。CS231n 的作业中，CIFAR-10 图像在送入模型前必须做均值减除（mean subtraction），有时还会做归一化（normalization）。

为什么这一步不可省略？原始像素值范围是 [0, 255]，不同通道的量纲和分布差异巨大。如果不做中心化，损失函数的等高线会呈现极度扁长的椭圆形——梯度下降将在短轴方向剧烈振荡，在长轴方向却龟速爬行。中心化后等高线趋近圆形，SGD 的每一步都朝着最优解更直接地前进。

```
# CS231n 标准预处理流程
mean_image = np.mean(X_train, axis=0)      # 计算训练集均值图像
X_train -= mean_image                      # 中心化：让数据分布以零为中心
X_test -= mean_image                       # 测试集必须用训练集的均值！
```

工程直觉：归一化不仅加速收敛，还能改善数值稳定性。当特征量级相差数个数量级时（如一个特征范围 [0, 1]，另一个 [0, 10000]），学习率的选择变得极其困难——对一个特征合适的学习率对另一个可能太大或太小。

### 2.2 kNN——理解距离的直觉

k-近邻（kNN）是最朴素的分类器：不学习任何参数，仅在预测时计算测试样本与所有训练样本的距离，取最近的  $k$  个邻居投票。

L2 距离公式：

$$d(x_i, x_j) = \sqrt{\sum_{d=1}^D (x_i^{(d)} - x_j^{(d)})^2} = \|x_i - x_j\|_2 \quad (1)$$

向量化实现的核心思路：

朴素实现需要两层循环，时间复杂度  $O(N_{\text{test}} \times N_{\text{train}} \times D)$ 。一个关键的代数恒等式可以消除所有显式循环：

$$\|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2a^\top b \quad (2)$$

这将距离计算转化为三个矩阵运算：

```
# 无循环向量化实现
test_sq = np.sum(X ** 2, axis=1)          # (num_test,)
train_sq = np.sum(X_train ** 2, axis=1)     # (num_train,)
inner = X.dot(X_train.T)                   # (num_test, num_train)
dists = np.sqrt(test_sq[:, None] + train_sq[None, :] - 2 * inner)
```

为什么 kNN 无法 scale？

预测时间复杂度为  $O(N_{\text{train}})$ ——每增加一个训练样本，预测就变慢一点。它没有“学习”的概念，不能将数据压缩为一组参数。这引出了参数化模型的动机。

## 2.3 SVM——Margin 的几何意义

支持向量机 (SVM) 是第一个真正的参数化分类器。它引入了一个核心概念：间隔 (margin)。

### 2.3.1 Hinge Loss 推导

给定权重矩阵  $W \in \mathbb{R}^{D \times C}$ , 对输入  $x_i$  计算得分  $s = x_i W$ , 其中  $s_j$  是第  $j$  类的得分。SVM 的多类 hinge loss 定义为：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta) \quad (3)$$

其中  $\Delta = 1$  是间隔超参数。几何含义清晰：正确类的得分必须比任何错误类的得分至少高出  $\Delta$ , 否则产生损失。

加上 L2 正则化后，总损失为：

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda \|W\|^2 \quad (4)$$

正则化项的意义是防止权重过大——它倾向于让权重分散而不是集中，从而提高泛化能力。

### 2.3.2 梯度推导

对式 3 求关于  $W$  的梯度。定义指示函数  $\mathbb{1}[s_j - s_{y_i} + \Delta > 0]$ , 则：

对于  $j \neq y_i$  的列：

$$\frac{\partial L_i}{\partial W_{\cdot,j}} = \mathbb{1}[s_j - s_{y_i} + \Delta > 0] \cdot x_i \quad (5)$$

对于正确类  $j = y_i$  的列：

$$\frac{\partial L_i}{\partial W_{\cdot,y_i}} = - \sum_{j \neq y_i} \mathbb{1}[s_j - s_{y_i} + \Delta > 0] \cdot x_i \quad (6)$$

直觉解释：每当某个错误类“侵入”间隔区域，正确类的权重列就被“推”向输入方向，而错误类的权重列被“推”离输入方向。

### 2.3.3 向量化实现

```
def svm_loss_vectorized(W, X, y, reg):
    num_train = X.shape[0]
    scores = X.dot(W)
    correct_scores = scores[np.arange(num_train), y][:, None]
    margins = np.maximum(0, scores - correct_scores + 1.0)
    margins[np.arange(num_train), y] = 0.0

    loss = np.sum(margins) / num_train + reg * np.sum(W * W)

    mask = (margins > 0).astype(np.float64)
    count = np.sum(mask, axis=1)
    mask[np.arange(num_train), y] = -count
    dW = X.T.dot(mask) / num_train + 2.0 * reg * W
    return loss, dW
```

核心技巧：构造 mask 矩阵，正确类位置填入负的违规计数，从而用一次矩阵乘法完成整个梯度计算。

## 2.4 Softmax——从打分到概率

SVM 的得分没有概率解释。Softmax 将原始得分转化为归一化概率分布，配合交叉熵损失使用。

### 2.4.1 公式体系

Softmax 函数：

$$P(y = k \mid x_i) = \frac{e^{f_k}}{\sum_{j=1}^C e^{f_j}} \quad (7)$$

其中  $f = x_i W$  是得分向量。交叉熵损失：

$$L_i = -\log P(y = y_i \mid x_i) = -f_{y_i} + \log \sum_{j=1}^C e^{f_j} \quad (8)$$

### 2.4.2 数值稳定性：exp 溢出是 Softmax 的经典数值灾难

直接计算式 7 时，若得分向量中存在较大值（如  $f_j = 1000$ ），`np.exp(1000)` 直接返回 `inf`。后续除法产生 `inf / inf = nan`，这个 `nan` 会沿计算图扩散，污染整个梯度张量，训练立即崩溃。

崩溃链路：`scores` 含大值  $\rightarrow \exp(\text{scores})$  溢出为 `inf`  $\rightarrow \text{inf} / \text{sum}(\text{inf}) = \text{nan} \rightarrow \text{loss} = \text{nan}$   $\rightarrow$  所有 `dW = nan`  $\rightarrow$  权重全部变 `nan`。

解决方案 — log-sum-exp 平移不变性：

$$P(y = k \mid x_i) = \frac{e^{f_k - \max_j f_j}}{\sum_{j=1}^C e^{f_j - \max_j f_j}} \quad (9)$$

为什么平移不改变结果？分子分母同时乘以  $e^{-\max_j f_j}$ ，完全等价。而减去最大值后，指数运算的最大输入为 0， $\exp(0) = 1$ ，彻底消除溢出风险。代码里只需一行：

```
scores -= np.max(scores, axis=1, keepdims=True) # 必须在 exp 之前
```

这行代码的缺失是 Softmax 实现中最高频的致命 Bug。

### 2.4.3 梯度推导

对得分  $f_k$  求梯度（省略样本下标  $i$ ）：

$$\frac{\partial L}{\partial f_k} = \begin{cases} P(y = k) - 1 & \text{if } k = y_i \\ P(y = k) & \text{if } k \neq y_i \end{cases} \quad (10)$$

统一写法： $\frac{\partial L}{\partial f} = p - e_{y_i}$ ，其中  $p$  是 softmax 概率向量， $e_{y_i}$  是 one-hot 向量。链式法则回传到权重： $\frac{\partial L}{\partial W} = x^\top \cdot \frac{\partial L}{\partial f}$ 。

```
def softmax_loss_vectorized(W, X, y, reg):
    num_train = X.shape[0]
    scores = X.dot(W)
    scores -= np.max(scores, axis=1, keepdims=True) # 数值稳定性：必须在 exp 前执行
    exp_scores = np.exp(scores)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    loss = -np.sum(np.log(probs[np.arange(num_train), y])) / num_train
```

```
loss += reg * np.sum(W * W)

dscores = probs.copy()
dscores[np.arange(num_train), y] -= 1
dW = X.T.dot(dscores) / num_train + 2.0 * reg * W
return loss, dW
```

### Debug 手册: 线性分类器

- Loss 正常但准确率停滞: 忘记减均值 → 损失面畸形 → 优化路径低效
- 分类器退化为随机猜测: 正则化系数过大, 权重被压到接近零
- 训练直接崩溃出 `nan`: `exp` 前未减最大值, 触发上述溢出链路。修复: 加一行 `scores -= max`

## 3 两层神经网络：第一次真正理解反向传播

### 3.1 网络结构

两层网络的前向传播可以写成：

$$h = \max(0, xW_1 + b_1) \quad s = hW_2 + b_2 \quad (11)$$

其中  $\max(0, \cdot)$  是 ReLU 激活函数。这是最简单的非线性模型，但它已经足够让我们深入理解反向传播。

### 3.2 计算图与链式法则

反向传播的本质是链式法则在计算图上的系统化应用。对于损失  $L$ ：

$$L = \text{softmax\_loss}(s, y) + \lambda(\|W_1\|^2 + \|W_2\|^2) \quad (12)$$

反向传播的梯度流动路径：

$$L \rightarrow s \rightarrow h \rightarrow (W_1, b_1) \quad (13)$$

第一步： $\partial L / \partial s$  由 softmax loss 直接给出（即  $p - e_y$ ）。

第二步： $\frac{\partial L}{\partial W_2} = h^\top \cdot \frac{\partial L}{\partial s}$ ,  $\frac{\partial L}{\partial b_2} = \sum_i \frac{\partial L}{\partial s_i}$ 。

第三步： $\frac{\partial L}{\partial h} = \frac{\partial L}{\partial s} \cdot W_2^\top$ 。

第四步：ReLU 的梯度是一个门控操作——对于  $h_j > 0$  的位置，梯度直接通过；对于  $h_j \leq 0$  的位置，梯度被截断为零。

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial h} \odot \mathbb{1}[a > 0] \quad (14)$$

其中  $a = xW_1 + b_1$  是 ReLU 的输入。

第五步： $\frac{\partial L}{\partial W_1} = x^\top \cdot \frac{\partial L}{\partial a}$ 。

```
# affine_backward 的核心
x_reshaped = x.reshape(N, -1)
dw = x_reshaped.T.dot(dout)
db = np.sum(dout, axis=0)
dx = dout.dot(w.T).reshape(x.shape)

# relu_backward 的核心
dx = dout * (x > 0)
```

### 3.3 梯度检查

数值梯度  $\frac{\partial L}{\partial \theta_i} \approx \frac{L(\theta_i + \varepsilon) - L(\theta_i - \varepsilon)}{2\varepsilon}$  是验证解析梯度正确性的黄金标准。相对误差应小于  $10^{-5}$ ：

$$\text{rel\_error} = \frac{\|g_{\text{analytic}} - g_{\text{numeric}}\|}{\|g_{\text{analytic}}\| + \|g_{\text{numeric}}\|} \quad (15)$$

反向传播 = 链式法则 + 前向 Cache 的复用。每个节点在前向时缓存输入，反向时用缓存计算局部雅可比，再与上游梯度做矩阵乘法。前向传播是“执行计算”，反向传播是“追溯责

任”——量化每个参数对 Loss 的贡献。整个过程没有任何启发式成分，纯粹是微积分链式法则的机械化实现。

### 3.4 权重初始化

训练能否启动，初始化是第一道门槛。

全零初始化 → 对称性灾难：所有权重相同时，同层神经元的前向输出、反向梯度、更新量完全相同。网络退化为单个有效神经元，无论宽度多大。

Xavier 初始化 (Sigmoid/Tanh)： $W \sim \mathcal{N}(0, 1/n_{in})$ ，保证每层输出方差不逐层放大或缩小。

Kaiming (He) 初始化 (ReLU)：ReLU 将约半数激活置零，方差减半。补偿方差为  $2/n_{in}$ ：

$$W \sim \mathcal{N}(0, \sqrt{2/n_{in}}) \quad (16)$$

```
# He 初始化 : ReLU 网络的标准选择
W = np.random.randn(n_in, n_out) * np.sqrt(2.0 / n_in)
```

#### Debug 手册：初始化故障

- 训练精度卡在 10% (CIFAR-10 随机水平)：初始化标准差过大（如 `weight_scale=1.0`），前向传播时激活值被推入极端区间，ReLU 大面积“死亡”——输出恒零、梯度恒零。修复：改用 He 初始化或 `weight_scale=1e-3`
- Loss 不动但不报错：全零初始化 → 对称性未打破 → 实质只有一个有效神经元
- 前几层梯度为零：忘记使用 He 初始化，ReLU 后方差逐层衰减至消失

## 4 深层网络与正则化技术

从两层网络到任意深度的全连接网络，架构变为：

$$\{\text{affine} \rightarrow [\text{BN/LN}] \rightarrow \text{ReLU} \rightarrow [\text{Dropout}]\} \times (L - 1) \rightarrow \text{affine} \rightarrow \text{softmax} \quad (17)$$

每一层独立实现 `forward / backward`，通过缓存（cache）连接前向和反向。

### 4.0.1 梯度消失/爆炸的结构性根源

深层网络的核心工程难题：反向传播时，梯度需要连乘每层的局部雅可比矩阵。设第  $l$  层的局部梯度为  $J_l$ ，则损失对第 1 层参数的梯度形如：

$$\frac{\partial L}{\partial W_1} \propto J_L \cdot J_{L-1} \cdot \dots \cdot J_2 \cdot J_1 \quad (18)$$

- 若  $\|J_l\| < 1$ （每层梯度衰减）： $L$  层连乘后梯度指数衰减 → 梯度消失，前几层参数几乎不更新
- 若  $\|J_l\| > 1$ （每层梯度放大）： $L$  层连乘后梯度指数爆炸 → 梯度爆炸，参数更新量级失控

这不是优化器能解决的问题——它是网络结构本身的缺陷。结构性的解决方案：

- **BatchNorm**：将每层激活归一化到标准分布，强制  $\|J_l\| \approx 1$ ，从源头稳定梯度幅度
- 残差连接（**ResNet**）： $h_l = F(h_{l-1}) + h_{l-1}$ ，梯度有一条“高速公路”直通底层，绕过连乘瓶颈
- **LSTM** 门控：遗忘门  $f \approx 1$  时梯度无衰减流过，等效于序列维度上的残差连接

## 4.1 激活函数的演进

激活函数直接决定梯度能否健康流过深层网络。

**Sigmoid** 的致命缺陷： $\sigma(x) = 1/(1 + e^{-x})$ ，输出范围  $(0, 1)$ 。饱和区导数趋近零 → 深层网络梯度消失。输出不以零为中心 → 下一层权重梯度全正或全负 → zig-zag 更新路径。

**ReLU**： $\text{ReLU}(x) = \max(0, x)$ ，正区间导数恒为 1，彻底解决正半轴饱和。代价是“死亡 ReLU”：神经元输入持续为负 → 输出恒零 → 梯度恒零 → 永远无法恢复。

**GELU**： $\text{GELU}(x) = x \cdot \Phi(x)$  ( $\Phi$  为标准正态 CDF)，ViT/GPT/BERT 的默认激活。零点附近平滑可导，对小负值施加概率性门控而非硬截断，避免死神经元。

激活函数	饱和问题	零中心	适用场景
Sigmoid	严重	✗	二分类输出层，隐藏层已淘汰
Tanh	严重	✓	RNN 隐藏状态
ReLU	负半轴	✗	CNN、全连接网络默认
GELU	无	近似✓	Transformer、ViT 等现代架构

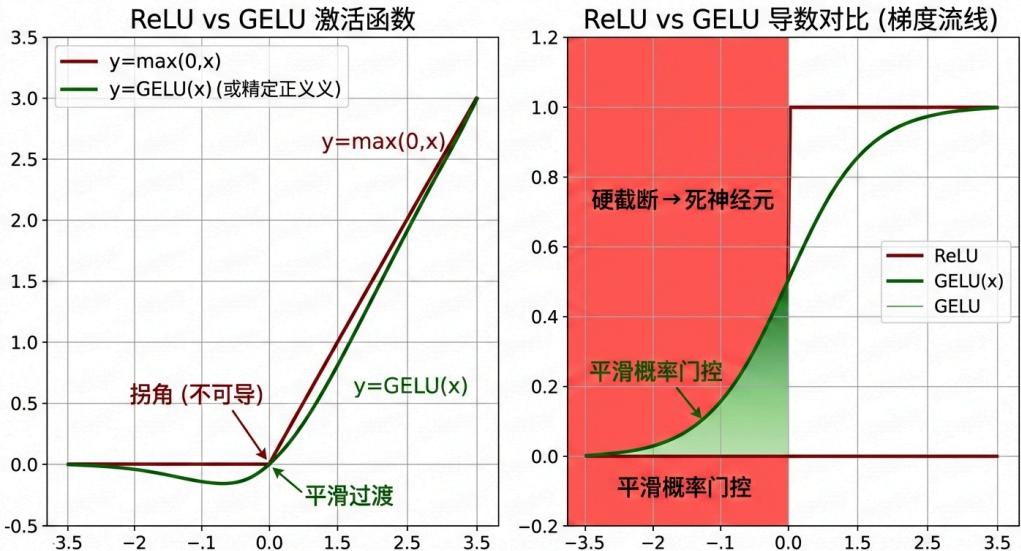


图 1 ReLU vs GELU: 函数曲线及导数曲线对比。ReLU 在  $x < 0$  区域导数恒为零 (硬截断 → 死神经元), GELU 平滑过渡避免梯度完全截断。

## 4.2 BatchNorm——最难的反向传播

### 4.2.1 前向传播

Batch Normalization 的前向公式分为三步:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i, \quad \sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2 \quad (19)$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad (20)$$

$$y_i = \gamma \hat{x}_i + \beta \quad (21)$$

训练时用 mini-batch 统计量, 测试时用指数移动平均的 running mean/var。

### 4.2.2 反向传播推导

BN backward 是 CS231n 中最容易写错的部分。根因:  $x_i$  同时出现在  $\hat{x}_i$  的分子 (直接路径)、 $\mu$  (均值路径)、 $\sigma^2$  (方差路径) 中。手推时漏掉任何一条路径, 梯度检查会显示  $10^{-3}$  量级的相对误差——看起来“差不多对”, 实际训练不收敛。

从输出端开始:  $\frac{\partial L}{\partial \gamma} = \sum_i \frac{\partial L}{\partial y_i} \cdot \hat{x}_i$ ,  $\frac{\partial L}{\partial \beta} = \sum_i \frac{\partial L}{\partial y_i}$ 。

对  $\frac{\partial L}{\partial x_i}$  的三条路径合并后的闭式公式:

$$\frac{\partial L}{\partial x_i} = \frac{\gamma}{N\sqrt{\sigma^2 + \epsilon}} \left( N d_i - \sum_j d_j - \hat{x}_i \sum_j d_j \hat{x}_j \right) \quad (22)$$

```
std_inv = 1. / np.sqrt(sample_var + eps)
dx = (gamma * std_inv / N) * (
    N * dout -
    np.sum(dout, axis=0) -
    x_norm * np.sum(dout * x_norm, axis=0)
)
```

直觉：最终梯度是三项的组合——原始梯度、减去均值梯度的贡献、减去方差梯度的贡献。这保证了梯度也是“零均值”的。

#### Debug 手册：BatchNorm

- **Inference 精度骤降**: 忘记在测试时切换到 `running_mean / running_var`。训练时用 mini-batch 统计量，而 Inference 时输入可能只有一张图——用单张图的统计量归一化完全无意义。这是 BN 最高频的致命 Bug
- **batch size 过小**: `batch_size=1` 时 mini-batch 统计量就是单个样本本身，BN 退化为恒等映射。改用 LayerNorm 或 GroupNorm
- 梯度检查“看起来差不多对”( $10^{-3}$  量级)： $\sigma^2$  的梯度回传有三条路径，漏掉任何一条都会导致微妙的梯度偏差。使用上面的闭式公式可以避免逐节点反向传播时的遗漏

### 4.3 LayerNorm

Layer Normalization 与 BatchNorm 的区别在于归一化维度：BN 沿 batch 维度归一化（每个特征），LN 沿特征维度归一化（每个样本）。实现上，只需将输入转置后复用 BN 代码。

LN 的优势在测试时不需要 running statistics，更适合序列模型。

### 4.4 Dropout——随机正则化

Inverted Dropout 的核心思想：训练时以概率  $p$  保留每个神经元，并除以  $p$  进行缩放，使得测试时无需任何修改。

$$\text{train : } h_{\text{drop}} = h \odot m/p, \quad m_i \sim \text{Bernoulli}(p) \quad (23)$$

$$\text{test : } h_{\text{drop}} = h \quad (24)$$

反向传播极其简单： $dx = d_{\text{out}} \odot m/p$ ——梯度只流过训练时“存活”的路径。

#### 4.4.1 为什么 Dropout 有效？——防止神经元“共谋”

从数学角度，Dropout 可以被理解为同时训练  $2^n$  个子网络的集成。但更直觉的解释是打破共适应（co-adaptation）：

在没有 Dropout 的网络中，某些神经元可能学会“依赖”特定的上游神经元——它们形成固定的合作关系，联合编码某种特征。这种共谋关系使得每个神经元单独的表征能力很弱，一旦合作伙伴出错，整个链路就会失败。

Dropout 强制每个神经元独立地学习有用的特征——因为它无法预测训练时哪些“队友”会在场。这迫使网络学习更加鲁棒、冗余的内部表示。

#### 4.4.2 过拟合与正则化的全景

正则化的本质是在模型的“表达能力”和“泛化能力”之间寻找平衡。CS231n 中出现的正则化手段可以统一理解：

- **L2 正则化**: 惩罚大权重，倾向于让权重分散 → 等价于对权重施加高斯先验
- **Dropout**: 随机丢弃神经元 → 等价于隐式模型集成
- **BatchNorm**: 归一化激活值 → 隐式地限制了每层激活分布漂移，降低了对初始化和学习率的敏感度
- **数据增强**: 增加训练数据的多样性 → 最直接的正则化

判断过拟合的最简单信号：训练精度  $\gg$  验证精度。当你看到训练 Loss 持续下降但验证 Loss 开始上升时，正则化手段则是必须要采用的了。

## 4.5 任意深度全连接网络

FullyConnectedNet 的实现核心是循环化前向/反向传播：

```
# Forward: 逐层推进
for layer_idx in range(1, self.num_layers):
    out, cache = affine_forward(input, W, b)
    if normalization: out, bn_cache = batchnorm_forward(out, gamma, beta, bn_param)
    out, relu_cache = relu_forward(out)
    if dropout: out, do_cache = dropout_forward(out, dropout_param)

# Backward: 逆序回传
for layer_idx in range(self.num_layers - 1, 0, -1):
    if dropout: dx = dropout_backward(dx, do_cache)
    dx = relu_backward(dx, relu_cache)
    if normalization: dx, dgamma, dbeta = batchnorm_backward(dx, bn_cache)
    dx, dW, db = affine_backward(dx, cache)
    grads[f"W{layer_idx}"] = dW + reg * W
```

## 5 卷积神经网络：从局部感受野到全局语义

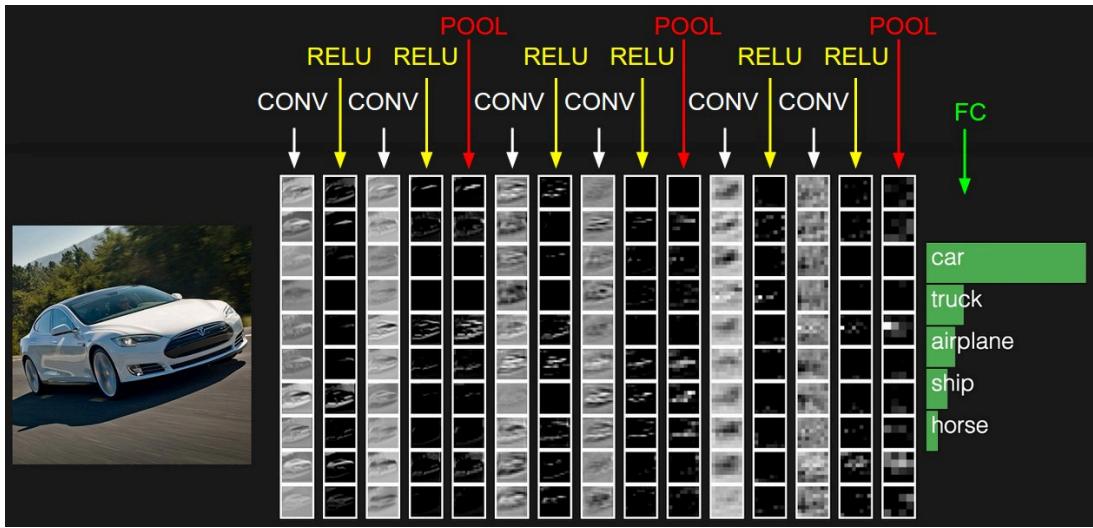


图 2 卷积神经网络典型架构：输入图像经过多层 Conv → Pool 提取特征，最终通过全连接层输出分类结果。

### 5.1 卷积层

#### 5.1.1 前向传播

卷积的数学定义：对输入  $x \in \mathbb{R}^{N \times C \times H \times W}$ ，滤波器  $w \in \mathbb{R}^{F \times C \times H_f \times W_f}$ ，输出为：

$$\text{out}[n, f, i, j] = \sum_{c=0}^{C-1} \sum_{h=0}^{H_f-1} \sum_{w'=0}^{W_f-1} x[n, c, i \cdot s + h, j \cdot s + w'] \cdot w[f, c, h, w'] + b[f] \quad (25)$$

输出尺寸公式：

$$H' = \frac{H - H_f + 2P}{S} + 1, \quad W' = \frac{W - W_f + 2P}{S} + 1 \quad (26)$$

#### 5.1.2 Padding 与 Stride 的工程权衡

**Same Padding** ( $P = \frac{H_f-1}{2}$ )：保持输出空间分辨率与输入相同 ( $H' = H$ )。在深层网络中至关重要——没有 Padding 的话，每过一层卷积空间尺寸缩小，几层之后特征图缩到无法操作。

**Stride** 作为下采样替代：传统做法是 Conv( $S = 1$ ) + Pooling 实现下采样。现代架构（如 ResNet）直接用  $S = 2$  的卷积替代 Pooling，减少一次前向运算，同时让网络自己学习下采样方式，而非依赖 max/avg 这种手工规则。

#### 5.1.3 感受野 (Receptive Field) 与网络深度

为什么需要堆叠多层卷积？单层  $3 \times 3$  卷积的每个输出像素只能“看到”原图  $3 \times 3$  的区域。堆叠  $L$  层后，感受野递推为：

$$\text{RF}_l = \text{RF}_{l-1} + (K_l - 1) \times \prod_{i=1}^{l-1} S_i \quad (27)$$

以  $K = 3, S = 1$  为例：2 层  $\rightarrow \text{RF} = 5$ ，3 层  $\rightarrow \text{RF} = 7$ ，5 层  $\rightarrow \text{RF} = 11$ 。深层网络的单个特征图像素能“看到”原图的巨大区域，这是深层 CNN 能捕获全局语义信息的原因。

两个  $3 \times 3$  卷积的感受野等价于一个  $5 \times 5$  卷积，但参数量为  $2 \times 3^2 C^2 = 18C^2$ ，远小于  $5^2 C^2 = 25C^2$ 。这解释了 VGGNet “全部用  $3 \times 3$  卷积”的设计动机。

#### 5.1.4 im2col 原理

朴素实现需要六重循环。`im2col` 将每个感受野展开为一行，把卷积转化为矩阵乘法：

**im2col Shape 变换 (CIFAR-10 输入 +  $5 \times 5$  滤波器):**

**Step 1** — 输入展开:  $(N, 3, 32, 32) \rightarrow$  滑动  $5 \times 5$  窗口  $\rightarrow (N \times 28 \times 28; ; 3 \times 5 \times 5) = (N \times 784; ; 75)$

**Step 2** — 滤波器展平:  $(F, 3, 5, 5) \rightarrow$  reshape  $\rightarrow (F; ; 75)$

**Step 3** — 矩阵乘法:  $(N \times 784; ; 75) \times (75; ; F) = (N \times 784; ; F)$

**Step 4** — 还原空间:  $(N \times 784; ; F) \rightarrow$  reshape  $\rightarrow (N; ; F; ; 28; ; 28)$

六重循环被压缩为一次 `np.dot`。

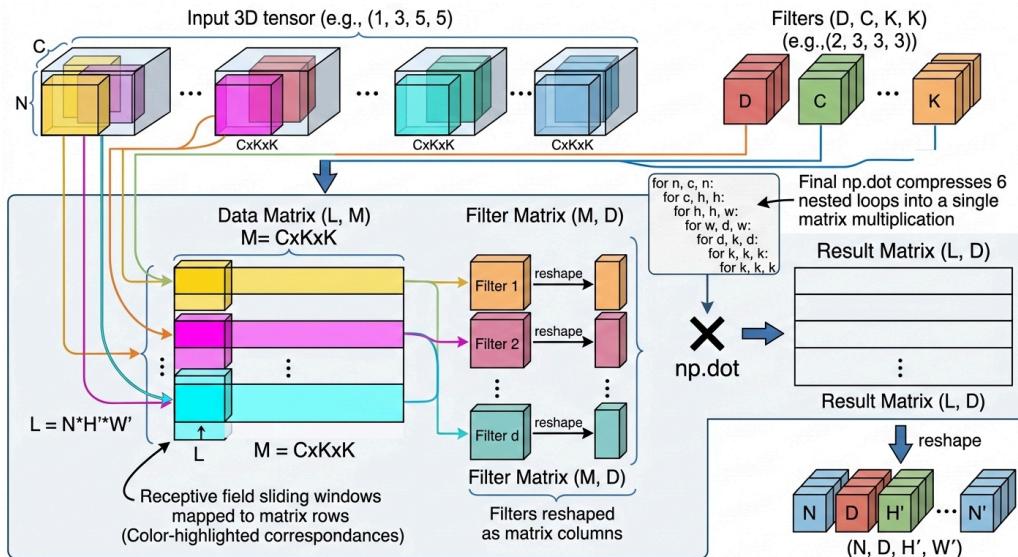


图 3 `im2col` 内存重排：3D 张量  $(N, C, H, W)$  的感受野滑动窗口被展平为 2D 矩阵的行，滤波器展平为列，卷积转化为一次矩阵乘法。

1. 输入  $(N, C, H, W) \rightarrow$  `im2col`  $\rightarrow$  列矩阵  $(N \cdot H' \cdot W', C \cdot H_f \cdot W_f)$
2. 滤波器  $(F, C, H_f, W_f) \rightarrow$  reshape  $\rightarrow (F, C \cdot H_f \cdot W_f)$
3. 输出 = 列矩阵  $\times$  滤波器  $\{\}^\top \rightarrow$  reshape  $\rightarrow (N, F, H', W')$

这是现代深度学习框架中卷积实现的标准方法。

#### 5.1.5 反向传播

卷积的反向传播本质是转置卷积 (transposed convolution)：

- $db$ : 对  $d_{\text{out}}$  沿空间维度求和
- $dw$ : 输入和  $d_{\text{out}}$  的互相关
- $dx$ :  $d_{\text{out}}$  和翻转滤波器的卷积

## 5.2 池化层

Max Pooling 的前向传播取窗口内最大值，反向传播将梯度路由到最大值位置，其余位置梯度为零。

```
# Max pool backward 的核心逻辑
for each pooling window:
    max_idx = np.argmax(window)
    dx[max_idx] += dout[output_position]
```

这意味着 Max Pooling 是一个“路由器”——它在前向传播时记住“谁是最大的”，反向传播时把梯度全部交给那个位置。

## 5.3 三层卷积网络

完整架构: conv → relu → 2×2 max pool → affine → relu → affine → softmax

```
# Forward
out_1, cache_1 = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
out_2, cache_2 = affine_relu_forward(out_1, W2, b2)
scores, cache_3 = affine_forward(out_2, W3, b3)

# Backward
loss, dscores = softmax_loss(scores, y)
dx3, dw3, db3 = affine_backward(dscores, cache_3)
dx2, dw2, db2 = affine_relu_backward(dx3, cache_2)
dx1, dw1, db1 = conv_relu_pool_backward(dx2, cache_1)
```

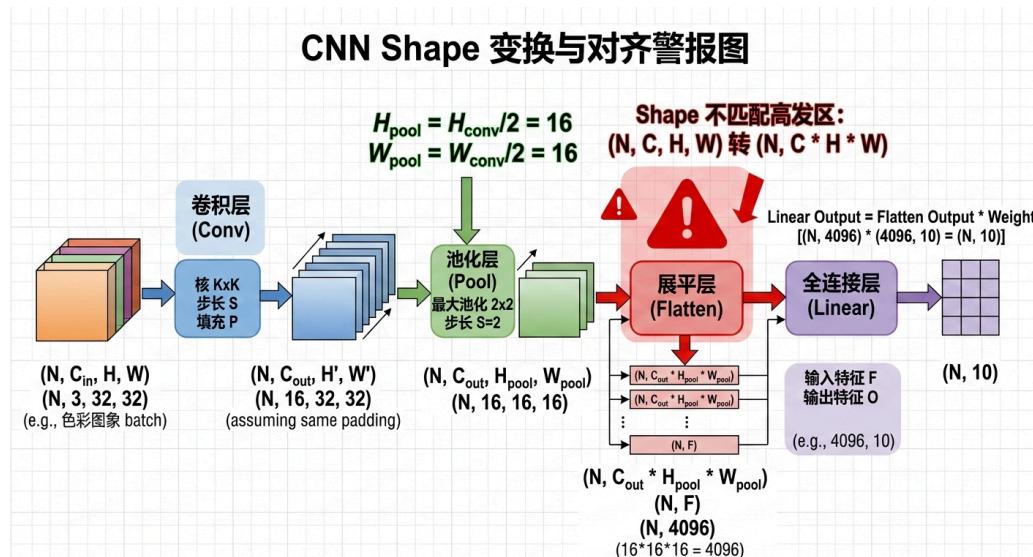


图 4 CNN Shape 变换流向: Conv → Pool → Flatten → Linear。Flatten 处是 Shape 不匹配高发区:  $(N, C, H, W)$  转  $(N, C \cdot H \cdot W)$ 。

### Debug 手册: CNN Shape 对齐

- ValueError: shapes (64,100) and (3072,100) not aligned:** 卷积层输出未 flatten，或忘记计算 pooling 后的空间尺寸。全连接层输入维度必须严格等于  $F \times H_{pool} \times W_{pool}$
- 深层 CNN 梯度消失:** 遗漏 Spatial BatchNorm。卷积层的 BN 沿  $(N, H, W)$  维度归一化，保留通道维  $C$

- 测试精度骤降: Spatial BN 的 `running_mean` / `running_var` 未正确更新

# 6 优化器：让网络真正学起来

## 6.1 SGD

最基础的优化规则：

$$w \leftarrow w - \eta \nabla L \quad (28)$$

问题：学习率固定，所有参数共享同一步长，在损失面上容易振荡。

## 6.2 SGD with Momentum

引入“速度”变量，模拟物理中的动量：

$$v_t = \mu v_{t-1} - \eta \nabla L, \quad w \leftarrow w + v_t \quad (29)$$

动量项使得梯度方向一致时加速，振荡时相互抵消，实际上相当于对梯度进行指数移动平均。

## 6.3 RMSProp

自适应学习率方法——用梯度平方的移动平均来归一化每个参数的学习率：

$$c_t = \rho c_{t-1} + (1 - \rho)(\nabla L)^2 \quad (30)$$

$$w \leftarrow w - \eta \cdot \frac{\nabla L}{\sqrt{c_t} + \epsilon} \quad (31)$$

梯度大的参数学习率自动缩小，梯度小的参数学习率自动放大。

## 6.4 Adam

结合了 Momentum 和 RMSProp 的优点：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla L \quad (\text{一阶矩估计}) \quad (32)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla L)^2 \quad (\text{二阶矩估计}) \quad (33)$$

偏置修正（因为  $m_0 = v_0 = 0$  导致初期估计偏低）：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (34)$$

更新规则：

$$w \leftarrow w - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \quad (35)$$

```
config["t"] += 1
config["m"] = beta1 * config["m"] + (1 - beta1) * dw
config["v"] = beta2 * config["v"] + (1 - beta2) * dw**2
m_hat = config["m"] / (1 - beta1 ** config["t"])
v_hat = config["v"] / (1 - beta2 ** config["t"])
next_w = w - lr * m_hat / (np.sqrt(v_hat) + epsilon)
```

### 6.4.1 优化器对比

优化器	自适应学习率	动量	特点
-----	--------	----	----

SGD	✗	✗	最简单，需精心调参
Momentum	✗	✓	加速收敛，减少振荡
RMSProp	✓	✗	自适应步长，适合稀疏梯度
Adam	✓	✓	默认首选，带偏置修正

## 6.5 训练技巧：避坑指南

### 6.5.1 学习率衰减 (Learning Rate Decay)

固定学习率几乎无法让网络达到最优。常见的策略是：训练初期用较大的学习率快速逼近最优区域，然后逐步降低学习率以精细调整。

常用调度方式：

- **Step Decay**: 每隔固定 epoch 将学习率乘以衰减因子（如 0.1）
- **Cosine Annealing**: 学习率按余弦曲线平滑衰减至接近零，然后可选地重启
- **Warm-up**: 训练最初几个 epoch 从极小学习率线性增长到目标值，避免初始不稳定的梯度导致参数“飞掉”

### 6.5.2 Batch Size 对梯度噪声的影响

Batch size 不仅影响训练速度，还深刻地影响优化的质量：

- 小 batch (如 32)：梯度估计噪声大，但噪声本身具有正则化效果——它帮助优化器跳出尖锐的局部极小值，找到更平坦、泛化更好的解
- 大 batch (如 1024+)：梯度估计准确，收敛路径平滑，但容易收敛到尖锐极小值，泛化能力较差。通常需要配合学习率线性缩放规则 (linear scaling rule)

工程经验：在实践中，Adam + Cosine Annealing + Warm-up 是最常用的组合。大多数情况下，Adam 的默认超参数 ( $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ,  $\epsilon = 10^{-8}$ ) 都能工作得很好——这也是它成为“默认首选”的原因。

# 7 序列模型：从 RNN 到 Transformer

## 7.1 Vanilla RNN

RNN 的核心公式极其简洁：

$$h_t = \tanh(x_t W_x + h_{t-1} W_h + b) \quad (36)$$

每个时间步接受当前输入  $x_t$  和上一步隐藏状态  $h_{t-1}$ ，通过线性变换和  $\tanh$  激活输出新的隐藏状态。

```
def rnn_step_forward(x, prev_h, Wx, Wh, b):
    a_t = x @ Wx + prev_h @ Wh + b
    next_h = torch.tanh(a_t)
    return next_h
```

完整序列的前向传播只需在时间维度上展开循环：

```
for t in range(T):
    prev_h = h0 if t == 0 else h[:, t-1, :]
    next_h = rnn_step_forward(x[:, t, :], prev_h, Wx, Wh, b)
    h = torch.cat((h, next_h.unsqueeze(1)), dim=1) if t else next_h.unsqueeze(1)
```

RNN 的根本问题是梯度消失——与深层网络同源的连乘效应。每个时间步反向传播都要乘以  $W_h$  的雅可比矩阵。当  $W_h$  的谱半径  $< 1$  时， $T$  步连乘后梯度指数衰减；谱半径  $> 1$  则指数爆炸。序列长度  $T$  扮演的角色等价于深层网络的层数  $L$ 。这个结构性缺陷无法通过调参解决，只能通过门控机制（LSTM/GRU）引入“梯度高速公路”。

## 7.2 LSTM

LSTM 通过门控机制解决梯度消失问题。它引入了细胞状态  $c_t$  作为信息高速公路：

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} (x_t W_x + h_{t-1} W_h + b) \quad (37)$$

$$c_t = f \odot c_{t-1} + i \odot g \quad (38)$$

$$h_t = o \odot \tanh(c_t) \quad (39)$$

其中  $i, f, o$  分别是输入门、遗忘门、输出门， $g$  是候选记忆。遗忘门  $f$  是关键——当  $f \approx 1$  时，梯度可以无衰减地流过多个时间步。

## 7.3 Image Captioning

CaptioningRNN 将 CNN 提取的图像特征作为 RNN 的初始隐藏状态，然后逐词生成描述：

1. 图像特征  $v$  通过线性层投影为初始隐藏状态  $h_0 = v W_{\text{proj}}$
2. 每个时间步，输入当前词的 embedding，输出下一个词的得分
3. 训练时使用 teacher forcing（输入 ground truth），推理时使用 Greedy Decoding

## 7.4 Transformer

Transformer 完全抛弃了循环结构，用注意力机制处理序列依赖。

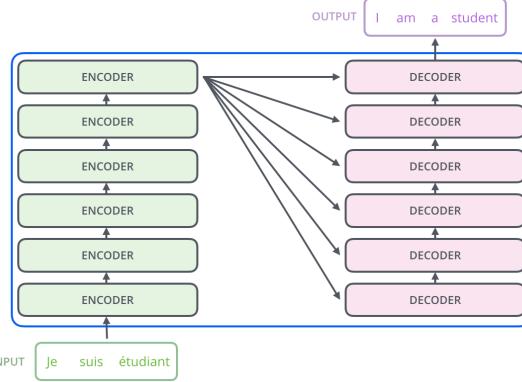


图 5 Transformer 架构：Encoder-Decoder 结构，核心是 Multi-Head Attention 和残差连接。

#### 7.4.1 位置编码

由于 Transformer 没有循环结构，需要显式注入位置信息：

$$PE(p, 2i) = \sin\left(\frac{p}{10000^{2i/d}}\right), \quad PE(p, 2i + 1) = \cos\left(\frac{p}{10000^{2i/d}}\right) \quad (40)$$

这种设计使得任意两个位置的编码之间的内积是位置差的函数，赋予模型感知相对位置的能力。

#### 7.4.2 Multi-Head Attention

注意力的核心公式：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \quad (41)$$

Multi-Head 将  $Q, K, V$  分别投影到  $h$  个子空间，独立计算注意力后拼接：

```

q = self.query(query).view(N, S, n_head, head_dim).transpose(1, 2) # (N,H,S,D)
k = self.key(key).view(N, T, n_head, head_dim).transpose(1, 2) # (N,H,T,D)
v = self.value(value).view(N, T, n_head, head_dim).transpose(1, 2) # (N,H,T,D)
scores = (q @ k.transpose(-2, -1)) * self.scale # (N,H,S,T)
attn = F.softmax(scores, dim=-1)
output = (attn @ v).transpose(1, 2).contiguous().view(N, S, E)
output = self.proj(output)

```

**工程本质：Attention 重新定义了“距离”**

抛开公式，Self-Attention 的核心工程价值是将序列中任意两个节点的信息传递距离从 RNN 的  $O(T)$  暴力压缩到了  $O(1)$ ——任意两个 token 通过一次矩阵乘法直接交互，无需经过中间时间步。这彻底解决了长程梯度消失：梯度不再需要连乘  $T$  个雅可比矩阵，而是通过 attention 权重直接回传到任意位置。代价是  $O(T^2)$  的计算和内存开销。

#### 7.4.3 Decoder Layer

每个 Decoder Layer 包含三个子层，均使用残差连接和 LayerNorm：

1. **Masked Self-Attention**: 目标序列的自注意力，使用因果掩码防止看到未来 token
2. **Cross-Attention**: 以编码器输出为 key/value，目标序列为 query
3. **Feed-Forward Network**: 两层 MLP (`Linear → GELU → Dropout → Linear`)

#### 7.4.4 Vision Transformer (ViT)

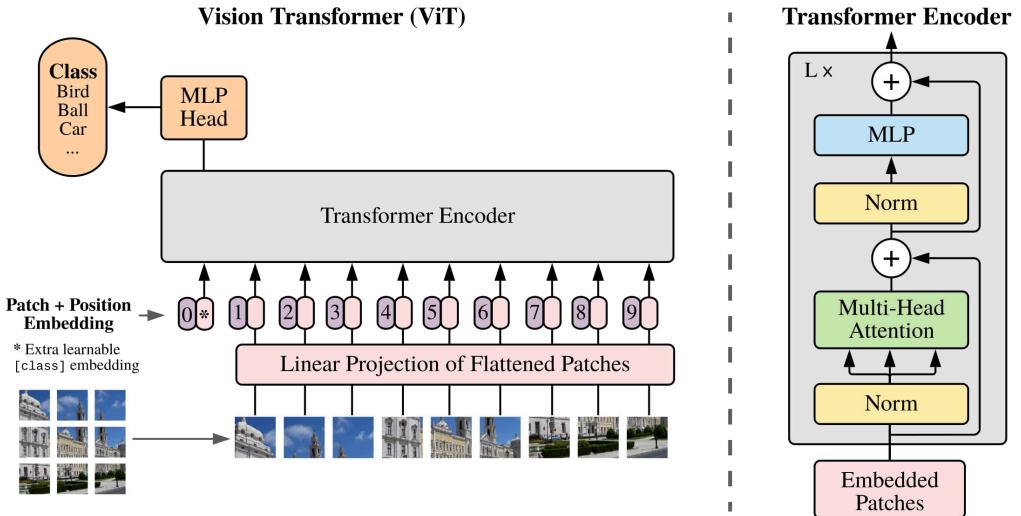


图 6 Vision Transformer (ViT): 图像切分为 patch 序列，经线性投影和位置编码后送入标准 Transformer Encoder。

ViT 将图像切分为固定大小的 patch，每个 patch 线性投影为 embedding：

```
# Patch Embedding: (N, C, H, W) -> (N, num_patches, embed_dim)
x = x.reshape(N, C, H//p, p, W//p, p)
x = x.permute(0, 2, 4, 1, 3, 5).reshape(N, num_patches, patch_dim)
out = self.proj(x)
```

加上可学习的 [CLS] token 和位置编码后，送入标准 Transformer Encoder，最终用 [CLS] token 的输出接分类头。

#### 7.4.5 Scaling Law: 为什么“简单架构 + 海量数据”能产生智能

Transformer 的成功还揭示了一个深刻的经验规律——**Scaling Law**。OpenAI 在 2020 年的研究发现，语言模型的性能（以 loss 衡量）与三个因素呈可预测的幂律关系：

- 模型参数量：增大模型 → loss 降低
- 训练数据量：增加高质量数据 → loss 降低
- 计算量：增加训练 FLOPs → loss 降低

这意味着：只要你愿意投入足够的算力和数据，一个足够大的 Transformer 就能持续变得更强。这解释了为什么 GPT 系列、ViT 等模型的架构设计惊人地简洁——它们的成功不在于复杂的归纳偏置，而在于规模本身。

架构的简洁性恰恰是 Scaling 的前提：只有足够通用、没有过多领域特定假设的架构，才能在大规模扩展时持续受益。这也解释了为什么 ViT 在小数据集上不如 CNN，但在 JFT-300M 等巨型数据集上彻底超越。

## 8 迁移学习：站在巨人的肩膀上

在进入自监督学习之前，有必要理解一个更基本的问题：为什么我们很少从零训练模型？

### 8.1 为什么要用预训练模型？

现代深度学习的一个核心洞察是：底层特征是通用的。在 ImageNet 上训练的 CNN，其底层卷积核学到的是边缘、纹理、颜色等低级视觉特征——这些特征在任何视觉任务中都有用。只有高层特征才是任务特定的。

迁移学习的标准流程：

1. 选择预训练模型：如在 ImageNet 上训练好的 ResNet-50
2. 冻结底层：保持低层卷积核不变（它们已经学到了通用特征）
3. 替换并训练顶层：用新的分类头替换原来的，在目标数据集上微调

这带来了巨大的实际优势：

- 节省算力：从零训练 ResNet-50 需要数天 GPU 时间，微调只需数小时
- 小数据集也能工作：预训练提供了强大的特征先验，即使目标数据集只有几千张图
- 更好的泛化：预训练模型见过百万张图像，其特征空间比从零学习更加鲁棒

自监督学习（SimCLR、CLIP、DINO）可以被理解为迁移学习的“终极形态”——它们学习的是不依赖人工标注的通用特征表示。

## 9 自监督学习与基础模型

### 9.1 SimCLR 对比学习

SimCLR 的核心思想：对同一张图像施加两种不同的数据增强，它们的特征应该相似（正样本对），不同图像的特征应该不同（负样本对）。

NT-Xent Loss (归一化温度缩放的交叉熵)：

$$\ell(i, j) = -\log \frac{e^{\frac{\text{sim}(z_i, z_j)}{\tau}}}{\sum_{k \neq i} e^{\frac{\text{sim}(z_i, z_k)}{\tau}}} \quad (42)$$

其中  $\text{sim}(z_i, z_j) = \frac{z_i^\top z_j}{\|z_i\| \cdot \|z_j\|}$  是余弦相似度， $\tau$  是温度参数。

向量化实现的关键是高效计算  $2N \times 2N$  的相似度矩阵：

```
norm_out = out / torch.linalg.norm(out, dim=1, keepdim=True)
sim_matrix = norm_out @ norm_out.T # (2N, 2N)
exponential = torch.exp(sim_matrix / tau)
mask = ~torch.eye(2*N, dtype=torch.bool) # 排除自身
denom = (exponential * mask).sum(dim=1, keepdim=True)
loss = torch.mean(-torch.log(numerator / denom))
```

### 9.2 CLIP

CLIP (Contrastive Language-Image Pre-training) 将图像和文本映射到同一嵌入空间。对齐方式与 SimCLR 类似，但正样本对是匹配的图文对。

实现了三个应用：

1. 零样本分类：计算图像特征与所有类别文本特征的余弦相似度，取最大值对应的类别
2. 图像检索：给定文本 query，在图像库中找到最相似的图像
3. 相似度矩阵：全向量化的余弦相似度计算

```
def get_similarity_no_loop(text_features, image_features):
    text_norm = text_features / text_features.norm(dim=-1, keepdim=True)
    image_norm = image_features / image_features.norm(dim=-1, keepdim=True)
    return text_norm @ image_norm.T # (N, M)
```

### 9.3 DINO 视频分割

DINO (Self-Distillation with No Labels) 的预训练特征具有惊人的语义聚类性质。我们利用 DINO 的 patch token 特征进行视频目标分割：

1. 提取每帧图像的 DINO patch features
2. 在第一帧使用标注的 mask 训练简单分类器
3. 将分类器应用于后续帧，实现零样本视频分割

## 10 扩散模型：从噪声中生成图像

### 10.1 DDPM 理论框架

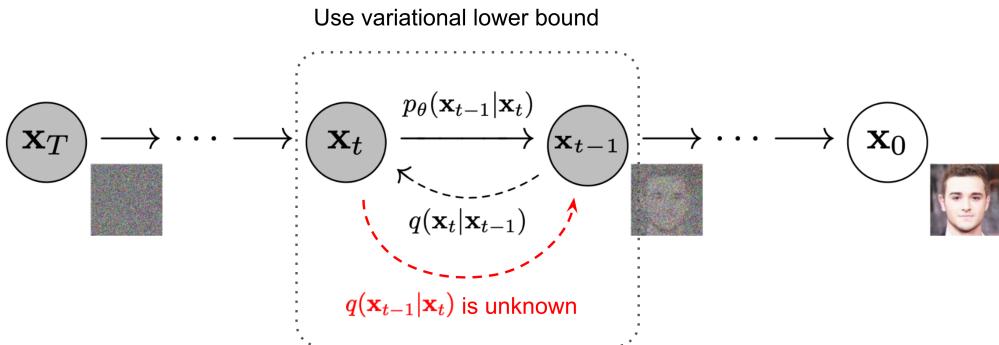


图 7 DDPM 前向与反向过程：前向过程逐步加噪直到纯高斯噪声，反向过程通过学习去噪逐步恢复原始图像。

Denoising Diffusion Probabilistic Models 定义了一个从数据逐步加噪到纯噪声的马尔可夫链。

前向过程（加噪）：

$$q(x_t | x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1 - \bar{\alpha}_t)I) \quad (43)$$

即  $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon$ , 其中  $\varepsilon \sim \mathcal{N}(0, I)$ 。

```
def q_sample(self, x_start, t, noise):
    sqrt_alpha_bar = extract(self.sqrt_alphas_cumprod, t, x_start.shape)
    sqrt_one_minus = extract(self.sqrt_one_minus_alphas_cumprod, t, x_start.shape)
    return sqrt_alpha_bar * x_start + sqrt_one_minus * noise
```

反向过程（去噪）的后验分布：

$$q(x_{t-1} | x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t, \tilde{\sigma}_t^2 I) \quad (44)$$

其中  $\tilde{\mu}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t$ 。

训练目标简化为预测噪声：

$$L = \mathbb{E}_{t, x_0, \varepsilon} [\|\varepsilon - \varepsilon_{\theta(x_t, t)}\|^2] \quad (45)$$

```
def p_losses(self, x_start, model_kwargs={}):
    t = torch.randint(0, nts, (b,), device=x_start.device).long()
    noise = torch.randn_like(x_start)
    x_t = self.q_sample(x_start, t, noise)
    model_pred = self.model(x_t, t, model_kwargs=model_kwargs)
    loss = F.mse_loss(model_pred, noise, reduction="none")
    loss = (loss * loss_weight).mean()
```

### 10.2 U-Net 架构

噪声预测网络使用 U-Net 结构：

- 下采样路径：逐步缩小空间分辨率，增加通道数
- 上采样路径：逐步恢复分辨率，通过跳连接（skip connections）融合高分辨率特征

- **时间嵌入**: 使用正弦位置编码将时间步  $t$  嵌入为向量，通过 FiLM 调制 (scale + shift) 注入每个 ResNet Block
- **Classifier-Free Guidance**: 训练时以概率  $p$  随机丢弃条件信息，推理时通过加权有条件和无条件预测的差值来增强条件控制

### 10.3 噪声调度

实现了三种  $\beta$  调度策略：

调度方式	特点
Linear	原始 DDPM 论文提出， $\beta$ 线性增长
Cosine	改善 SNR 分布，避免后期信息过早丢失
Sigmoid	适合高分辨率图像 ( $> 64 \times 64$ )