基于 Vision Transformer 的分类任务系统设计及其探索实验

摘要

虽然 Transformer 架构已成为 NLP 任务的事实标准，但它在 CV 中的应用仍然有限。在视觉上，注意力要么与卷积网络结合使用，要么用于替换卷积网络的某些组件，同时保持其整体结构。ViT 证明了这种对 CNNs 的依赖是不必要的，直接应用于图像块序列的纯 Transformer 可以很好地执行图像分类任务。当对大量数据进行预训练并迁移到多个中小型图像识别基准时 (ImageNet、CIFAR-100、VTAB 等)，与 SOTA 的 CNN 相比，VisionTransformer(ViT) 可获得更优异的结果，同时仅需更少的训练资源。故本项目以有监督方式训练了图像分类模型。还通过微调预训练模型，实现了图片分类的迁移学习，大大提升了分类系统正确率；并且探索了不同输出 Token 作为分类器输入进行分类时的效果差异；以及探索将 Token 以各种方式混合，作为分类器输入时分类效果差异。

关键词： NLP；VisionTransformer；卷积网络；图片分类；预训练模型

1 引言

基于自注意力的架构，尤其是 Transformer [7]，已成为 NLP 中的首选模型。由于 Transformers 的计算效率和可扩展性，训练具有超过 100B 个参数的、前所未有的模型成为了可能。随着模型和数据集的增长，仍未表现出饱和的迹象。然而，在 CV 中，卷积架构仍然占主导地位。受到 NLP 成功的启发，多项工作尝试将类似 CNN 的架构与自注意力相结合，有些工作完全取代了卷积。后一种模型虽然理论上有效，但由于使用了特定的注意力模式，尚未在现代硬件加速器上有效地扩展。因此，在大规模图像识别中，经典的类 ResNet [4] 架构仍是最先进的。故 ViT [3] 尝试将标准 Transformer 直接应用于图像，并尽可能减少修改。它将图像拆分为块 (patch)，并将这些图像块的线性嵌入序列作为 Transformer 的输入。

为了展示 ViT 的效果，故本项目以有监督方式训练了图像分类模型。复现了论文中的内容。还通过微调预训练模型，实现了图片分类的迁移学习，大大提升了分类系统正确率，在披萨、牛排和寿司图像数据集中的分类准确度达到了 99%。并且探索了不同输出 Token 作为分类器输入进行分类时的效果差异验证了论文中添加 [CLS] Token 的必要性和有效性，以及揭示了不同位置的 Patch 融合到的信息是有区别的，Patch 越靠中间，能融合到的信息越多；以及探索将 Token 以各种方式混合，作为分类器输入时分类效果差异。其中混合方式包括：直接加合混合、平均混合、加权混合。实验结果显示，使用加权混合具有更好的效果。

2 相关工作

Transformers 是由 Vaswani 等人提出的机器翻译方法，并已成为许多 NLP 任务中的 SOTA 方法。基于大型 Transformers 的模型通常在大型语料库 (corpus) 上预训练，然后根据所需的下游任务 (down-streamtasks) 进行微调 (finetune)。注意，BERT [2] 使用去噪自监督预训练任务，而 GPT 系列使用语言建模 (LM) 作为预训练任务。

应用于图像的简单自注意力要求每个像素关注所有其他像素。由于像素数量的二次方成本，其无法放缩到符合实际的输入尺寸。因此，曾经有研究者尝试过几种近似方法以便于在图像处理中应用 Transformer。Parmar 等人只在每个 query 像素的局部邻域而非全局应用自注意力，这种局部多头点积自注意力块完全可以代替卷积。在另一种工作中，稀疏 Transformer 采用可放缩的全局自注意力，以便适用于图像。衡量注意力的另一种方法是将其应用于大小不同的块中，在极端情况下仅沿单个轴。许多这种特殊的注意力架构在 CV 任务上显示出很好的效果，但是需要在硬件加速器上有效地实现复杂的工程。

与 ViT 最相关的是 Cordonnier [1] 等人的模型，该模型从输入图像中提取 2×2 大小的块，并在顶部应用完全的自注意力。该模型与 ViT 非常相似，但 ViT 的工作进一步证明了大规模的预训练使普通的 Transformers 能够与 SOTA 的 CNNs 竞争 (甚至更优)。此外，Cordonnier 等人使用 2×2 像素的小块，使模型只适用于小分辨率图像，而 ViT 也能处理中分辨率图像。将 CNN 与自注意力的形式相结合有很多有趣点，例如增强用于图像分类的特征图，或使用自注意力进一步处理 CNN 的输出，如用于目标检测、视频处理、图像分类，无监督目标发现，或统一文本视觉任务。

另一个最近的相关模型是图像 GPT(iGPT) [6]，它在降低图像分辨率和颜色空间后对图像像素应用 Transformers。该模型以无监督的方式作为生成模型进行训练，然后可以对结果表示进行微调或线性探测以提高分类性能，在 ImageNet 上达到 72% 的最大精度。

ViT 的工作增加了在比标准 ImageNet 数据集更大尺度上探索图像识别的论文的数量。使用额外的数据源可以在标准基准上取得 SOTA 的成果。此外，Sun 等人研究了 CNN 性能如何随数据集大小而变化，Kolesnikov、Djolonga 等人从 ImageNet-21k 和 JFT-300M [5] 等大规模数据集对 CNN 迁移学习进行了实证研究。ViT 也关注后两个数据集，但是是训练 Transformers 而非以前工作中使用的基于 ResNet 的模型。

3 本文方法

3.1 本文方法概述

本项目以有监督方式训练了图像分类模型。使用 PyTorch 复现了论文中 ViT 的架构。还通过微调预训练模型，实现了图片分类的迁移学习。本项目重点探索了不同输出 Token 作为分类器输入进行分类时的效果差异验证了论文中添加 [CLS] Token 的必要性和有效性，以及揭示了不同位置的 Patch 融合到的信息是有区别的，Patch 越靠中间，能融合到的信息越多；以及参考 ResNet 的方法，探索将 Token 以各种方式混合，混合方式包括：直接加合混合、平均混合、加权混合。作为分类器输入时分类效果差异。

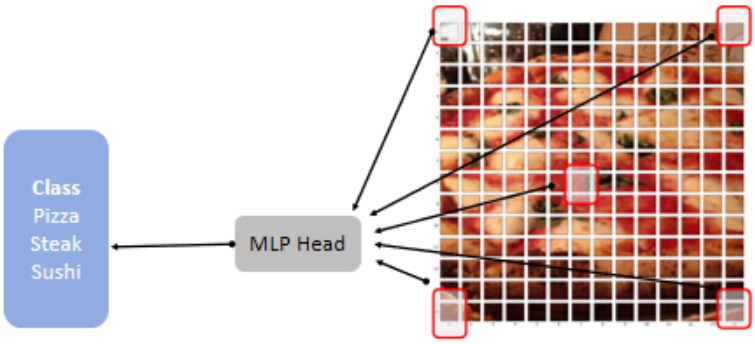


图 1. Token 探索示意图

3.2 复现核心公式

这串公式是 ViT 的核心公式，它十分简明的概括了整个 ViT 的数据流向和 ViT 的结构。从这个公式可以看出 ViT 的大致数据流向是：ViT 会先将图片按照 16x16 的大小进行 Patch Embedding 并展成 patch 序列，然后给 patch 添加位置编码和 CLS 字符，接着将 Embedding 完的序列传入标准 Transformer Encoder 模块中进行编码；最后利用编码后的 CLS 输入分类器中进行分类输出。这个公式还能反映出 ViT 的结构，下面的图展示了公式和 ViT 模块的对应情况。

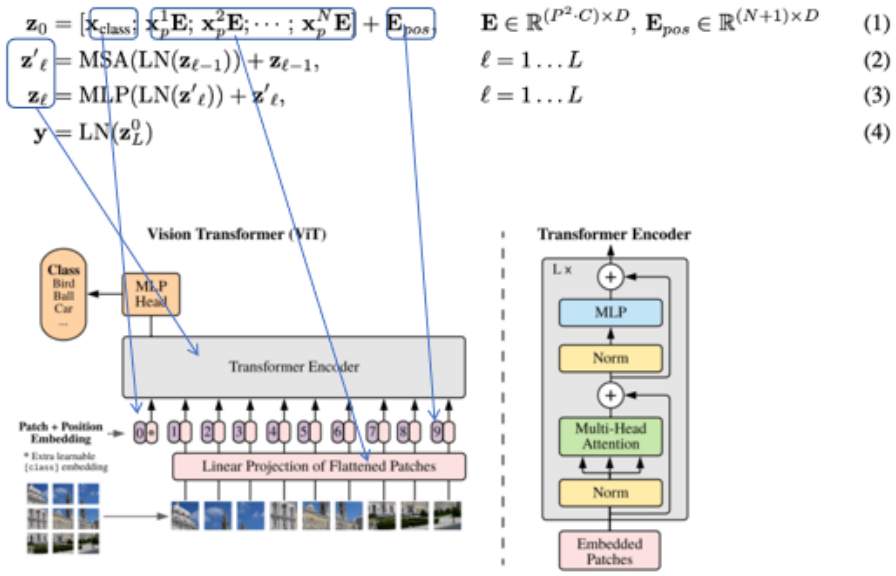


图 2. 公式和 ViT 模块的对应情况

4 复现细节

4.1 与已有开源代码对比

本项目主要参考了 ViT 公开的源码 https://github.com/google-research/vision_transformer，与源码不同的是，本项目在源码的基础上，对各模型的训练流程进行了模块化封装，非常适合快速迭代探索和运行实验。

并且本项目在源码基础上进行修改，实现了探索不同输出 Token 作为分类器输入进行分类时的效果差异的目的。

封装的主要模块：

- data_setup.py - 用于准备和下载数据的文件。

```
import os
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

NUM_WORKERS = os.cpu_count()
def create_dataloaders(
        train_dir: str,
        test_dir: str,
        transform: transforms.Compose,
        batch_size: int,
        num_workers: int = NUM_WORKERS
):
    train_data = datasets.ImageFolder(train_dir, transform=transform)
    class_names = train_data.classes
    train_dataloader = DataLoader(
        train_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
    )
    test_dataloader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        pin_memory=True,
    )
    return train_dataloader, test_dataloader, class_names
```

- engine.py - 包含各种训练函数的文件。

```
import torch
from tqdm.auto import tqdm
from typing import Dict, List, Tuple
def train_step(model: torch.nn.Module,
               dataloader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
```

```
              optimizer: torch.optim.Optimizer,
              device: torch.device) -> Tuple[float, float]:
    model.train()

    train_loss, train_acc = 0, 0

    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        y_pred = model(X)
        loss = loss_fn(y_pred, y)
        train_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        y_pred_class = torch.argmax(torch.softmax(y_pred, dim=1), dim=1)
        train_acc += (y_pred_class == y).sum().item() / len(y_pred)
    train_loss = train_loss / len(dataloader)
    train_acc = train_acc / len(dataloader)
    return train_loss, train_acc

def test_step(model: torch.nn.Module,
              dataloader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              device: torch.device) -> Tuple[float, float]:
    model.eval()
    test_loss, test_acc = 0, 0
    with torch.inference_mode():
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.to(device), y.to(device)
            test_pred_logits = model(X)
            loss = loss_fn(test_pred_logits, y)
            test_loss += loss.item()
            test_pred_labels = test_pred_logits.argmax(dim=1)
            test_acc += ((test_pred_labels == y).sum().item() /
                len(test_pred_labels))
    test_loss = test_loss / len(dataloader)
    test_acc = test_acc / len(dataloader)
    return test_loss, test_acc

def train(model: torch.nn.Module,
          train_dataloader: torch.utils.data.DataLoader,
          test_dataloader: torch.utils.data.DataLoader,
          optimizer: torch.optim.Optimizer,
```

```
        loss_fn: torch.nn.Module,
        epochs: int,
        device: torch.device) -> Dict[str, List]:
    results = {"train_loss": [],
               "train_acc": [],
               "test_loss": [],
               "test_acc": []
               }
    model.to(device)
    for epoch in tqdm(range(epochs)):
        train_loss, train_acc = train_step(model=model,
                                           dataloader=train_dataloader,
                                           loss_fn=loss_fn,
                                           optimizer=optimizer,
                                           device=device)
        test_loss, test_acc = test_step(model=model,
                                        dataloader=test_dataloader,
                                        loss_fn=loss_fn,
                                        device=device)
        print(
            f"Epoch: {epoch + 1} | "
            f"train_loss: {train_loss:.4f} | "
            f"train_acc: {train_acc:.4f} | "
            f"test_loss: {test_loss:.4f} | "
            f"test_acc: {test_acc:.4f}"
        )
        results["train_loss"].append(train_loss)
        results["train_acc"].append(train_acc)
        results["test_loss"].append(test_loss)
        results["test_acc"].append(test_acc)
    return results
```

- model.py - 用于创建 PyTorch 模型的文件。

```
import matplotlib.pyplot as plt
import torch
import torchvision
from torch import nn
from torchvision import transforms
from torchinfo import summary
import data_setup, engine
from helper_functions import download_data, set_seeds, plot_loss_curves

```

```
class PatchEmbedding(nn.Module):
    def __init__(self,
                 in_channels:int=3,
                 patch_size:int=16,
                 embedding_dim:int=768):
        super().__init__()
        self.patcher = nn.Conv2d(in_channels=in_channels,
                                 out_channels=embedding_dim,
                                 kernel_size=patch_size,
                                 stride=patch_size,
                                 padding=0)
        self.flatten = nn.Flatten(start_dim=2, end_dim=3)
    def forward(self, x):
        x_patched = self.patcher(x)
        x_flattened = self.flatten(x_patched)
        return x_flattened.permute(0, 2, 1)

class MultiheadSelfAttentionBlock(nn.Module):
    def __init__(self,
                 embedding_dim:int=768,
                 num_heads:int=12,
                 attn_dropout:float=0):
        super().__init__()
        self.layer_norm = nn.LayerNorm(normalized_shape=embedding_dim)
        self.multihead_attn = nn.MultiheadAttention(embed_dim=embedding_dim,
                                                    num_heads=num_heads,
                                                    dropout=attn_dropout,
                                                    batch_first=True)
    def forward(self, x):
        x = self.layer_norm(x)
        attn_output, _ = self.multihead_attn(query=x,
                                             key=x,
                                             value=x,
                                             need_weights=False)
        return attn_output
class MLPBlock(nn.Module):
    def __init__(self,
                 embedding_dim:int=768,
                 mlp_size:int=3072,
                 dropout:float=0.1):
        super().__init__()
        self.layer_norm = nn.LayerNorm(normalized_shape=embedding_dim)
        self.mlp = nn.Sequential(
```

```
            nn.Linear(in_features=embedding_dim,
                      out_features=mlp_size),
            nn.GELU(),
            nn.Dropout(p=dropout),
            nn.Linear(in_features=mlp_size,
                      out_features=embedding_dim),
            nn.Dropout(p=dropout)
        )
    def forward(self, x):
        x = self.layer_norm(x)
        x = self.mlp(x)
        return x
class TransformerEncoderBlock(nn.Module):
    def __init__(self,
                 embedding_dim:int=768,
                 num_heads:int=12,
                 mlp_size:int=3072,
                 mlp_dropout:float=0.1,
                 attn_dropout:float=0):
        super().__init__()
        self.msa_block =
            MultiheadSelfAttentionBlock(embedding_dim=embedding_dim,
                                        num_heads=num_heads,
                                        attn_dropout=attn_dropout)
        self.mlp_block = MLPBlock(embedding_dim=embedding_dim,
                                  mlp_size=mlp_size,
                                  dropout=mlp_dropout)

    def forward(self, x):
        x = self.msa_block(x) + x
        x = self.mlp_block(x) + x
        return x
class ViT(nn.Module):
    def __init__(self,
                 img_size:int=224,
                 in_channels:int=3,
                 patch_size:int=16,
                 num_transformer_layers:int=12,
                 embedding_dim:int=768,
                 mlp_size:int=3072,
                 num_heads:int=12,
                 attn_dropout:float=0,
                 mlp_dropout:float=0.1,
```

```
                embedding_dropout:float=0.1,
                num_classes:int=1000):
        super().__init__()

        self.num_patches = (img_size * img_size) // patch_size**2
        self.class_embedding = nn.Parameter(data=torch.randn(1, 1,
            embedding_dim),
                                            requires_grad=True)
        self.position_embedding = nn.Parameter(data=torch.randn(1,
            self.num_patches+1, embedding_dim),
                                               requires_grad=True)
        self.embedding_dropout = nn.Dropout(p=embedding_dropout)
        self.patch_embedding = PatchEmbedding(in_channels=in_channels,
                                              patch_size=patch_size,
                                              embedding_dim=embedding_dim)

        self.transformer_encoder =
            nn.Sequential(*[TransformerEncoderBlock(embedding_dim=embedding_dim,
                                    num_heads=num_heads,
                                    mlp_size=mlp_size,
                                    mlp_dropout=mlp_dropout)
                                    for _ in range(num_transformer_layers)]
                                                )
        self.classifier = nn.Sequential(
            nn.LayerNorm(normalized_shape=embedding_dim),
            nn.Linear(in_features=embedding_dim,
                      out_features=num_classes)
        )
    def forward(self, x):
        batch_size = x.shape[0]
        class_token = self.class_embedding.expand(batch_size, -1, -1)
        x = self.patch_embedding(x)
        x = torch.cat((class_token, x), dim=1)
        x = self.position_embedding + x
        x = self.embedding_dropout(x)
        x = self.transformer_encoder(x)
        x = self.classifier(x[:, 0])
        return x

if __name__ == '__main__':
    set_seeds()
    random_image_tensor = torch.randn(1, 3, 224, 224)
    vit = ViT(num_classes=3)
```

```
    print(vit(random_image_tensor))
```

- train.py - 用于利用所有其他文件并训练目标 PyTorch 模型的文件。

```
import matplotlib.pyplot as plt
import torch
import torchvision
from torch import nn
from torchvision import transforms
from torchinfo import summary
import data_setup, engine
from helper_functions import download_data, set_seeds, plot_loss_curves
import model, DATASET
if __name__ == '__main__':

    train_dataloader, test_dataloader, class_names = DATASET.getdata()
    vit = model.ViT(num_classes=len(class_names))
    optimizer = torch.optim.Adam(params=vit.parameters(),
                                 lr=1e-3,
                                 betas=(0.9, 0.999),
                                 weight_decay=0.3)

    loss = nn.CrossEntropyLoss()

    set_seeds()

    results = engine.train(model=vit,
                           train_dataloader=train_dataloader,
                           test_dataloader=test_dataloader,
                           optimizer=optimizer,
                           loss_fn=loss,
                           epochs=10,
                           device='cuda')

    from helper_functions import plot_loss_curves

    plot_loss_curves(results)
    plt.show()
```

- utils.py - 专用于有用实用功能的文件, 例如快速打印训练结果, 快速保存模型权重。

```
import matplotlib.pyplot as plt
import torch
from pathlib import Path
```

```
4
5  def plot_loss_curves(results):
6      loss = results["train_loss"]
7      test_loss = results["test_loss"]
8      accuracy = results["train_acc"]
9      test_accuracy = results["test_acc"]
10     epochs = range(len(results["train_loss"]))
11     plt.figure(figsize=(15, 7))
12     plt.subplot(1, 2, 1)
13     plt.plot(epochs, loss, label="train_loss")
14     plt.plot(epochs, test_loss, label="test_loss")
15     plt.title("Loss")
16     plt.xlabel("Epochs")
17     plt.legend()
18     plt.subplot(1, 2, 2)
19     plt.plot(epochs, accuracy, label="train_accuracy")
20     plt.plot(epochs, test_accuracy, label="test_accuracy")
21     plt.title("Accuracy")
22     plt.xlabel("Epochs")
23     plt.legend()
24
25 def save_model(model: torch.nn.Module,
26                target_dir: str,
27                model_name: str):
28   target_dir_path = Path(target_dir)
29   target_dir_path.mkdir(parents=True,
30                         exist_ok=True)
31   assert model_name.endswith(".pth") or model_name.endswith(".pt"),
         "model_name should end with '.pt' or '.pth'"
32   model_save_path = target_dir_path / model_name
33   print(f"[INFO] Saving model to: {model_save_path}")
34   torch.save(obj=model.state_dict(),
35              f=model_save_path)
```

4.2 数据集准备

本项目用的数据集来自 Github 中开源的披萨、牛排和寿司图像数据集，该数据集一共有 3 个类别，每一类都有 100 张图片供训练，由于 ViT 论文中使用的图片大小为 3x224x224，故需要对数据进行预处理，处理完后，将数据装入 DataLoader 中，并设置 batchsize，论文中使用的是 4096，考虑到本人电脑配置，我这里设置成了 32。

4.3 图像切分过程

我这里使用 plt 可视化了图像切分的过程，切分出来的效果如下，左边这是未切分前的照片，然后按照 16x16 进行切分的效果，可以看到，一张图片切分后，会被分成 196 个小块，切分后的图片可以视作一个 14x14 的矩阵。

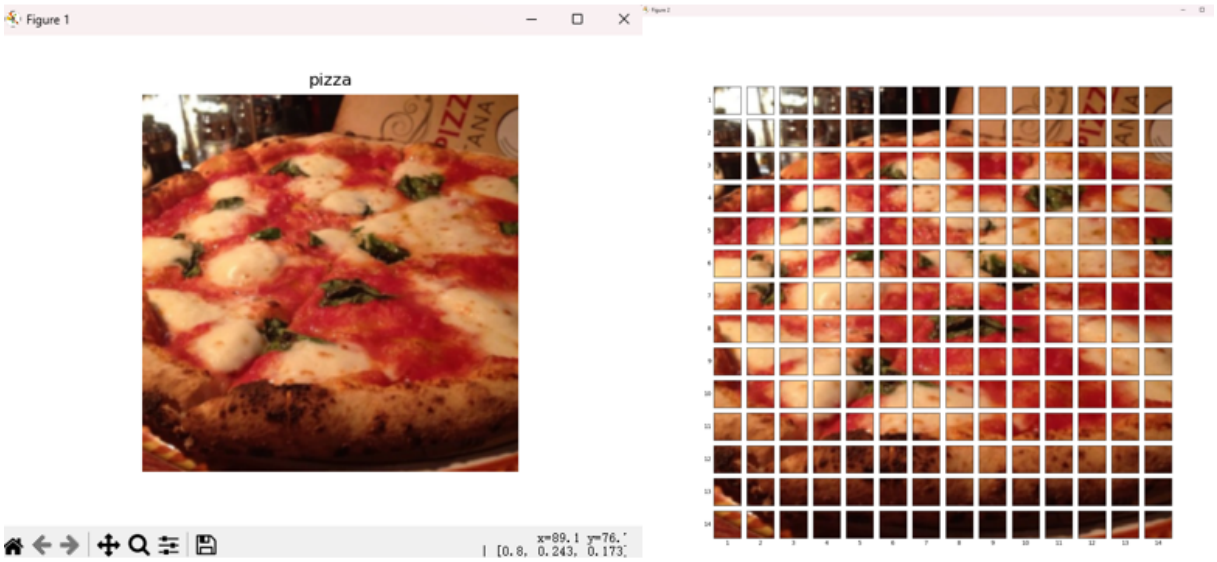


图 3. 图像切分过程可视化展示

实际上在论文中，切分操作和 embedding 是可以通过卷积操作同时完成的，具体来说，我们可以利用 768 个大小 16x16 的卷积核和 16 的卷积步长，来实现将图像转换成为 768x14x14 的特征图，这里之所以用 768 个卷积核是因为论文中要将每一个 patch 嵌入为 768 个维度的向量。我们使用 plt 可视化其中的五个卷积后的特征图，可以看到这个 14x14 的特征矩阵非常抽象，是人类无法理解的。

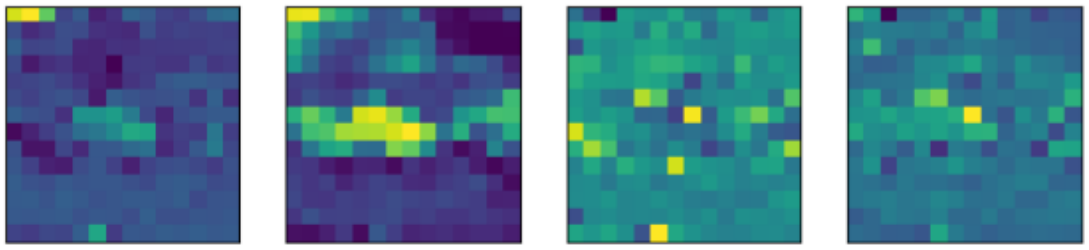


图 4. 卷积后的特征图节选

Transformer 的输入需要的是一维的序列形式，但图像经过卷积后的格式是一个二维格式，可以利用 nn.Flatten 来将二维格式展平成一维，展平后的维度是 196x768，展平的过程就像下面这个图片所展示的这样。

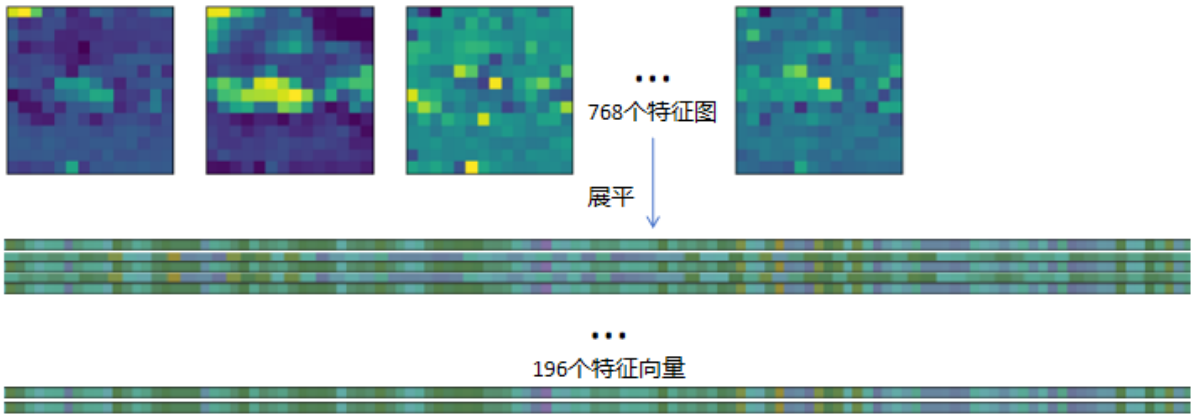


图 5. 展平过程展示

4.4 创新点

图片分类的迁移学习，可以大大提升了分类系统正确率；我们还可以探索不同输出 Token 作为分类器输入进行分类时的效果差异；以及探索将 Token 以各种方式混合，作为分类器输入时的分类效果差异。

4.4.1 迁移学习

为了解决分类效果差的问题，我们可以考虑使用迁移学习，使用别人训练好的预训练权重，再此基础上进行训练。这边使用的预训练权重是 torchvision 里内置的 ViT_B_16 预训练权重，这个权重是在 ImageNet-1k 上进行预训练的。通过冻结参数，我们只训练模型的输出层，总参数只有两千多个而已。比起之前的全调参，参数量少了非常多。下面是结果图，可以看到，使用预训练模型训练的效果非常好，正确率到达了 93% 左右，已经非常理想了。

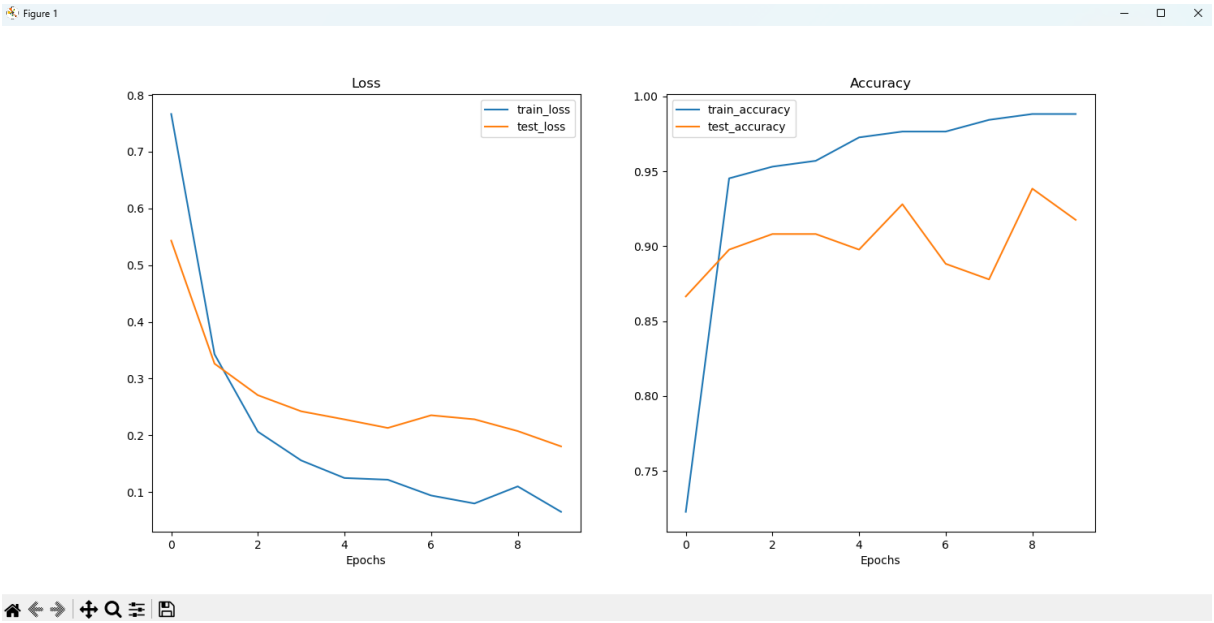


图 6. 卷积后的特征图节选

4.4.2 探索使用不同的 Token 进行分类输入

平时做分类任务时，都是用 [CLS] 作为分类器的输入，但我将探索使用别的 Token 作为分类器的输入进行训练，并观察效果。本次实验选用了第 1、14、91、182、196 个 Token 用作分类器输入，这些 Token 分别对应图中所标记的位置。

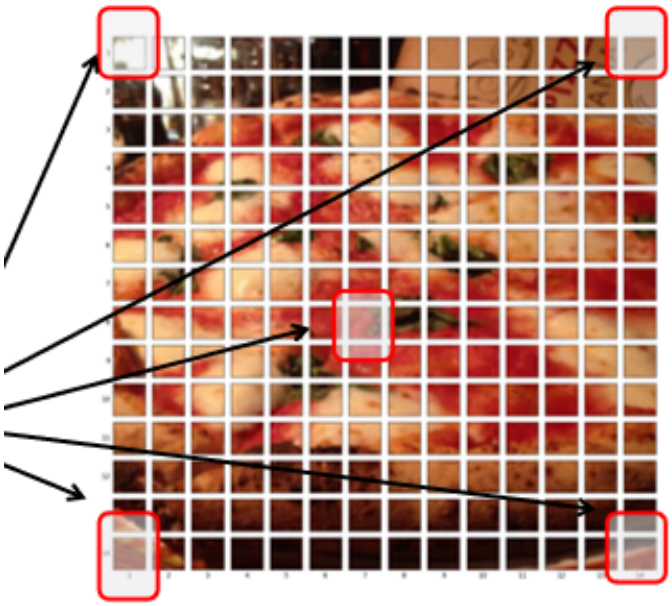


图 7. 卷积后的特征图节选

4.4.3 探索使用混合的 Token 进行分类输入

混合方式包括：直接加合混合、平均混合、加权混合。直接加合混合是指将所有的 Token 加和，作为分类器的输入；平均混合是将所有的 Token 取平均数，作为分类器的输入；加权混合比较复杂，它是将除了 CLS 之外的 Token 进行加权求和，然后和 CLS 一起，作为分类器的输入，其中权重是可学习的。

5 实验结果分析

5.1 探索使用不同的 Token 进行分类输入的实验分析

第 1 个 Token 在训练 10 次后，最高正确率只有 60% 左右，不过正确率曲线呈现上涨趋势

图 8. 实验结果示意

第 14 个的最高正确率只有 56% 左右。并且训练集准确率和验证集差距太大，存在严重过拟合现象。

图 9. 实验结果示意

第 91 个 Token 效果还行，说明中间的 Patch 还是有融合到了不少全局信息的

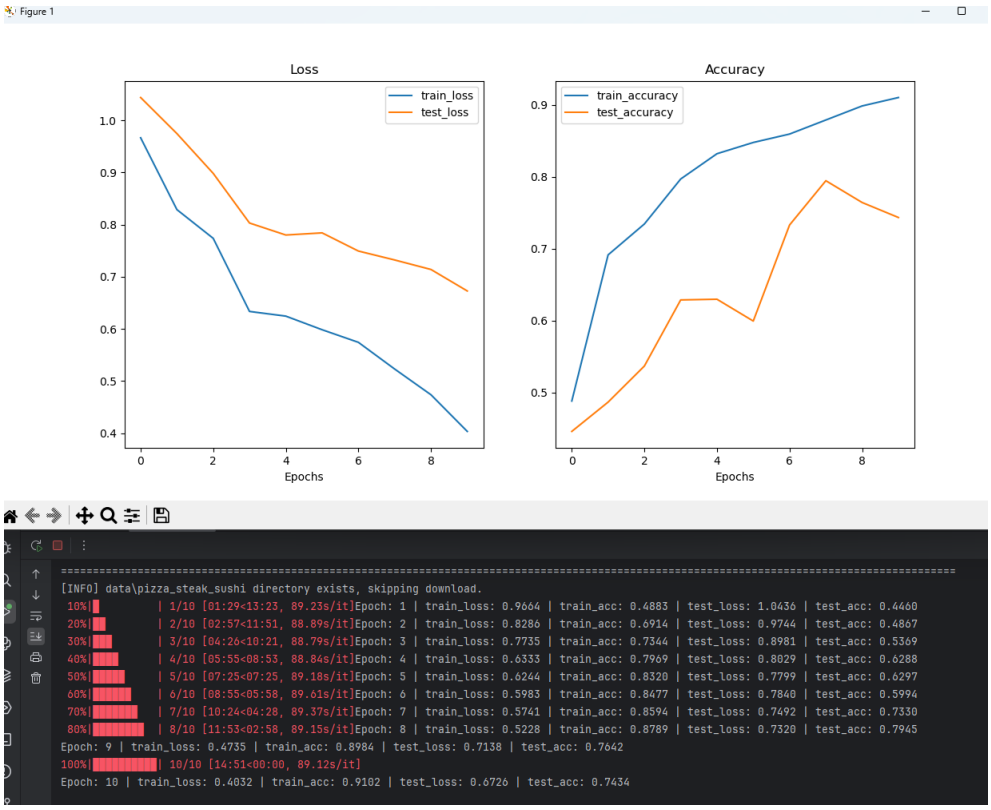


图 10. 实验结果示意

第 182 和 196 个效果都很一般。

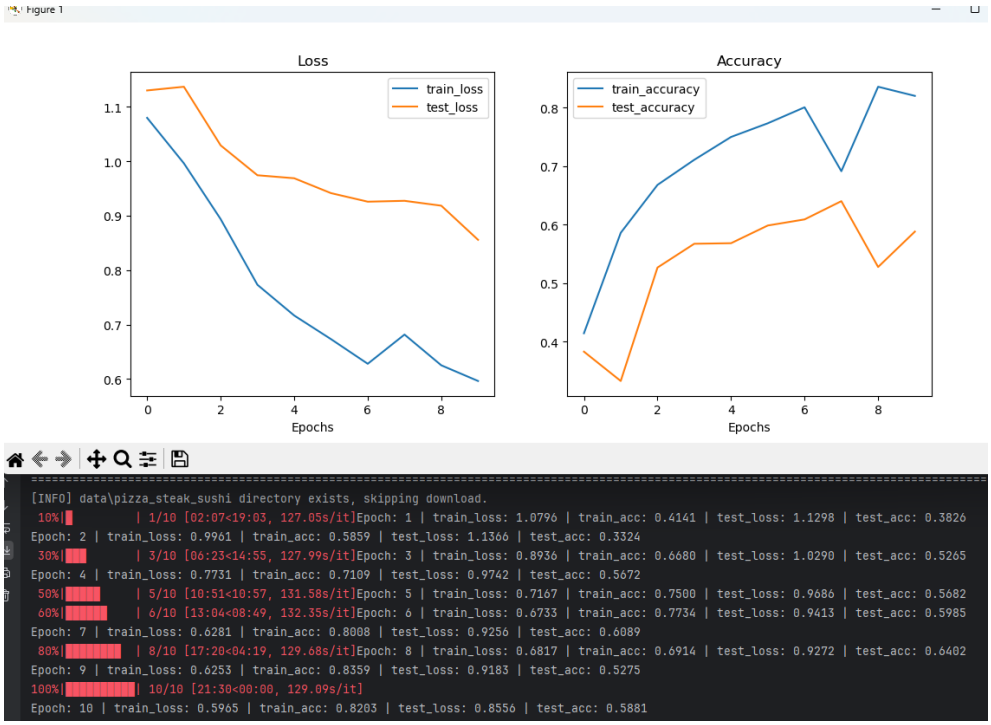
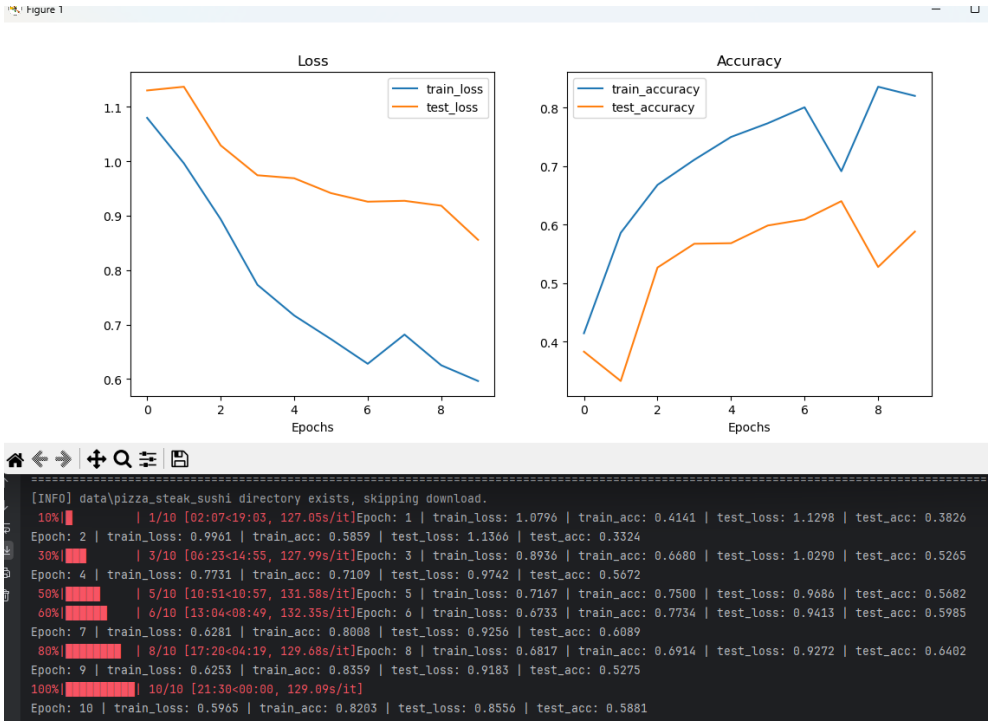


图 11. 实验结果示意




```
[INFO] data\pizza_steak_sushi directory exists, skipping download.
 10%|          | 1/10 [02:07<19:03, 127.05s/it]Epoch: 1 | train_loss: 1.0796 | train_acc: 0.4141 | test_loss: 1.1298 | test_acc: 0.3826
Epoch: 2 | train_loss: 0.9961 | train_acc: 0.5859 | test_loss: 1.1366 | test_acc: 0.3324
 30%|          | 3/10 [06:23<14:55, 127.99s/it]Epoch: 3 | train_loss: 0.8936 | train_acc: 0.6680 | test_loss: 1.0290 | test_acc: 0.5265
Epoch: 4 | train_loss: 0.7731 | train_acc: 0.7109 | test_loss: 0.9742 | test_acc: 0.5672
 50%|          | 5/10 [10:51<10:57, 131.58s/it]Epoch: 5 | train_loss: 0.7167 | train_acc: 0.7500 | test_loss: 0.9686 | test_acc: 0.5682
 60%|          | 6/10 [13:04<08:49, 132.35s/it]Epoch: 6 | train_loss: 0.6733 | train_acc: 0.7734 | test_loss: 0.9413 | test_acc: 0.5985
Epoch: 7 | train_loss: 0.6281 | train_acc: 0.8008 | test_loss: 0.9256 | test_acc: 0.6089
 80%|          | 8/10 [17:20<04:19, 129.68s/it]Epoch: 8 | train_loss: 0.6817 | train_acc: 0.6914 | test_loss: 0.9272 | test_acc: 0.6402
Epoch: 9 | train_loss: 0.6253 | train_acc: 0.8359 | test_loss: 0.9183 | test_acc: 0.5275
100%|          | 10/10 [21:30<00:00, 129.09s/it]
Epoch: 10 | train_loss: 0.5965 | train_acc: 0.8203 | test_loss: 0.8556 | test_acc: 0.5881
```

图 12. 实验结果示意

5.2 探索使用混合 Token 进行分类输入的实验分析

直接加合混合：发现效果不错，准确率有 92% 左右但是有振荡的情况出现


```
[INFO] data\pizza_steak_sushi directory exists, skipping download.
 10%|          | 1/10 [02:47<25:07, 167.52s/it]Epoch: 1 | train_loss: 1.0872 | train_acc: 0.4102 | test_loss: 1.1556 | test_acc: 0.3419
 20%|          | 2/10 [05:06<20:04, 150.59s/it]Epoch: 2 | train_loss: 0.8991 | train_acc: 0.5781 | test_loss: 1.0874 | test_acc: 0.3835
Epoch: 3 | train_loss: 0.8723 | train_acc: 0.6719 | test_loss: 1.0588 | test_acc: 0.4754
 40%|          | 4/10 [09:43<14:15, 142.57s/it]Epoch: 4 | train_loss: 0.7401 | train_acc: 0.6797 | test_loss: 1.0435 | test_acc: 0.4754
 50%|          | 5/10 [12:02<11:45, 141.13s/it]Epoch: 5 | train_loss: 0.6983 | train_acc: 0.7305 | test_loss: 1.0307 | test_acc: 0.4858
Epoch: 6 | train_loss: 0.6673 | train_acc: 0.7578 | test_loss: 0.9949 | test_acc: 0.4962
 70%|          | 7/10 [16:40<07:00, 140.06s/it]Epoch: 7 | train_loss: 0.6409 | train_acc: 0.7930 | test_loss: 1.0169 | test_acc: 0.5066
 80%|          | 8/10 [18:58<04:38, 139.38s/it]Epoch: 8 | train_loss: 0.6308 | train_acc: 0.7969 | test_loss: 1.0070 | test_acc: 0.4858
Epoch: 9 | train_loss: 0.6530 | train_acc: 0.7930 | test_loss: 0.9435 | test_acc: 0.4773
100%|          | 10/10 [23:35<00:00, 141.54s/it]
Epoch: 10 | train_loss: 0.5949 | train_acc: 0.7773 | test_loss: 0.8310 | test_acc: 0.5672
```

图 13. 实验结果示意

平均混合：发现效果不错，准确率竟然有 95% 左右，比 [CLS] 还高

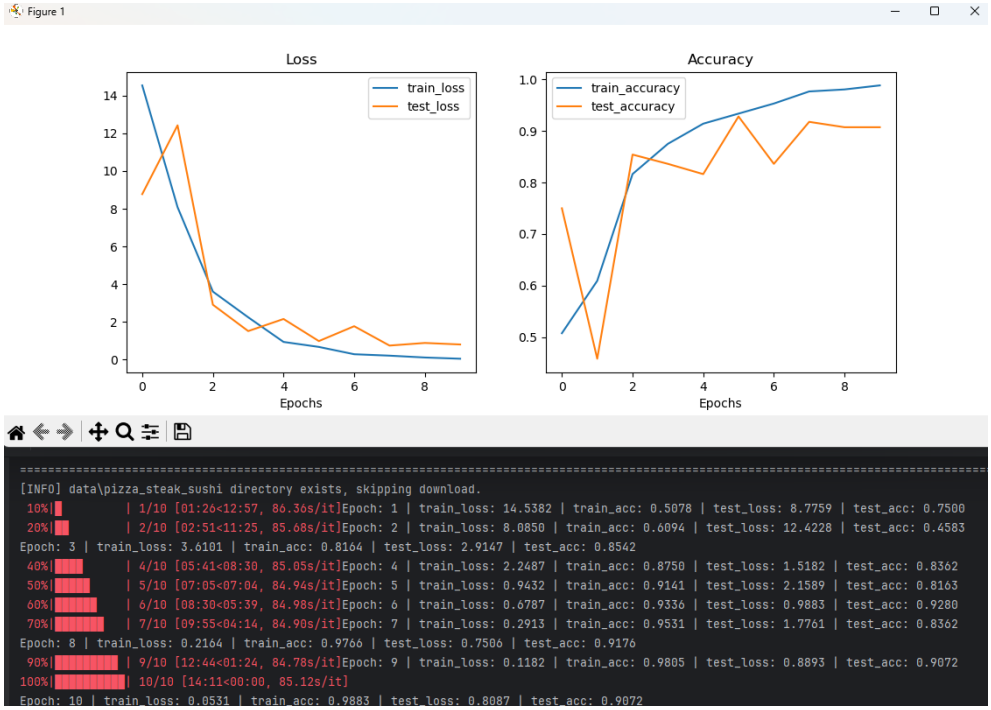


图 14. 实验结果示意

加权混合：准确率有 93% 左右，而且收敛速度比 [CLS] 快

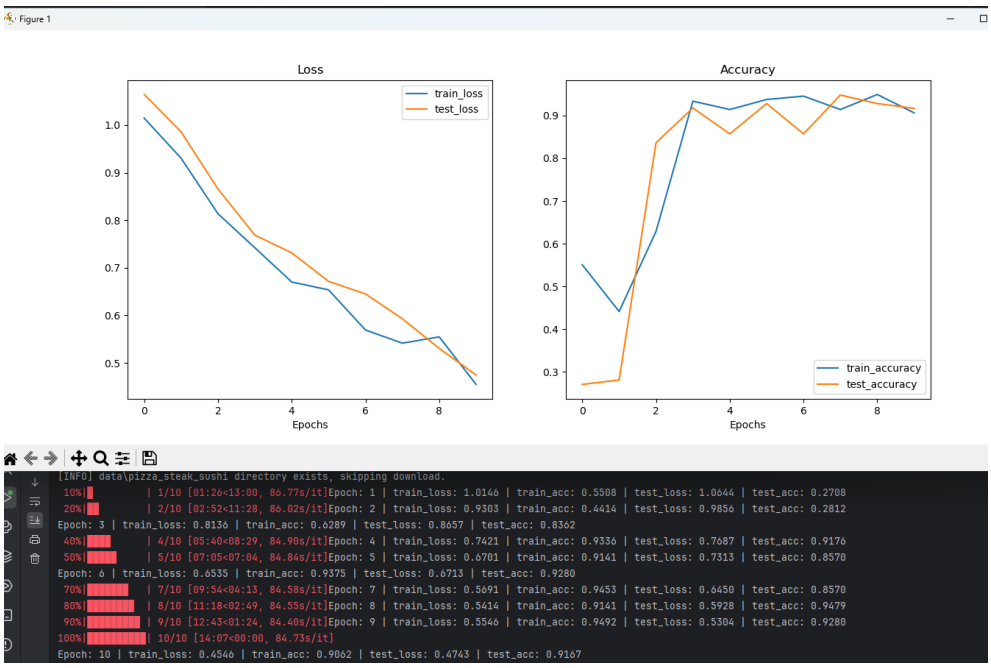


图 15. 实验结果示意

6 总结

4 个边角的 Patch 经过 Encoder 后融合到的周围信息不如中间 Patch 融合到得多，故效果差。中间 Patch 作为分类器输入虽然也能达到不错的效果，但比起 [CLS] 还是有不少的差距。本次实验验证了 [CLS] 在做分类任务时的有效性。直接加和效果不如 [CLS] 的原因，可

能是由于所有的 Patch 特征没经过筛选就直接全部叠加，导致无用特征也被输入到分类器中，影响结果

综合下来看，平均混合的效果比 [CLS] 稍微好些，准确率能有 1% 的提升，但是还是有些振荡的情况存在，稳定性不如 [CLS] 本项目的加权混合，借鉴了 ResNet 的思想，将 CLS 单独取出，不参与加权求和，而是和其他加权后的 Token 一起输入分类器中，同时融合了 [CLS] 和平均混合的特点，使他它同时兼备 [CLS] 的快速收敛性和平均混合的准确性，也具有不错的效果。

参考文献

[1] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. Multi-head attention: Collaborate instead of concatenate, 2021.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[3] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[5] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning, 2020.

[6] Sucheng Ren, Zeyu Wang, Hongru Zhu, Junfei Xiao, Alan Yuille, and Cihang Xie. Rejuvenating image-gpt as strong visual representation learners, 2024.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.