
Pie-Lab 2025年暑期培训 Practice1

徐浩
Pie-Lab
北京理工大学计算机学院
北京市海淀区中关村南大街
526358612@qq.com

Abstract

本文是“实践II：掌握注意力机制”的实验报告。

1 数学原理

1.1 Scaled Dot-Product Attention

对于输入序列 (x_1, x_2, \dots, x_n) ，希望能经过attention计算出每个输入 x_i 的上下文表示 c_i ，它蕴含着 x_i 和它的上下文之间的联系，整体结构如图1。

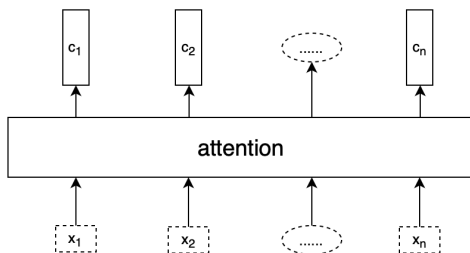


Figure 1: attention_overview

下面以计算 c_2 为例，阐述其数学原理，计算过程如图2。

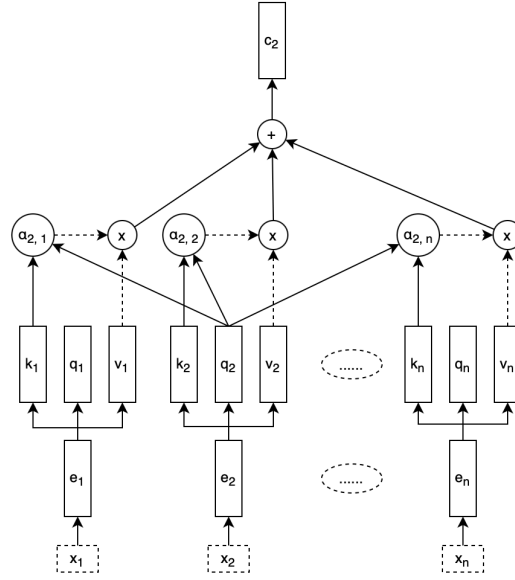


Figure 2: attention_calculation

这里与transformer原文保持一致，所有向量都是行向量。

首先将每个输入 x_i 转换为嵌入向量（即维度为 $n \times d_{\text{model}}$ 矩阵的第 i 行），并加入位置编码（因为attention不带有位置信息），得到向量 $e_i \in \mathbb{R}^{d_{\text{model}}}$ 。

将每个 e_i 分别做三次投影变换，得到：

$$\begin{aligned} q_i &= e_i W^Q, & W^Q &\in \mathbb{R}^{d_{\text{model}} \times d_k}, & q_i &\in \mathbb{R}^{d_k} \\ k_i &= e_i W^K, & W^K &\in \mathbb{R}^{d_{\text{model}} \times d_k}, & k_i &\in \mathbb{R}^{d_k} \\ v_i &= e_i W^V, & W^V &\in \mathbb{R}^{d_{\text{model}} \times d_v}, & v_i &\in \mathbb{R}^{d_v} \end{aligned}$$

然后使用点积注意力，将 q_2 分别和 $k_1, k_2 \dots k_n$ 做点积，得到 $a_{2,1}, a_{2,2} \dots a_{2,n}$ ，对它们求softmax，得到 $\alpha_{2,1}, \alpha_{2,2} \dots \alpha_{2,n}$ ，它们被称作相关性因子， $\alpha_{i,j}$ 越接近1，则 i 就与 j 的相关性越强。

再将 $\alpha_{2,i}$ 和 v_i 相乘、相加，就得到了 x_2 的上下文向量 c_2 。

因此， x_i 与 x_2 越相关，其相关性因子就越大，其对 c_2 的贡献就越多。

将上述结果整理为矩阵形式，定义输入矩阵 E ：

$$E = \begin{bmatrix} e_1 \\ \vdots \\ e_n \end{bmatrix} \in \mathbb{R}^{n \times d_{\text{model}}}$$

矩阵 Q 、 K 、 V ：

$$\begin{aligned} Q &= \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix} = E W^Q, & W^Q &\in \mathbb{R}^{d_{\text{model}} \times d_k}, & Q &\in \mathbb{R}^{n \times d_k} \\ K &= \begin{bmatrix} k_1 \\ \vdots \\ k_n \end{bmatrix} = E W^K, & W^K &\in \mathbb{R}^{d_{\text{model}} \times d_k}, & K &\in \mathbb{R}^{n \times d_k} \end{aligned}$$

$$V = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = EW^V, \quad W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}, \quad V \in \mathbb{R}^{n \times d_v}$$

则Scaled Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \in \mathbb{R}^{n \times d_v}$$

其中, 当维度 d_k 比较大时, q 和 k 点积的数值可能会变得很大, 这会导致各个数值大小很不均匀, 经过 softmax 之后, 这种不均匀被进一步放大 (因为它是指数缩放), 会导致某个相关性因子极大, 其他都极小, 这会影响训练效果。因此除以 $\sqrt{d_k}$ 可以缩小各个数值的差距, 进而缓解这种情况。

1.2 Multi-Head Attention

为了充分提取序列的特征, 引入了多头自注意力, 我们希望每个头提取的特征都不太一样。每个头的注意力计算同上, 但是每一头的权重 W 都不一样, 所以提取的特征也不一样。

第 i 头注意力:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \in \mathbb{R}^{n \times d_v}$$

注意, 这里的 Q 、 K 、 V 应该就等于上述的输入矩阵 E , 不再默认它乘了投影矩阵 W 。

合并 h 头注意力:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}, \quad \text{MultiHead} \in \mathbb{R}^{n \times d_{\text{model}}}$$

最后输出 MultiHead 的维度和输入 E 的维度一致, 因此该attention模块可以叠加多层。

1.3 ViT

ViT是将transformer用到cv领域的代表作, 整体方法仿照nlp领域, 其架构见图3左侧。

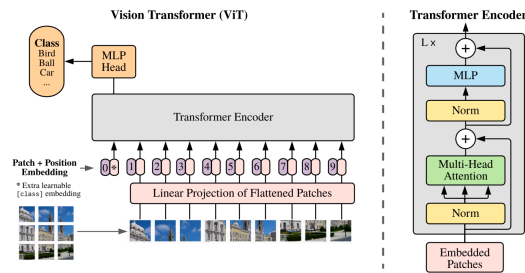


Figure 3: vit

ViT的思想非常简单, 先将图片切成许多个patch, 将每个patch展开成向量, 经过全连接层得到`patch_embedding`, 再加上位置编码`position_embedding`, 这样就把一张图像转换成了一串序列。这里由于是分类任务, 因此在序列前面加上CLS, 它的输出代表了整个序列的信息, 这种做法和nlp领域一致。

将序列输入 L 层`transformer_encoder`模块, 见图3右侧, 其内部主要是Multi-Head Attention和残差连接。最后的输出也是序列, 每个元素都是一个向量。

CLS对应的输出向量蕴涵了整个序列的信息, 将它经过线性分类层即可得到预测的类别。

2 模型结构与细节

在这一部分，我用文字结合代码的方式，把“模型结构”和“实现细节与关键代码”一起阐述。

首先看整体的vit结构，见图4。对输入x进行patch_embed，再在前面拼接cls（它被初始化为embedded维向量，所以无需patch_embed），再加上position_embed，这样就构成了输入的序列，每个元素都是向量。将序列经过transformer_encoder计算，然后取出cls的输出，经过线性层分类即可。

```
def forward(self, x):
    B = x.size(0) # 批量大小
    x = self.patch_embed(x) # [B, 3, 32, 32] -> [B, N, E]
    cls = self.cls.expand(B, -1, -1) # 把[CLS]向量扩展为批量个
    x = torch.cat((cls, x), dim=1) # cls与x拼接: [B, N+1, E]
    x = self.pos_embed(x)
    x = self.transformer_encoder(x)
    cls_out = self.layer_norm(x[:, 0]) # 归一化并取出CLS: [B, E]
    return self.classifier(cls_out) # 用CLS分类: [B, C]
```

Figure 4: vit_code

然后看patch_embed，见图5。按从左到右、从上到下的顺序，以4*4大小划分patch。这里可以用代码中的卷积层实现，其本质就是将每个patch展成向量，经过全连接层投影到embed_dim维的向量。

```
def __init__(self):
    super().__init__()
    # 用卷积层实现：将图像切成patch，每个patch映射为E维向量，相当于全连接
    self.proj = nn.Conv2d(IMG_CHANNELS, EMBED_DIM, kernel_size=PATCH_SIZE, stride=PATCH_SIZE)

def forward(self, x):
    x = self.proj(x) # [B, 3, 32, 32] -> [B, E, H/P, W/P]，相当于 H/P * W/P 个patch
    x = x.flatten(2).transpose(1, 2) # 将H/P * W/P展开为一列，并交换1, 2维度: [B, H/P * W/P, E]
    return x
```

Figure 5: patch_embedding

然后看position_embed，见图6。这里采用最简单的“可学习的1D位置编码”，和vit原文一致，即定义一个自学习的矩阵，以每个patch所在的位置为索引，去查矩阵中的该行，这个行向量就作为它的位置编码。

```
def __init__(self):
    super().__init__()
    # 1: 利用广播机制，将位置编码添加到整个批量上，这里采用自学习矩阵
    self.position_embed = nn.Parameter(torch.randn(1, NUM_PATCHES+1, EMBED_DIM))

def forward(self, x):
    return x + self.position_embed # x是PatchEmbedding并加上cls之后的
```

Figure 6: position_embedding

然后看transformer_encoder，见图7。它的实现和图3右侧的架构图一模一样，其内部用到了自定义的MultiheadAttention，剩余部分很简单。

```

def __init__(self):
    super().__init__()
    self.layer_norm1 = nn.LayerNorm(EMBED_DIM)
    self.attn = MultiheadAttention()
    self.layer_norm2 = nn.LayerNorm(EMBED_DIM)
    self.mlp = nn.Sequential(
        nn.Linear(EMBED_DIM, int(EMBED_DIM * MLP_RATIO)),
        nn.GELU(), # 激活函数
        nn.Dropout(DROP_RATE),
        nn.Linear(int(EMBED_DIM * MLP_RATIO), EMBED_DIM),
        nn.Dropout(DROP_RATE),
    )

def forward(self, x):
    x = x + self.attn(self.layer_norm1(x))
    x = x + self.mlp(self.layer_norm2(x))
    return x

```

Figure 7: transformer_encoder

最后看MultiheadAttention，见图8。首先将x经过线性投影得到q、k、v，将投影的结果分为h份，即h个头。对每个头，按照第一节的“Attention(Q, K, V)”公式计算，再把所有头的结果拼接，得到该层encoder的输出。该输出的维度和输入一致，因此可以叠加多层。

```

def __init__(self):
    super().__init__()
    self.scale = HEAD_DIM ** -0.5 # 缩放因子，避免点积过大
    self.qkv = nn.Linear(EMBED_DIM, EMBED_DIM * 3) # 一次性得到q, k, v
    self.dropout1 = nn.Dropout(DROP_RATE)
    self.out = nn.Linear(EMBED_DIM * 2, EMBED_DIM)
    self.dropout2 = nn.Dropout(DROP_RATE)

def forward(self, x):
    B, N, E = x.shape # 批量大小, patch数量 (含cls), 嵌入维度
    # 首先对x线性变换，再拆分q, k, v三部分，每部分都是把x映射到E维
    # 每部分h个头，每个头hd维，且h*hd = E
    # 调整维度顺序: [3, B, h, N, hd]
    qkv = self.qkv(x).reshape(B, 3, NUM_HEADS, HEAD_DIM).permute(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2] # [B, h, N, hd] 这里hd就是原文的dk, dv
    attn_scores = (q @ k.transpose(-2, -1)) * self.scale # [B, h, N, N]
    attn_probs = attn_scores.softmax(dim=-1)
    attn_probs = self.dropout1(attn_probs)
    attn_out = (attn_probs @ v) # [B, h, N, hd]
    attn_out = attn_out.transpose(1, 2).reshape(B, N, E) # 拼接各个头: -> [B, N, h, hd] -> [B, N, E]
    attn_out = self.dropout2(self.out(attn_out))
    return attn_out

```

Figure 8: MultiheadAttention

3 实验结果与分析

本实验的设置见图9

```

BATCH_SIZE = 128
NUM_CLASSES = 10
EPOCHS = 100 # 150
DROP_RATE = 0.2
IMG_CHANNELS = 3 # 图像为3通道
IMG_SIZE = 32 # 图像: 32*32*3
PATCH_SIZE = 4 # 切出的patch: P*P*3
NUM_PATCHES = (IMG_SIZE // PATCH_SIZE) ** 2 # 一张图切出的patch的数量
# 重要调参
WEIGHT_DECAY = 1e-3
LR = 1e-3
EMBED_DIM = 192
NUM_HEADS = 3 # 注意力头数, 这里要求E是h的倍数
HEAD_DIM = EMBED_DIM // NUM_HEADS # 每个头的维度
NUM_ENCODERS = 6 # encoder层数
MLP_RATIO = 4 # encoder-mlp中将E投影到x倍

```

Figure 9: configs

损失见图10，准确率见图11。

概要输出信息如下（完整输出见<https://github.com/xuhao00/pie-lab>）：

Training time: 27m 37s

Best test accuracy: 0.8309 at epoch 83

Time at best accuracy: 22m 56s

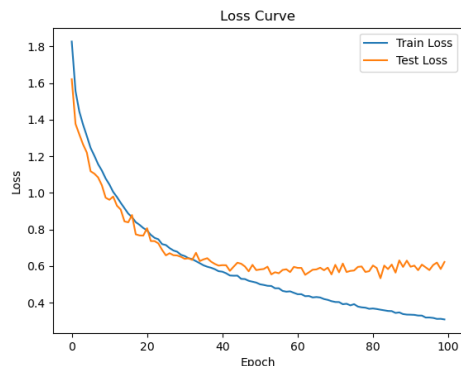


Figure 10: vit_loss

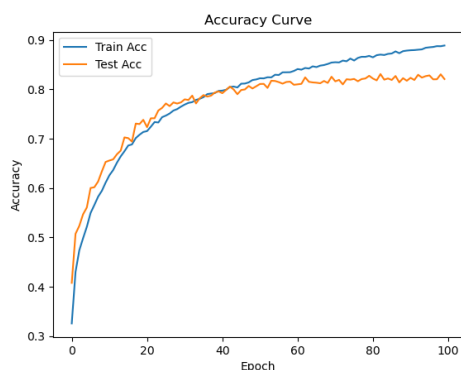


Figure 11: vit_accuracy

对上述实验结果分析如下：

- **准确率** 经过多组超参数实验（包括修改嵌入维度、注意力头数、encoder层数）发现，测试准确率基本都是停在0.82左右，然后测试损失开始上升、训练准确率保持上升，也就是过拟合。
- **效率** resnet34只需要2分钟就能到0.8，而vit需要13分钟。从最高准确率来看，resnet34只需要10分钟就能达到训练准确率0.98、测试准确率0.84，而vit训练了27分钟只能达到训练准确率0.89、测试准确率0.83。

4 疑惑

本实验存在的疑惑如下：

- **损失和准确率** 上图中的损失和准确率为什么一直在抖动，这是正常现象吗？
- **最高准确率** 我测了很多组超参数都在0.8左右，我不清楚是因为参数设置还不够合理，还是因为这种架构的模型能力确实有限。此外，我很想知道有没有一组超参数让它的测试准确率达到0.95左右。
- **效率** 为什么resnet每个周期的准确率提升非常明显（最高0.05，最低0.01），而vit的准确率提升极其缓慢（最高0.01，最低0.001），再加上它训一个周期的时间很长，给我的感觉就是训练极其困难。
- **调参** 没有很好掌握如何调超参数，从实验结果就只能看出什么时候过拟合了，然后去数据增强、降低模型复杂度、dropout等等一顿往上加，感觉自己的调参经验很粗糙，不知道有没有比较系统完整的调参经验可以学学。