残差连接 (Residual Connection)

1. 梯度范数是什么?

简单说，**梯度范数**就是梯度的“长度”或“大小”。梯度本身是一个包含了很多偏导数的向量（比如 `[∂L/∂w1, ∂L/∂w2, ...]`），它指明了损失函数下降最快的方向。而范数（通常是L2范数，即向量所有元素的平方和再开方）就是用来衡量这个“下降趋势”有多强烈的**一个标量值**。梯度范数大，说明下降趋势陡峭；梯度范数小，说明趋势平缓。

2. 梯度消失/爆炸是什么?

在很深的网络里，梯度需要通过链式法则一层层地从后往前传播。

- **梯度消失 (Vanishing Gradients)**：如果每一层的雅可比矩阵（可以简单理解为激活函数的导数）都小于1，那么梯度在反向传播时会**连乘**很多个小于1的数，导致梯度越往前传就越小，最后变得几乎为0。靠近输入层的网络基本上收不到更新信号，学不到东西，就像信号衰减了。
- **梯度爆炸 (Exploding Gradients)**：反之，如果每一层的雅可比矩阵都大于1，梯度就会**连乘**很多个大于1的数，变得异常巨大，导致参数更新的步子迈得太大，直接“飞出”了最优解的区域，模型无法收敛。

3. 残差连接的计算公式? 求导公式?

假设一个网络块的输入是 `x`，这个块要学习的函数是 `F(x)`。

- **计算公式**：输出 `H(x)` 不再是 `F(x)`，而是 `H(x) = F(x) + x`。这个 `+ x` 的操作，就是**残差连接**（也叫恒等映射或短路连接）。网络块 `F(x)` 现在学习的是**残差** `H(x) - x`。
- **求导公式**：根据链式法则，损失 `L` 对输入 `x` 的导数 `∂L/∂x` 变为：
  `∂L/∂x = ∂L/∂H(x) * ∂H(x)/∂x = ∂L/∂H(x) * (∂F(x)/∂x + 1)`
  这个 `+ 1` 是关键!

4. 残差连接为什么可以解决梯度消失/爆炸?

看上面的求导公式，那个 `+ 1` 的项确保了**梯度在反向传播时，至少有一条“高速公路”可以直接传过去**，而不会被 `∂F(x)/∂x` 这个连乘项完全“掐断”或“放大”。即使 `∂F(x)/∂x` 非常小（趋近于0），总梯度里也还有一个 `1` 来保底，保证了梯度信号至少能“原封不动”地传到前一层。这就好像给梯度信号买了个“全损险”，极大地缓解了梯度消失问题。同理，它也稀释了梯度爆炸的风险。

卷积计算 (Convolution)

1. 计算机采用什么数据结构存储、处理图像?

通常是一个**三维的数组（或张量）**。它的维度是 `[Height, Width, Channels]` (高、宽、通道数) 或者 `[Channels, Height, Width]`。

- **Height/Width**：就是图像的像素尺寸。
- **Channels**：对于灰度图，通道数是1。对于彩色图（RGB），通道数是3，分别代表红、绿、蓝三个颜色通道的强度。

2. 什么是卷积操作？在图像处理中是怎么做的？

卷积操作就像一个**“带权重的局部信息提取器”**。
在图像处理中，它是一个小的矩阵（称为**卷积核**或**滤波器**），在整个输入图像上**滑动**。在每一个位置，卷积核都会和它覆盖的那一小块图像区域进行**元素对应相乘，然后求和**，得到一个单一的输出值。这个输出值就代表了卷积核在当前位置“提取”到的特征。整个图像滑动一遍后，就生成了一张新的、更小的“特征图”。

3. 卷积操作中常用的几个超参数有哪些？

- **卷积核大小 (Kernel Size)**：决定了“看”多大范围的局部信息，比如 `3x3` 或 `5x5`。
- **步长 (Stride)**：卷积核每次滑动的像素格数。步长为1就是逐格滑动，步长为2就是隔一格滑动，会使输出的特征图尺寸变小。
- **填充 (Padding)**：在图像的边缘填充一圈0。这主要是为了控制输出特征图的尺寸，避免因为卷积操作导致图像尺寸缩小得太快，同时也能更好地处理边缘信息。
- **输出通道数 (Output Channels)**：也就是使用多少个不同的卷积核。每个卷积核负责提取一种特定的特征（比如横向边缘、竖向边缘、某个颜色等），用10个卷积核就会产生10张特征图。

4. 卷积层一定会减少参数数量吗？

不一定，但它通常能极大地减少与全连接层相比的参数量。

- **相比全连接层**：是的，参数量大大减少。一个 `3x3` 的卷积核只有9个参数，它在整个图像上共享使用（**参数共享**）。而如果用全连接层处理图像，每个像素点都要和下一层的每个神经元连接，参数量会是天文数字。
- **相比前一层**：不一定。如果一个卷积层的输入通道数是3，输出通道数是64，即使卷积核很小，总参数量也可能比前一层多。但它的计算方式仍然是局部的和共享的。

5. 卷积神经网络为什么对图像处理有效？

主要基于两个核心思想，完美地契合了图像数据的特性：

- **局部相关性 (Local Connectivity)**：图像中的一个像素，只和它周围的像素关系最密切。卷积操作只在局部区域进行计算，正好捕捉了这种特性。
- **平移不变性 (Translation Invariance)**：图像中的一个物体（比如一只猫），无论出现在左上角还是右下角，它仍然是一只猫。卷积的**参数共享**机制意味着，同一个卷积核（比如一个“猫耳朵检测器”）可以在图像的任何位置工作，大大提高了模型的泛化能力和效率。

6. 卷积时有的特征可能会有损失，有什么改进办法吗？

是的，尤其是池化层（Pooling）和大的步长（Stride）会导致信息损失。

- **使用更小的步长和卷积核**：用多个 `3x3` 步长为1的卷积层，代替一个大的 `7x7` 卷积层，可以在保持感受野的同时，更精细地提取特征。
- **残差连接**：允许底层信息直接“跳跃”到高层，弥补在卷积过程中可能丢失的细节。
- **空洞卷积 (Dilated Convolution)**：在不增加参数和计算量的前提下，增大感受野，能看到更广阔的范围，有助于捕捉上下文信息，而不会像池化那样直接丢弃信息。
- **注意力机制**：引入注意力模块（如 SE-Net, CBAM），让网络自适应地去“关注”那些信息量最大的通道或空间区域，而不是平等对待所有特征。

好的，我们继续攻克 **Task 2** 剩下的硬核部分。

Transformer

注意力机制 (Attention Mechanism)

1. 用 Python 实现点乘注意力机制

点乘注意力（Dot-Product Attention）是 Transformer 的核心。它回答了一个问题："当处理一个序列中的某个元素时，我应该给予序列中其他元素多少关注度？"

- **自注意力 (Self-Attention)**：Query, Key, Value 都来自同一个输入序列。它计算的是序列内部元素之间的依赖关系。
- **交叉注意力 (Cross-Attention)**：Query 来自一个序列（比如解码器的输出），而 Key 和 Value 来自另一个序列（比如编码器的输出）。它计算的是两个不同序列之间的对齐关系。

下面的代码实现了这个机制，通过一个 `context` 参数来区分自注意力和交叉注意力。

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

def dot_product_attention(query, key, value, mask=None, context=None):
    """
    计算点乘注意力。

    Args:
        query (torch.Tensor): 查询张量，形状 (batch_size, num_queries, d_k)
        key (torch.Tensor): 键张量，形状 (batch_size, num_keys, d_k)
        value (torch.Tensor): 值张量，形状 (batch_size, num_keys, d_v)
        mask (torch.Tensor, optional): 掩码张量. Defaults to None.
        context (torch.Tensor, optional): 用于交叉注意力的上下文. Defaults to None.

    Returns:
        torch.Tensor: 注意力输出
        torch.Tensor: 注意力权重
    """
    # 如果提供了 context，则执行交叉注意力
    if context is not None:
        # 在交叉注意力中，key 和 value 来自 context
        key = context
        value = context

    # 1. 计算 Query 和 Key 的点乘分数
    #    Q @ K.T -> (..., num_queries, d_k) @ (..., d_k, num_keys) -> (...,
num_queries, num_keys)
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) /
torch.sqrt(torch.tensor(d_k, dtype=torch.float32))

    # 2. 应用掩码 (Mask)
    #    在计算 softmax 之前，将需要忽略的位置的分数设置为一个非常小的负数
```

```
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # 3. 计算注意力权重 (Softmax)
    #    对分数的最后一个维度（num_keys）进行 softmax，得到概率分布
    attention_weights = F.softmax(scores, dim=-1)

    # 4. 将权重应用于 Value
    #    Weights @ V -> (..., num_queries, num_keys) @ (..., num_keys, d_v) -> (...,
num_queries, d_v)
    output = torch.matmul(attention_weights, value)

    return output, attention_weights

# --- 示例 ---
# 假设 batch_size=1, 序列长度=4, 特征维度=8
seq_len = 4
d_model = 8
x = torch.randn(1, seq_len, d_model) # 我们的输入序列

# 在实际模型中，Q, K, V 是由 x 经过不同的线性变换得到的
# 这里为了简化，我们假设它们就是 x
query, key, value = x, x, x

# --- 自注意力示例 ---
print("--- Self-Attention ---")
output_self, weights_self = dot_product_attention(query, key, value)
print("Output shape:", output_self.shape) # (1, 4, 8)
print("Weights shape:", weights_self.shape) # (1, 4, 4) - 4x4的权重矩阵，表示4个词之间的
关注度

# --- 交叉注意力示例 ---
print("\n--- Cross-Attention ---")
# 假设有一个来自编码器的上下文 context
context = torch.randn(1, seq_len, d_model)
# query 来自解码器（这里用x模拟），key和value来自context
output_cross, weights_cross = dot_product_attention(query, None, None,
context=context)
print("Output shape:", output_cross.shape) # (1, 4, 8)
print("Weights shape:", weights_cross.shape) # (1, 4, 4)
```

位置编码 (Positional Encoding)

1. **为什么Transformer本身没有任何位置意识?**
   因为它处理输入序列的核心操作——自注意力机制，是**置换不变 (Permutation-invariant)** 的。自注意力计算的是两两向量之间的点乘相似度，这个计算过程与向量在序列中的位置无关。你把句子“我爱你”和“你爱我”输入进去，如果不加位置信息，对于“爱”这个字来说，它看到的“我”和“你”的上下文是完全一样的，无法区分这两个句子的语义差异。Transformer 看待输入，就像是把所有词丢进一个“袋子”里，失去了顺序。
2. **绝对/相对/可学习位置编码分别是什么，有何优劣?**

- **绝对位置编码 (Absolute)**：给序列中的每个位置分配一个固定的、唯一的向量。
  - **是什么**：最经典的是 `sin/cos` 编码。它用不同频率的正弦和余弦函数来为每个位置生成一个独特的向量。位置 `pos` 的编码向量的第 `i` 维，是由 `pos` 和 `i` 共同决定的。
  - **优劣**：优点是无需训练，并且理论上可以外推到比训练时更长的序列。缺点是它强加了一种固定的位置表示，可能不是对所有任务都最优。
- **相对位置编码 (Relative)**：不关心一个词的绝对位置（比如"第5个词"），只关心两个词之间的相对距离（比如"你比我晚出现2个位置"）。
  - **是什么**：在计算注意力分数时，除了考虑 `Query` 和 `Key` 的内容相似度，还额外加上一个表示它们相对位置差异的偏置项。
  - **优劣**：优点是泛化能力更强，对序列长度的变化更不敏感。缺点是实现起来通常比绝对位置编码更复杂。
- **可学习位置编码 (Learnable)**：创建一个位置编码矩阵（比如大小为 `[最大序列长度, 词向量维度]`），然后像模型的其他权重一样，让它在训练过程中**自己去学习**每个位置应该如何表示。
  - **是什么**：就是 `nn.Embedding(max_seq_len, d_model)`。
  - **优劣**：优点是模型可以根据具体任务自适应地学习出最优的位置表示。缺点是它无法外推到比训练时设定的"最大序列长度"更长的序列，并且增加了模型的参数量。

3. **LLM中常用的位置编码有哪些?**
   - **RoPE (Rotary Positional Embedding)**：旋转位置编码。这是目前**大模型领域最主流、效果最好的方案之一**（比如 LLaMA, Qwen 等模型都在用）。它通过将位置信息编码为旋转矩阵，并作用于 Query 和 Key 向量上，巧妙地将绝对位置信息融入到了自注意力计算中，同时又天然地具备了很好的相对位置特性。
   - **ALiBi (Attention with Linear Biases)**：是另一种主流的相对位置编码方案。它非常简单，直接在注意力分数矩阵上，根据 Query 和 Key 的距离，加上一个线性的惩罚项，距离越远，惩罚越大。它的优点是简单高效，并且外推能力极强。

层归一化 (LayerNorm)

1. **什么是 Layer Normalization? 为什么需要它?**
   - **是什么**：是一种归一化技术。它对**单个样本**的**所有特征**（即一个词向量的所有维度）进行归一化，计算这些特征的均值和方差，然后用它们来标准化这个样本。
   - **为什么需要**：在深层网络中，每一层的输出分布会随着训练不断变化（内部协变量偏移），这使得下一层很难适应。LayerNorm 强制将每一层输入的分布都稳定在均值为0、方差为1的"舒适区"，使得训练过程更稳定、收敛更快。
2. **LayerNorm 和 BatchNorm 的区别?**
   这是面试高频题！核心区别在于**归一化的维度**不同：
   - **BatchNorm (批量归一化)**：对一个 **batch** 内的**所有样本**，在**同一个特征维度**上进行归一化。它计算的是"这个batch里所有句子的第一个词的第一个维度"的均值和方差。它受 `batch_size` 大小影响很大，不适合处理变长序列（RNN/Transformer）。
   - **LayerNorm (层归一化)**：对 **batch** 内的**每一个样本**，在它**自己的所有特征维度**上进行归一化。它计算的是"第一个句子的第一个词的所有维度"的均值和方差。它完全独立于其他样本和 `batch_size`，非常适合处理变长序列。

Mask

1. **什么是 Causal Mask / Look-ahead Mask？作用是什么？**
   - **是什么**：是一个上三角矩阵，对角线及以下都是1，以上都是0。
   - **作用**：用在 Transformer 的**解码器 (Decoder)** 自注意力层中。它的作用是**"遮住未来"**。在生成式任务中（比如语言模型预测下一个词），模型在预测第 `t` 个词时，**只能看到 `t` 位置以及之前的所有词**，绝不能让它"偷看"到 `t+1` 位置及之后的答案。Causal Mask 在计算注意力分数时，会把所有"未来"位置的分数设置为负无穷，这样在 Softmax 之后，这些位置的注意力权重就变成了0，相当于被忽略了。

束搜索 (Beam Search)

1. **为什么LLM输出序列是一个搜索问题?**
   因为在每一步生成词的时候，LLM 输出的都不是一个确定的词，而是一个**覆盖整个词汇表的概率分布。**比如，在"我爱你"后面，模型可能认为"中国"的概率是0.1，"北京"的概率是0.05，"..."。要生成一个完整的、概率最高的句子，你需要在每一步都做出选择，而这些选择会影响后续的概率分布。这本质上就是在一个巨大的、由概率构成的树状空间中，**搜索一条从根节点到叶节点的、累积概率最大（或最优）的路径。**
2. **用 Python 分别实现贪心搜索和束搜索**

```
import numpy as np

# 假设这是一个假的LLM输出，表示在3个时间步，词汇表大小为5的概率分布
# 每一行都是一个时间步的输出概率
fake_logits = np.array([
    [0.1, 0.2, 0.3, 0.1, 0.3], # t=0
    [0.5, 0.1, 0.1, 0.2, 0.1], # t=1
    [0.1, 0.1, 0.1, 0.5, 0.2]  # t=2
])
vocab = ["a", "b", "c", "d", "e"]
log_probs = np.log(fake_logits) # 通常使用对数概率，避免连乘导致数值下溢

def greedy_search(log_probs):
    """贪心搜索：每一步都选择当前概率最高的词"""
    sequence = []
    for t in range(log_probs.shape[0]):
        best_word_idx = np.argmax(log_probs[t])
        sequence.append(vocab[best_word_idx])
    return "".join(sequence)

def beam_search(log_probs, beam_size=3):
    """束搜索：每一步都保留概率最高的k个候选序列"""
    sequences = [[list(), 0.0]] # [[序列], 累积对数概率]

    for row in log_probs:
        all_candidates = list()
        for i in range(len(sequences)):
            seq, score = sequences[i]
            for j in range(len(row)):
```

```
                candidate = [seq + [j], score - row[j]] # 使用减法因为log_probs是负数
                all_candidates.append(candidate)

        # 排序所有候选，选择最好的 beam_size 个
        ordered = sorted(all_candidates, key=lambda tup: tup[1])
        sequences = ordered[:beam_size]

    # 将索引转换为词
    best_seq_indices = sequences[0][0]
    return "".join([vocab[i] for i in best_seq_indices])

print("Greedy Search Result:", greedy_search(log_probs)) # 输出: c a d
# 为什么是cad? t=0时c(0.3)最高, t=1时a(0.5)最高, t=2时d(0.5)最高

print("Beam Search Result:", beam_search(log_probs, beam_size=2)) # 输出可能不同

# 为什么需要束搜索？
# 贪心搜索只看眼前，可能导致“一步错，步步错”。比如第一步选了c，但也许第一步选b，
# 虽然b的初始概率低，但它可能解锁一个后续概率极高的序列。
# 束搜索通过保留多个候选（beam_size），在一定程度上缓解了这个问题。
# 它在搜索的“广度”和计算成本之间做了一个很好的权衡，通常能生成比贪心搜索更流畅、更合理的句子。
```

总结

1. **Transformer为什么可以做到这么大?**
   - **并行计算能力**：与必须按顺序计算的 RNN 不同，Transformer 的自注意力机制可以一次性计算完序列中所有词之间的关系，这使得它能极好地利用现代 GPU 的大规模并行计算能力。
   - **更短的路径依赖**：在 Transformer 中，任意两个位置的词都可以通过自注意力直接建立联系，路径长度是 `O(1)`。而在 RNN 中，信息需要一步步传递，路径长度是 `O(n)`，这使得 Transformer 更容易捕捉长距离依赖关系。
   - **强大的表达能力**：多头注意力、残差连接、层归一化等组件的组合，被证明是一个极其强大和灵活的函数逼近器。
2. **它的哪些设计可以缓解梯度爆炸?**
   - **残差连接 (Residual Connection)**：这是**最主要**的功臣。它为梯度提供了一条“高速公路”，保证了梯度信号在反向传播时不会因为深层网络的连乘效应而消失或爆炸。
   - **层归一化 (Layer Normalization)**：通过在每一层之后都将输出的分布“拉回”到一个稳定的、标准的状态，它也极大地稳定了网络的动态，使得梯度不会剧烈波动，从而缓解了梯度爆炸/消失的风险。

Diffusion

1. 什么是正态分布？什么是高斯噪声?

- **正态分布 (Normal Distribution)**：也叫高斯分布，是自然界中最常见的一种概率分布。它的形状像一个对称的“钟形曲线”，由均值 `μ`（决定了钟的中心位置）和标准差 `σ`（决定了钟的“胖瘦”程度）两个参数决定。

- **高斯噪声 (Gaussian Noise)**：就是从一个均值为0的正态分布中随机采样的数值。它看起来就像电视机没信号时的那种“雪花点”，是一种完全随机、无规律的扰动。

2. 什么是扩散模型？如何理解它的正向/反向过程？

- **是什么**：是一种生成模型，灵感来源于物理学中的扩散过程（比如一滴墨水在清水中散开）。
- **正向过程 (Forward Process / Denoising)**：这是一个**固定的、不需要学习**的过程。它模拟了“把一张清晰的图片逐渐变模糊”的过程。具体做法是，从一张真实的、清晰的图片 `x_0` 开始，在很多个时间步 `T` 中，每一步都往图片上加一点点高斯噪声，直到第 `T` 步，图片完全变成了一张纯粹的高斯噪声图 `x_T`。
- **反向过程 (Reverse Process / Denoising)**：这是**需要模型学习**的核心过程。它要做的是正向过程的逆操作：“从一张纯噪声图 `x_T` 开始，一步步地把噪声去掉，最终还原出一张清晰的图片 `x_0`”。模型（通常是一个 U-Net 结构）在每一步 `t`，都需要预测出在 `x_t` 这张“半模糊”的图片里，到底哪些是“噪声”，然后把它减掉，得到更清晰一点的 `x_{t-1}`。

3. Diffusion模型如何进行图像生成？

非常简单：

1. 从一个标准正态分布中，随机采样一张**纯粹的高斯噪声图** `x_T`。
2. 把这张噪声图和当前的时间步 `T`，输入到我们已经**训练好的那个“去噪”神经网络**中。
3. 模型会预测出在 `T` 时刻应该去掉的噪声，我们用它来得到一张稍微清晰一点的图片 `x_{T-1}`。
4. 重复这个过程，把 `x_{T-1}` 和时间步 `T-1` 再输入模型，得到 `x_{T-2}`...
5. ...直到时间步 `t=0`，我们就得到了一张全新的、清晰的、由模型“想象”出来的图片 `x_0`。

- **如果加入了文本条件 (Text Conditioning)**，比如 Stable Diffusion，那么在每一步去噪时，除了输入图片 `x_t` 和时间步 `t`，还会输入文本的编码，引导模型朝着“一只正在太空骑马的宇航员”这个方向去噪。

4. Diffusion 一定是 CNN 吗？

不一定，但目前最成功、最主流的 Diffusion 模型架构，其核心确实是基于 CNN 的 U-Net。

- **为什么 U-Net (CNN) 这么好用？** 因为图像去噪这个任务，和图像分割非常相似，都需要模型既能理解图像的局部纹理（比如噪声的模式），又能理解全局结构（比如这是一张人脸）。U-Net 的编码器-解码器结构加上跳跃连接，完美地契合了这种多尺度的信息处理需求。卷积的局部性和平移不变性也非常适合处理图像。
- **未来的可能性**：已经有研究在探索使用 Transformer (ViT) 或者其他架构来作为 Diffusion 模型的去噪网络，但这还不是主流。所以可以说，Diffusion 模型的成功，与 CNN 架构的强大能力是密不可分的，但两者之间没有绝对的绑定关系。