第 4 次实验报告

2022111663　刘博飞

2024 年 11 月 10 日

1　CLIP 运行原理

1.1　transformer 的结构与作用

Transformer 由论文《Attention is All You Need》提出，现在是谷歌云 TPU 推荐的参考模型。论文相关的 Tensorflow 的代码可以从 GitHub 获取，其作为 Tensor2Tensor 包的一部分。哈佛的 NLP 团队也实现了一个基于 PyTorch 的版本，并注释该论文。

由于 CLIP 主要使用到了 transformer 架构中的 encoder，因此下文将主要介绍 encoder 相关原理，而不是 decoder。

1.1.1　transformer 整体结构

Transformer 的核心是基于“自注意力（Self-Attention）”机制，旨在解决传统 RNN 和 LSTM 模型中序列数据处理的局限性。其结构分为两个主要部分：

- **编码器（Encoder）**
- **解码器（Decoder）**

a. 编码器（Encoder）

编码器的任务是接收输入序列，将其转换为一组表示（embedding）。编码器包含若干个堆叠的相同结构的层，每一层包含两个主要部分：

1. **自注意力机制（Self-Attention Mechanism）**：该机制允许每个单词关注输入序列中所有其他单词的信息，从而得到更丰富的表示。每个单词通过与其他单词的关系来更新其表示。
2. **前馈神经网络（Feedforward Neural Network）**：自注意力层后面跟着一个前馈神经网络，它对每个位置的表示独立地进行变换。

每一层都包括**层归一化（Layer Normalization）**和**残差连接（Residual Connections）**，帮助网络的训练更加稳定。

b. 解码器（Decoder）

解码器的作用是生成输出序列，解码器与编码器结构类似，也由若干层堆叠而成，每一层包含：

1. **自注意力机制**：与编码器中的自注意力机制相似。
2. **编码-解码注意力（Encoder-Decoder Attention）**：这是解码器特有的部分，它允许解码器关注编码器的输出，提取相关信息来生成每个词。
3. **前馈神经网络**：与编码器中的前馈神经网络类似。

解码器层同样采用层归一化和残差连接。
transformer 结构图如下所示：

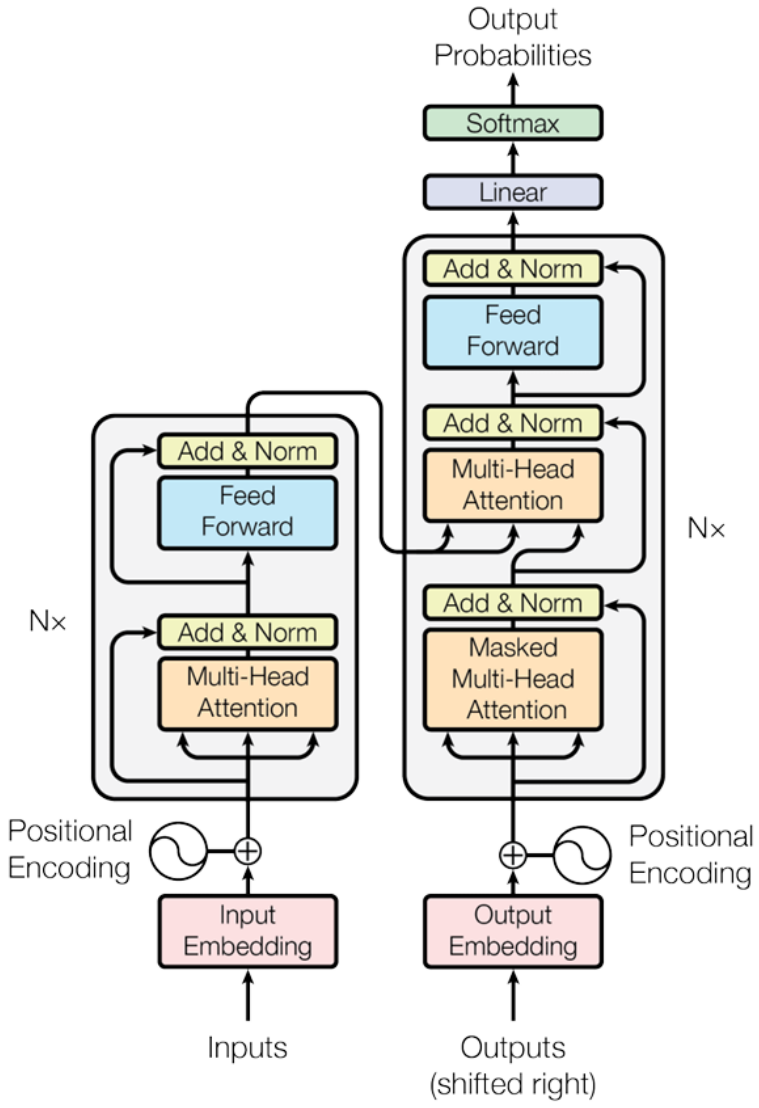

图 1: transformer 结构图

1.1.2 embedding 编码实现

由于 Transformer 不使用传统的 RNN 结构，因此它没有内建的顺序信息。为了解决这个问题，Transformer 通过**位置编码**来为每个词添加位置信息。位置编码是一种对序列中每个元素的位置信息进行编码的方式，通常使用正弦和余弦函数进行编码，确保不同位置的编码是唯一的。

位置编码的计算公式如下：

$$PE_{i,2d} = \sin\left(\frac{i}{10000^{2d/d_{\text{model}}}}\right) \tag{1}$$

$$PE_{i,2d+1} = \cos\left(\frac{i}{10000^{2d/d_{\text{model}}}}\right) \tag{2}$$

在 Transformer 模型中，位置编码与输入嵌入通过相加的方式结合，以将位置信息注入到输入的单词表示中。假设输入序列的长度为 N，模型的维度为 d_{model}，那么输入嵌入矩阵 $\mathbf{E}$ 和位置编码矩阵 $\mathbf{PE}$ 的维度都是 $N \times d_{\text{model}}$，相加的过程如下：

$$\mathbf{X} = \mathbf{E} + \mathbf{PE} \tag{3}$$

1.1.3 encoder block 的实现

一个 encoder block 由 Multi-Head Attention, Add & Norm, Feed Forward, Add & Norm 组成，transformer 结构共有 6 个 encoder block，输入嵌入向量需要依次经过 6 个 encoder block 后才算编码完毕。

a. 自注意力机制（Self-Attention）

自注意力机制是 Transformer 的核心，它使得模型可以在处理输入时，考虑到序列中所有位置的依赖关系，而不是仅仅依赖于序列中的前后文。自注意力机制的主要步骤包括：

1. **计算注意力权重**：通过计算输入的三个向量——**查询（Query）**、**键（Key）**和**值（Value）**之间的关系来决定哪些部分的信息应该被关注。这个过程通过计算点积来实现，得到的权重决定了各个单词的影响力。

2. **加权求和**：根据计算得到的权重，对值（Value）进行加权求和，得到最终的表示。

自注意力的具体计算过程如下：

首先，通过线性变换得到查询（Query）、键（Key）和值（Value）：

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{4}$$

然后，计算查询和键的相似度，得到注意力分数：

$$\text{Attention}(Q, K) = \frac{QK^T}{\sqrt{d_k}} \tag{5}$$

接着，应用 Softmax 函数计算注意力权重：

$$\text{Attention Weights} = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \tag{6}$$

最后，将注意力权重与值矩阵 V 相乘，得到输出：

$$\text{Output} = \text{Attention Weights} \times V \tag{7}$$

因此，自注意力的计算公式为：

$$\text{Self-Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{8}$$

自注意力示意图如下：

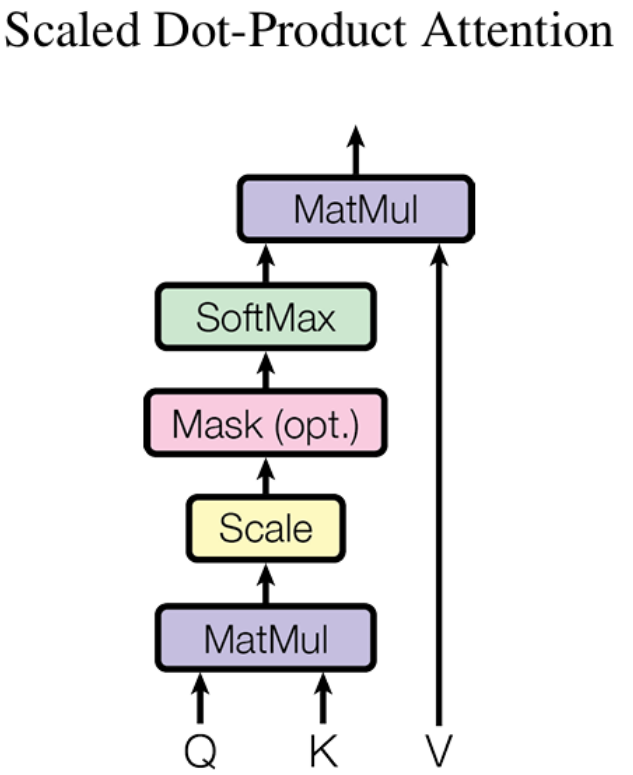

图 2: 自注意力示意图

b. 多头注意力机制（Multi-Head-Attention）

在多头注意力机制中，模型将查询、键和值分成多个头，每个头都有不同的权重矩阵。通过这种方式，模型可以学习到不同的注意力模式。

在多头注意力机制中，首先将查询、键和值矩阵分成 h 个头：

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V, \quad i \in \{1, 2, \ldots, h\} \tag{9}$$

然后，对于每个头 i，计算自注意力输出：

$$\text{Head}_i = \text{Self-Attention}(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)V_i \tag{10}$$

将所有头的输出拼接在一起，并通过线性变换映射回模型维度：

$$\text{Multi-Head Output} = \text{Concat}(\text{Head}_1, \text{Head}_2, \ldots, \text{Head}_h)W^O \tag{11}$$

通过这种机制，Transformer 能够在处理输入的每个元素时，考虑到整个输入序列的上下文信息，而不需要按顺序处理每个元素。

多头注意力示意图如下：

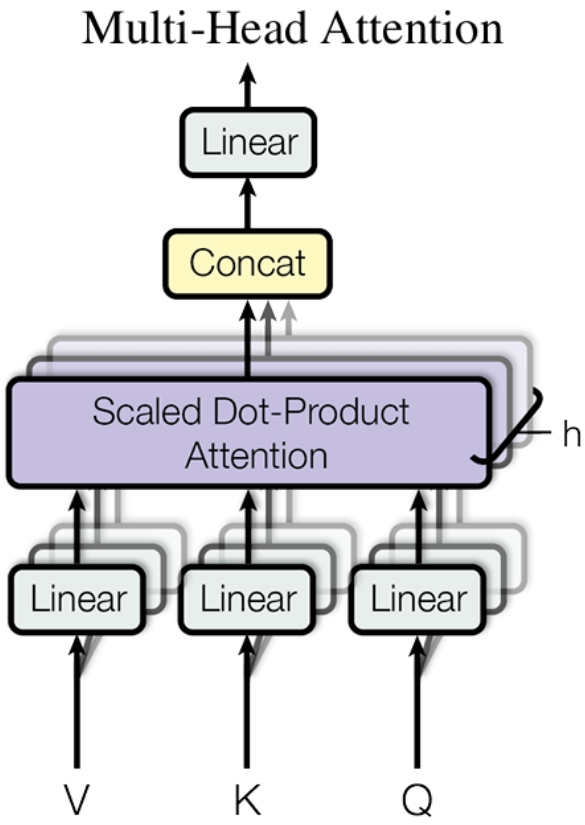

图 3: 多头注意力示意图

c. 前馈神经网络层（Feed Forward） Feed Forward 层比较简单，是一个两层的全连接层，第一层的激活函数为 Relu，第二层不使用激活函数。第一个线性变换将输入映射到一个更高维空间（通常是更大的维度，比如 2048），然后通过 ReLU 激活函数。接着，第二个线性变换将特征映射回原始维度。对应的公式如下:

$$\mathbf{Y} = \text{ReLU}(\mathbf{X}W_1 + b_1)W_2 + b_2 \tag{12}$$

d. 残差连接与归一化层（Add & Norm）

Add & Norm 层包括残差连接和层归一化。假设子层的输入是 $\mathbf{X}$，输出是 $\mathbf{Y}$，首先进行残差连接：

$$\mathbf{Y}_{\text{res}} = \mathbf{X} + \mathbf{Y} \tag{13}$$

然后，对结果进行层归一化：

$$\text{LayerNorm}(\mathbf{Z}) = \frac{\mathbf{Z} - \mu}{\sigma} \cdot \gamma + \beta \tag{14}$$

1.1.4 transformer 的作用

- **捕捉长程依赖**：自注意力机制能够捕捉长距离的依赖关系，这使得 Transformer 特别适合处理长序列，避免了传统 RNN 和 LSTM 在处理长序列时出现的梯度消失或爆炸问题。
- **并行化训练**：与 RNN 不同，Transformer 不需要逐步地处理序列中的每个元素，因此可以在训练时进行并行计算，显著提高训练效率。
- **高效性**：Transformer 的计算复杂度相对较低，能够处理更大的输入和更复杂的任务。

1.2 CLIP 代码逻辑

CLIP（Contrastive Language-Image Pre-Training）模型是一种多模态预训练神经网络，由 OpenAI 在 2021 年发布，是从自然语言监督中学习的一种有效且可扩展的方法。CLIP 在预训练期间学习执行广泛的任务，包括 OCR，地理定位，动作识别，并且在计算效率更高的同时优于公开可用的最佳 ImageNet 模型。

CLIP 模型采用了一个典型的双塔（dual-encoder）架构，其中一个塔用于处理图像输入，另一个塔用于处理文本输入。模型通过对比学习来优化这两个塔的输出，使得相同语义的文本和图像能够在同一空间中靠得更近。CLIP 模型架构如下所示：

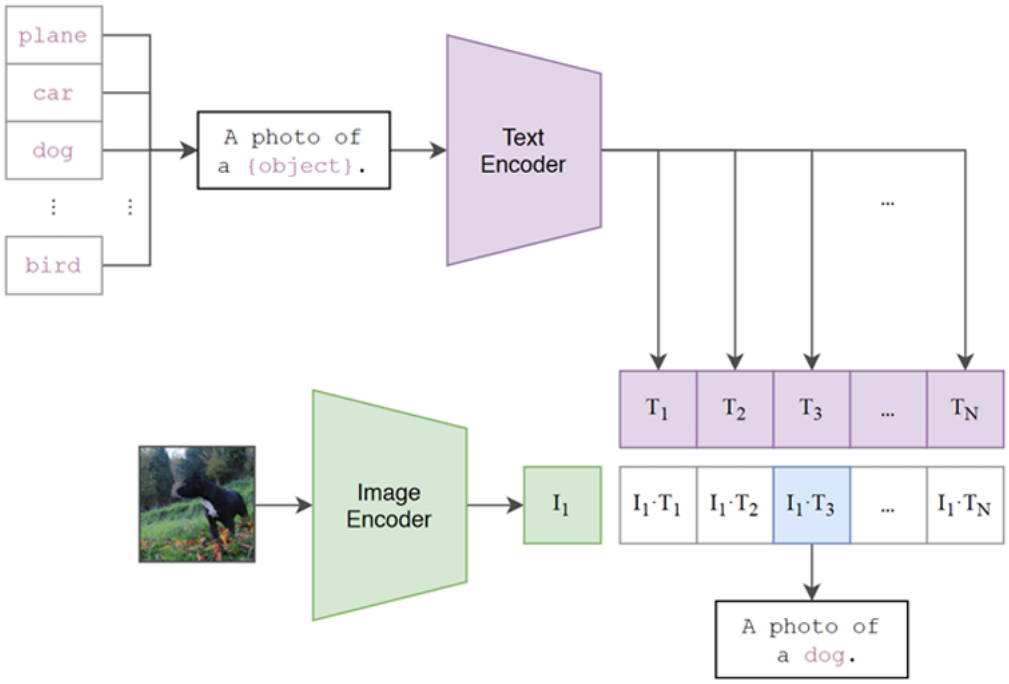

图 4: CLIP 示意图

1.2.1 图像编码器（Image Encoder）

图像编码器采用了卷积神经网络（如 ResNet）或者视觉变换器（ViT）。ViT 将图像分割成多个固定大小的块，并使用 Transformer 进行编码。

CLIP 中 ViT 的核心代码如下所示：

```python
def forward(self, x: torch.Tensor):
    x = self.conv1(x)  # shape = [*, width, grid, grid]
    x = x.reshape(x.shape[0], x.shape[1], -1)  # shape = [*, width, grid
        ↪ ** 2]
    x = x.permute(0, 2, 1)  # shape = [*, grid ** 2, width]
    x = torch.cat([self.class_embedding.to(x.dtype) + torch.zeros(x.shape
        ↪ [0], 1, x.shape[-1], dtype=x.dtype, device=x.device), x], dim
        ↪ =1)  # shape = [*, grid ** 2 + 1, width]
    x = x + self.positional_embedding.to(x.dtype)
    x = self.ln_pre(x)
    x = x.permute(1, 0, 2)  # NLD -> LND
    x = self.transformer(x)
    x = x.permute(1, 0, 2)  # LND -> NLD
    x = self.ln_post(x[:, 0, :])
```

1.2.2 文本编码器（Text Encoder）

文本编码器基于 Transformer 架构（如 BERT 或 GPT），将文本输入映射到一个高维向量空间。

CLIP 中文本编码的核心代码如下所示：

```python
def encode_text(self, text):
    x = self.token_embedding(text).type(self.dtype)  # [batch_size, n_ctx
        ↪ , d_model]
    x = x + self.positional_embedding.type(self.dtype)
    x = x.permute(1, 0, 2)  # NLD -> LND
    x = self.transformer(x)
    x = x.permute(1, 0, 2)  # LND -> NLD
    x = self.ln_final(x).type(self.dtype)
    # x.shape = [batch_size, n_ctx, transformer.width]
    # take features from the eot embedding (eot_token is the highest
        ↪ number in each sequence)
    x = x[torch.arange(x.shape[0]), text.argmax(dim=-1)] @ self.
        ↪ text_projection
```

2 zero-shot 分类效果

应用 CLIP 对 Caltech-101 进行 zero-shot 分类的实现程序为/CLIP-zero-shot/zero_shot.py, 实验中使用了三个不同的随机数进行数据集划分，准确率如下图所示：

```
Batch[25/28]:finished  Val Accuracy: 0.6562         time_costed:5min 23s    time_predicted:1min 1s
Batch[26/28]:finished  Val Accuracy: 0.8125         time_costed:5min 34s    time_predicted:0min 47s
Batch[27/28]:finished  Val Accuracy: 0.8125         time_costed:5min 46s    time_predicted:0min 34s
Batch[28/28]:finished  Val Accuracy: 0.7344         time_costed:5min 58s    time_predicted:0min 20s
Batch[29/28]:finished  Val Accuracy: 0.7568         time_costed:6min 7s    time_predicted:0min 7s
Final Val Accuracy: 0.7551 , seed: 26
```

图 5: seed:26

```
Batch[25/28]:finished  Val Accuracy: 0.6719         time_costed:6min 10s    time_predicted:1min 10s
Batch[26/28]:finished  Val Accuracy: 0.7344         time_costed:6min 23s    time_predicted:0min 54s
Batch[27/28]:finished  Val Accuracy: 0.7031         time_costed:6min 34s    time_predicted:0min 38s
Batch[28/28]:finished  Val Accuracy: 0.7344         time_costed:6min 44s    time_predicted:0min 23s
Batch[29/28]:finished  Val Accuracy: 0.7838         time_costed:6min 52s    time_predicted:0min 8s
Final Val Accuracy: 0.7706 , seed: 42
```

图 6: seed:42

```
Batch[25/28]:finished  Val Accuracy: 0.7812         time_costed:5min 1s     time_predicted:0min 57s
Batch[26/28]:finished  Val Accuracy: 0.7812         time_costed:5min 12s    time_predicted:0min 44s
Batch[27/28]:finished  Val Accuracy: 0.7031         time_costed:5min 24s    time_predicted:0min 31s
Batch[28/28]:finished  Val Accuracy: 0.7500         time_costed:5min 35s    time_predicted:0min 19s
Batch[29/28]:finished  Val Accuracy: 0.7568         time_costed:5min 44s    time_predicted:0min 6s
Final Val Accuracy: 0.7696 , seed: 37
```

图 7: seed:37

验证过程中记录的准确率波动如下图所示，最终的平均准确率为：76.51%

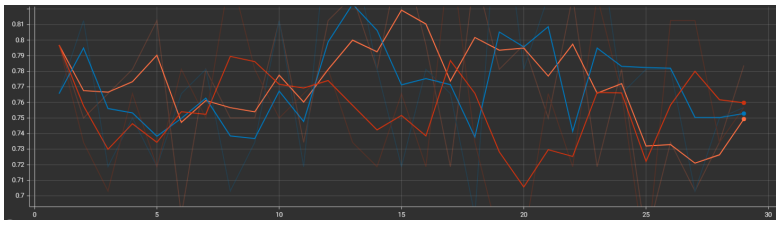

图 8: 准确率波动

3 CoOp 的实现与分类效果

CoOp 就是将 zero-shot CLIP 中的人工 prompt(“a photo of”) 换成可学习的 text embedding。CoOp 示意图如下所示：

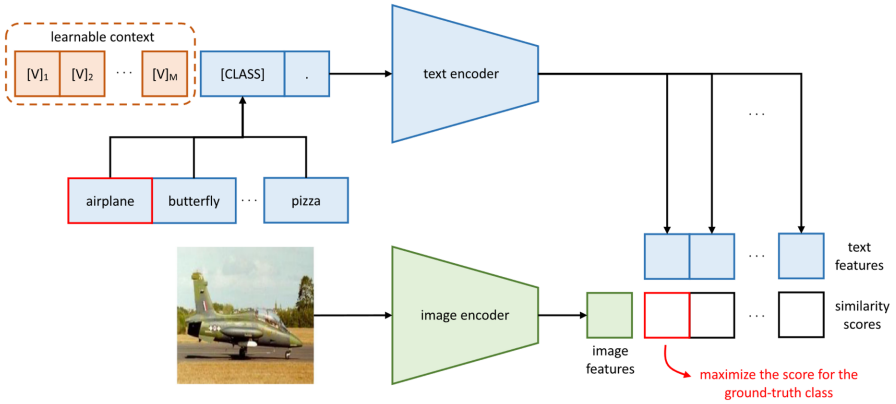

图 9: CoOp 示意图

CoOp 嵌入可学习 prompt 的关键代码如下所示：

```python
class AutoPrompt(nn.Module):
    def __init__(self, classnames, clip_model):
        super().__init__()
        n_cls = len(classnames)
        n_ctx = 16  # 嵌入数量
        dtype = clip_model.dtype
        ctx_dim = clip_model.ln_final.weight.shape[0]
        ctx_vectors = torch.empty(n_ctx, ctx_dim, dtype=dtype)
        if torch.cuda.is_available():
            ctx_vectors = ctx_vectors.cuda()
        nn.init.normal_(ctx_vectors, std=0.02)  # 方差为0.02的高斯分布
        prompt_prefix = "␣".join(["X"] * n_ctx)
        self.ctx = nn.Parameter(ctx_vectors)  # 训练时自动更新
        classnames = [name.replace("_", "␣") for name in classnames]
        prompts = [prompt_prefix + "␣" + name + "." for name in
            ↪ classnames]
        tokenized_prompts = torch.cat([clip.tokenize(p) for p in prompts
            ↪ ])
        if torch.cuda.is_available():
            tokenized_prompts = tokenized_prompts.cuda()
        with torch.no_grad():
            embedding = clip_model.token_embedding(tokenized_prompts).
                ↪ type(dtype)
        self.token_prefix = embedding[:, :1, :]
        self.token_suffix = embedding[:, 1+n_ctx:, :]
        self.tokenized_prompts = tokenized_prompts
        self.n_cls = n_cls
        self.n_ctx = n_ctx
    def forward(self):
        ctx = self.ctx
        if ctx.dim() == 2:
            ctx = self.ctx.unsqueeze(0).expand(self.n_cls, -1, -1)
        prefix = self.token_prefix
        suffix = self.token_suffix
        prompts = torch.cat(
            [
                prefix,  # (n_cls, 1, dim)
                ctx,  # (n_cls, n_ctx, dim)
                suffix,  # (n_cls, *, dim)
            ])
```

应用 CoOp 对 Caltech-101 进行 1、2、4-shot 分类的实现程序为/CLIP-CoOp/CoOp.py, 实验中使用了三个不同的随机数进行数据集划分。

训练集损失如下图所示：

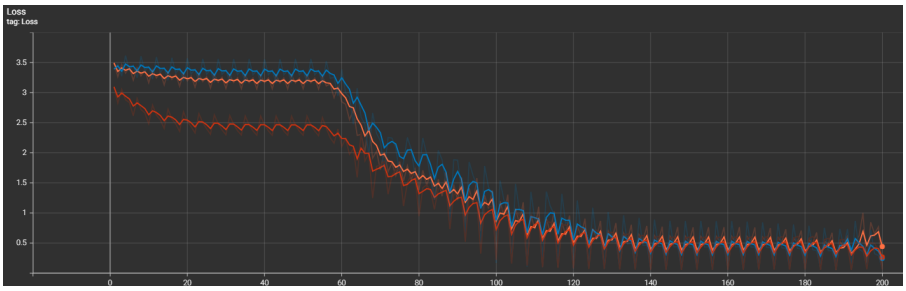

图 10: 1-shot

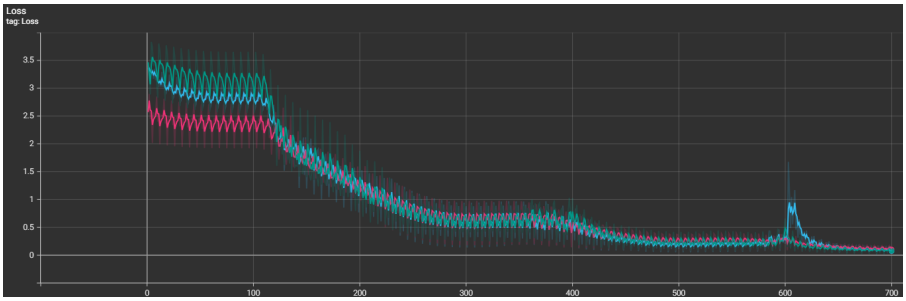

图 11: 2-shot

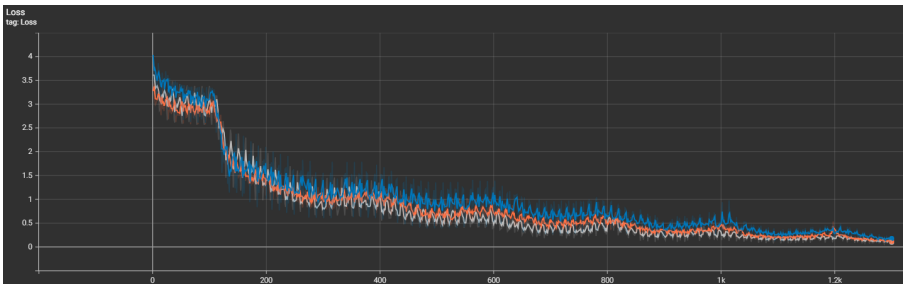

图 12: 4-shot

1、2、4-shot 分类的准确率分别为：60.10%、62.38%、65.52%. 验证集准确率变化如下图所示：

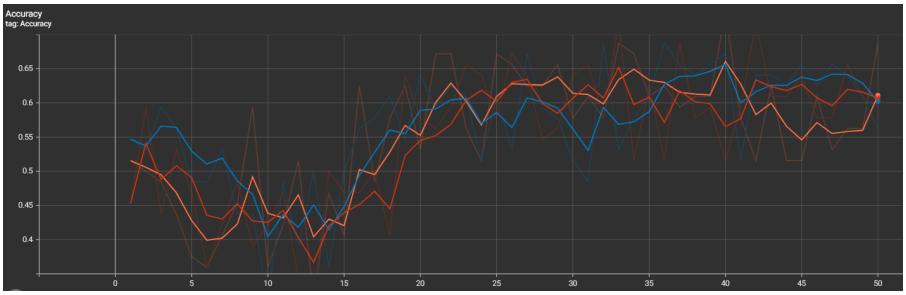

图 13: 1-shot

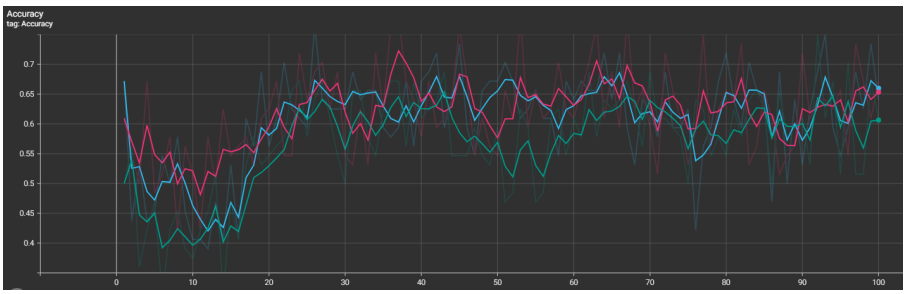

图 14: 2-shot

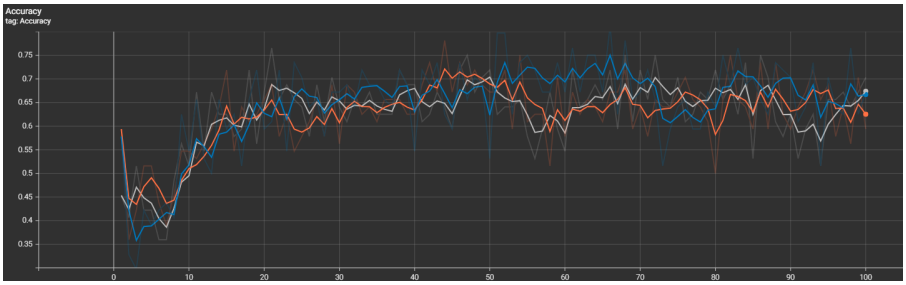

图 15: 4-shot

最终绘制的 CLIP 和 CoOp 的准确率如下图所示：

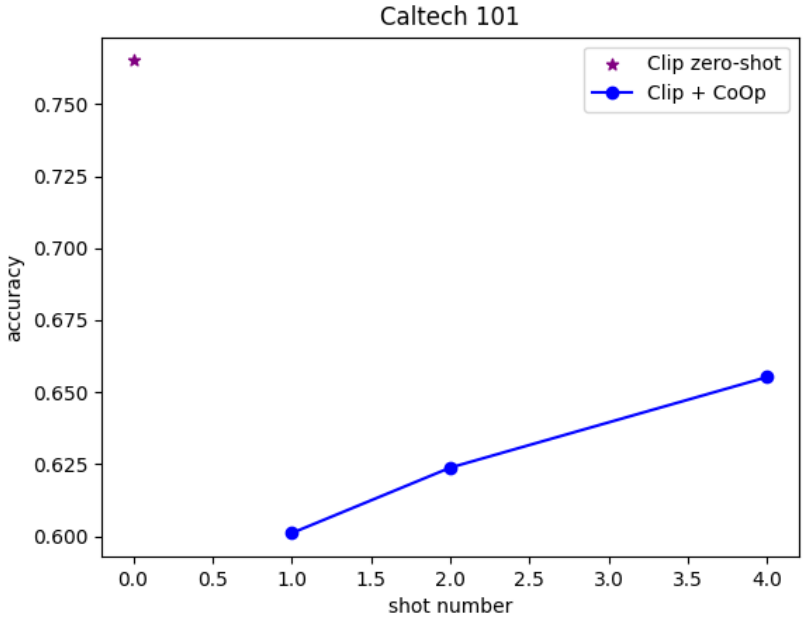

图 16: 准确率

4 附加项：transformer block 的改进方法

标准的 Transformer 模型采用全局自注意力机制，这导致在长序列的任务中计算复杂度较高。为了提高计算效率，本文提出了一种基于局部自注意力机制的修改方法，其中每个位置的表示仅依赖于其周围的局部上下文，从而减少了计算复杂度。

4.1 背景问题

标准的自注意力机制需要计算序列中每对位置之间的相似度,这使得计算复杂度为$O(N^2)$，其中 N 是序列长度。对于长序列，计算成本非常高。

4.2 提出修改方法：局部自注意力机制

在局部自注意力机制中，每个位置的表示仅与其局部上下文进行交互，计算复杂度从 $O(N^2)$ 降低到 $O(Nk)$，其中 k 是窗口大小。具体来说，对于每个查询位置 i，我们只考虑它与窗口内其他位置的注意力，而不需要计算全序列范围的注意力。

4.3 具体修改方法

4.3.1 局部注意力窗口

对于每个位置 i，我们只关注从位置 $i-k$ 到 $i+k$ 范围内的位置，其中 k 是窗口的大小。也就是说，位置 i 的注意力仅计算它和周围 k 个位置的相似度。

4.3.2 计算步骤

修改后的自注意力计算包括以下几个步骤：

1. 选择局部窗口：对于每个位置 i，确定其局部窗口范围 $[i-k, i+k]$，其中 k 是一个超参数，表示局部窗口的大小。

2. 计算注意力权重：对于每个位置 i 和位置 j（$j \in [i-k, i+k]$），计算它们之间的注意力权重：

$$\alpha_{ij} = \frac{\exp(\mathbf{q}_i^T \mathbf{k}_j)}{\sum_{j' \in [i-k, i+k]} \exp(\mathbf{q}_i^T \mathbf{k}_{j'})} \tag{15}$$

其中：$\mathbf{q}_i$ 是位置 i 的查询向量。$\mathbf{k}_j$ 是位置 j 的键向量。

3. 计算加权和：对于每个位置 i，计算它的输出表示为窗口内位置的加权和：

$$\mathbf{z}_i = \sum_{j \in [i-k, i+k]} \alpha_{ij} \mathbf{v}_j \tag{16}$$

其中：$\mathbf{v}_j$ 是位置 j 的值向量。

4.3.3 影响与优化

通过这种局部自注意力机制，模型只需计算一个固定大小窗口内的注意力，而不需要计算全序列的注意力，从而减少了计算复杂度。

实验尝试了将注意力窗口大小定义为 8，最终结果表明：训练时间有所减小，并且准确率没有明显下降。