

nanoVLM：在纯 PyTorch 中训练视觉语言模型（VLM）的最简代码库

目录索引

- [1 VLM 快速入门-实战介绍QA](#)
- [2 nanoVLM介绍](#)
 - [概要](#)
 - [什么是视觉语言模型？](#)
 - [代码库使用指南](#)
 - [架构概览](#)
 - [训练你自己的 VLM](#)
 - [模型推理](#)
 - [结论](#)
- [3 视觉大模型VLM实战：代码解读与模型训练](#)
 - [3.1 环境安装（本机运行）](#)
 - [3.2 评估与训练](#)
 - [3.3 数据介绍](#)
 - [3.4 代码解读：模型架构](#)
 - [3.5 图像处理过程](#)

1 VLM 快速入门-实战介绍QA

本教程通过nanoVLM，一个纯pytorch的视觉大模型的训练的代码库，来一起学习如何训练，微调VLM大模型。通过此教程，你可以，

- 快速上手VLM模型的训练，微调
- 深入了解VLM模型的实现细节，以及处理过程
- 没有GPU也可以学（此教程可以在CPU或者小显存GPU上运行）

带着问题来学习，本教程可以帮你回答下面问题：

1. 谁需要学习VLM训练代码的内容？
只使用VLM做应用开发 --> 不需要
进行VLM模型训练，微调，进一步改进性能 --> 需要
2. 视觉encoder，projector，LLM decoder都采用了什么架构？
3. 三个模块之间是如何进行链接的？
4. 训练数据是什么样的格式？
https://huggingface.co/datasets/HuggingFaceM4/the_cauldron
5. 图像是如何一步步被处理的？
6. 图像相关：不同分辨率的图像该如何处理？
7. 图像相关：像素重排（pixel shuffle）是做什么？
8. 图像相关：什么是图像patch embedding？
9. 这么一个小的模型，效果如何呢？
模型虽小，但是可以明显的看出训练后，模型对图像的理解能力。

2 nanoVLM介绍

概要

nanoVLM 是使用纯 PyTorch 开始训练你自己的视觉语言模型（VLM）的最简便方式。它是一个轻量级工具包，允许你在免费额度的 Colab 笔记本上启动 VLM 训练。

nanoVLM 的核心是一个工具集，帮助你构建并训练一个既能理解图像又能理解文本的模型，然后基于此生成文本。nanoVLM 的魅力在于它的简洁性。整个代码库故意保持最小化且易读，非常适合初学者，或任何想在不被繁杂细节淹没的情况下，深入了解 VLM 内部原理的人。

在这篇博客文章中，将介绍该项目背后的核心理念，并提供一种简单的方式与你的代码库进行交互。本项目不仅深入探讨项目细节，还将所有内容封装好，帮助你快速上手。

主要内容：

- 什么是视觉语言模型？
- 代码库使用指南
- 架构设计
- 训练你自己的 VLM
- 在预训练模型上运行推理
- 结论
- 参考文献

你可以通过以下步骤，使用 nanoVLM 工具包开始训练视觉语言模型：

```
# 克隆代码库
git clone https://github.com/huggingface/nanoVLM.git

# 执行训练脚本
python train.py
```

同时，还提供了一个 Colab 笔记本，让你无需任何本地环境配置即可启动训练！

什么是视觉语言模型？

顾名思义，视觉语言模型（Vision Language Model, VLM）是一种多模态模型，同时处理视觉和文本两种信息。这类模型通常以图像和/或文本作为输入，并生成文本作为输出。

基于对图像和文本的理解来生成文本（输出）是一种非常强大的范式。它能够支持广泛的应用场景，从图像描述（Image Captioning）、目标检测（Object Detection），到对视觉内容进行问答（Visual Question Answering）。下面的表格展示了几种常见的任务及其示例：



输入示例	输出示例	应用
Caption the image	Two cats lying down on a bed with remotes near them	图像描述（Captioning）
Detect the objects in the image	<locxx><locxx><locxx><locxx>	目标检测（Object Detection）
Segment the objects in the image	<segxx><segxx><segxx>	语义分割（Semantic Segmentation）
How many cats are in the image?	2	视觉问答（Visual Question Answering）

注意： nanoVLM 在训练时仅聚焦于“视觉问答”（Visual Question Answering）这一目标任务。

如果你想深入了解视觉语言模型，强烈推荐阅读我们最新的专题博客：Vision Language Models (Better, Faster, Stronger <https://huggingface.co/blog/vlms-2025>)。

代码库使用指南

“空谈无用，展示代码。” — Linus Torvalds

在本节中，将带你浏览整个代码库。建议在本地或浏览器中打开一个标签页，边看边操作，帮助理解。

以下是代码库的主要目录结构（已省略部分辅助文件）：

```
.
├── data
│   ├── collators.py
│   ├── datasets.py
│   └── processors.py
├── generate.py
├── models
│   ├── config.py
│   ├── language_model.py
│   ├── modality_projector.py
│   ├── utils.py
│   ├── vision_language_model.py
│   └── vision_transformer.py
└── train.py
```

架构概览

```
.
├── data
│   └── ...
├── models      # 📁 你当前查看的目录
│   └── ...
└── train.py
```

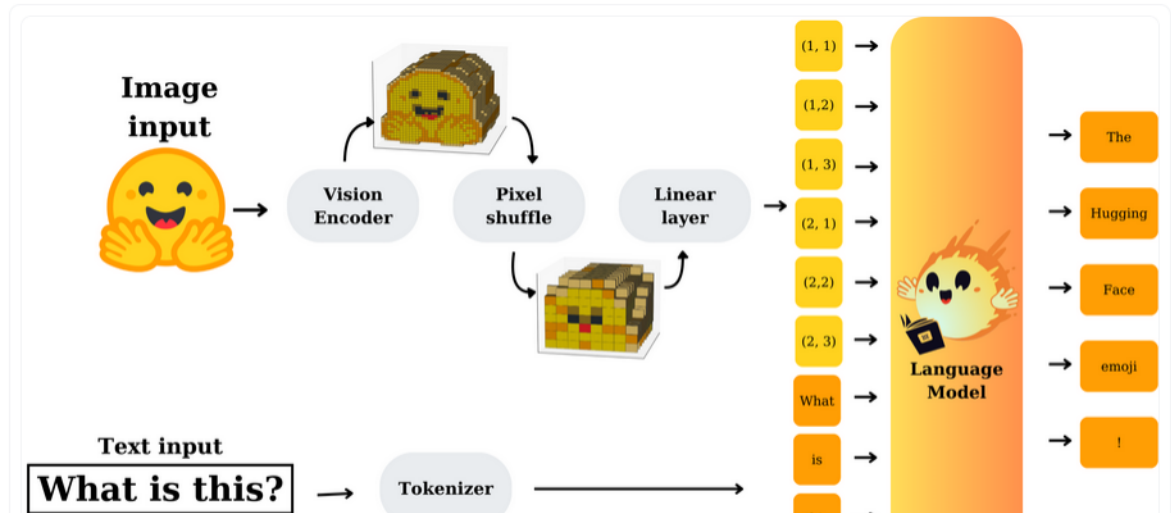
nanoVLM 的设计参考了两种广为流行的架构：

- 1. **视觉主干 (Vision Backbone)**
 - 位于 `models/vision_transformer.py`
 - 采用标准的 Vision Transformer 架构，具体来说是 Google 的 SigLIP 视觉编码器
- 2. **语言主干 (Language Backbone)**
 - 位于 `models/language_model.py`
 - 基于 Llama 3 架构

为了将视觉和文本两种模态对齐，我们引入了 **Modality Projection**（模态投射）模块。该模块流程如下：

- 1. 从视觉主干获取图像嵌入（image embeddings）。
- 2. 通过像素重排（pixel shuffle）操作，再接一个线性层，将图像嵌入映射到与语言模型嵌入层兼容的维度。
- 3. 将投射后的视觉嵌入与文本嵌入拼接（concatenate），一起送入语言解码器（language decoder）。

这样，模型就能在同一潜在空间中“看懂”图像并“说出”相应的文本了。



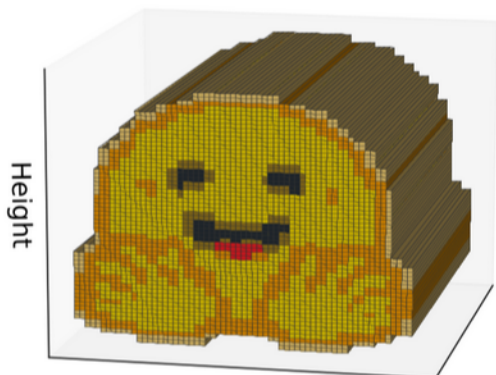


The architecture of the model (Source: Authors)

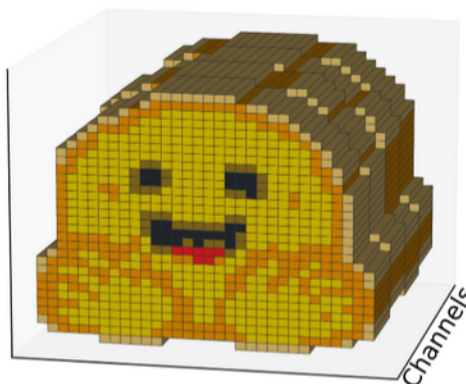
像素重排 (pixel shuffle) 可以减少图像 token 的数量，从而降低计算开销并加快训练速度——这一点对于对输入长度极其敏感的基于 Transformer 的语言解码器尤其重要。下图演示了这一概念。

Pixel Shuffle: Trading Resolution for Channels

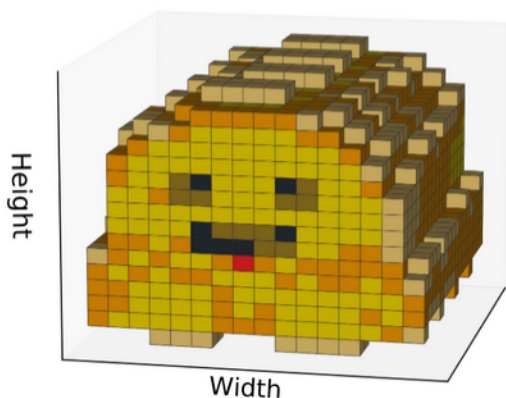
(a) Original Image ($64 \times 64 \times 1$)



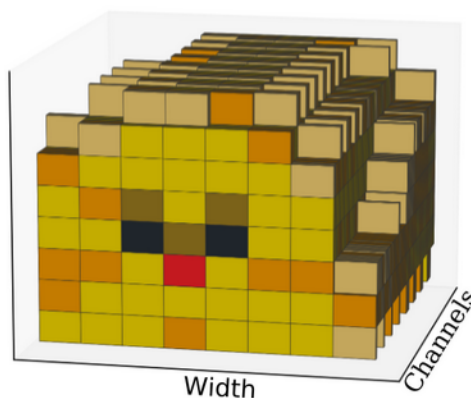
(b) $r=2$ ($32 \times 32 \times 4$)



(c) $r=4$ ($16 \times 16 \times 16$)



(d) $r=8$ ($8 \times 8 \times 64$)



Pixel Shuffle Visualized (Source: Authors)

所有文件都非常轻量且注释详尽。我们强烈建议你逐一查看这些脚本（如 `models/*.py`），以深入了解具体实现细节。

在训练过程中，我们使用了以下预训练骨干权重：

- **视觉骨干 (Vision backbone)** : `google/siglip-base-patch16-224`
- **语言骨干 (Language backbone)** : `HuggingFaceTB/SmolLM2-135M`

你也可以将骨干网络替换为其他版本的 SigLIP/SigLIP 2（用于视觉）或者 SmolLM2（用于语言），以满足不同的性能和资源需求。

训练你自己的 VLM

现在我们已经熟悉了模型架构，接下来讨论如何使用 `train.py` 来训练你自己的视觉语言模型。

```
.
├── data
│   └── ...
├── models
│   └── ...
└── train.py    # 📁 你现在所在的位置
```

你只需执行以下命令即可开始训练：

```
python train.py
```

此脚本是一站式训练管道，涵盖了：

- 数据集加载与预处理
- 模型初始化
- 优化与日志记录

配置 (Configuration)

脚本加载了 `models/config.py` 中的两个配置类：

- `TrainConfig`：训练相关的参数（如学习率、检查点路径等）
- `VLMConfig`：用于初始化 VLM 的参数（如隐藏维度、注意力头数等）

数据加载 (Data Loading)

数据管道的核心是 `get_dataloaders` 函数，它会：

1. 通过 Hugging Face 的 `load_dataset` API 加载数据集
2. （可选）合并并打乱多个数据集
3. 按索引对数据做训练/验证集划分
4. 使用自定义数据集类（`VQADataset`、`MMStarDataset`）和拼接器（`VQACollator`、`MMStarCollator`）进行封装

一个有用的调试参数是 `data_cutoff_idx`，可用于在小规模子集上快速测试。

模型初始化 (Model Initialization)

模型由 `VisionLanguageModel` 类构建。如果你想从检查点恢复训练，只需：

```
from models.vision_language_model import VisionLanguageModel

model = VisionLanguageModel.from_pretrained(model_path)
```

否则，将会得到一个全新初始化的模型，并可选择性地加载预训练的视觉和语言主权重。

优化器设置：双学习率 (Two LRs)

由于模态投射模块（MP）是从头开始训练的，而主干网络已预训练，优化器被划分为两个参数组，各自使用不同的学习率：

- 对 MP 使用较高的学习率
- 对视觉/语言主干使用较低的学习率

这样可以让投射模块快速学习，同时保留主干网络中已有的知识。

训练循环 (Training Loop)

训练循环虽然相对常规，但结构设计周到：

- 使用 `torch.autocast` 进行混合精度训练，以提升性能
- 通过 `get_lr` 实现带线性预热的余弦学习率调度
- 每个 batch 记录 token 吞吐量（tokens/sec）以便性能监控

每隔 250 步（可配置），脚本会在验证集和 MMStar 测试集上评估模型；若准确率提高，则保存检查点。

日志记录与监控 (Logging & Monitoring)

启用 `log_wandb` 后，训练过程中的 `batch_loss`、`val_loss`、`accuracy`、`tokens_per_second` 等指标将实时上传到 Weights & Biases，以便可视化和跟踪。

运行名称会根据样本大小、批次大小、训练轮次、学习率和日期等元数据自动生成，均由辅助函数 `get_run_name` 处理。

推送到 Hub (Push to Hub)

训练完成后，可将模型保存并推送到 Hugging Face Hub，方便他人下载和测试：

```
# 保存到本地目录
model.save_pretrained(save_path)

# 推送到 Hub
model.push_to_hub("hub/id")
```

模型推理

使用 nanoVLM 工具包，我们已经训练好了一个模型并将其发布到 Hugging Face Hub。训练时选用了 `google/siglip-base-patch16-224` 和 `HuggingFaceTB/SmolLM2-135M` 作为骨干网络，在单块 H100 GPU 上、约 1.7M 样本和大约 6 小时内完成训练。

此模型并非为了与最先进 (SoTA) 模型竞争，而是为了揭示 VLM 的各个组件和训练流程。

```
.
├── data
│   └── ...
├── generate.py    # 📁 你现在所在的位置
├── models
│   └── ...
└── ...
```

运行以下命令即可使用 `generate.py` 脚本对训练好的模型进行推理：

```
python generate.py
```

默认情况下，该脚本会对 `assets/image.png` 这张图片执行查询 “What is this?”。你也可以对自己的图片和提示词运行该脚本：

```
python generate.py --image path/to/image.png --prompt "Your prompt here"
```

以下是脚本的核心代码片段：

```
model = VisionLanguageModel.from_pretrained(source).to(device)
model.eval()



tokenizer = get_tokenizer(model.cfg.lm_tokenizer)
image_processor = get_image_processor(model.cfg.vit_img_size)

template = f"Question: {args.prompt} Answer:"
encoded = tokenizer.batch_encode_plus([template], return_tensors="pt")
tokens = encoded["input_ids"].to(device)

img = Image.open(args.image).convert("RGB")
img_t = image_processor(img).unsqueeze(0).to(device)

print("\nInput:\n ", args.prompt, "\n\nOutputs:")
for i in range(args.generations):
    gen = model.generate(tokens, img_t, max_new_tokens=args.max_new_tokens)
    out = tokenizer.batch_decode(gen, skip_special_tokens=True)[0]
    print(f"    >> Generation {i+1}: {out}")
```

1. 加载并切换到评估模式： `model.eval()`
2. 初始化分词器 (tokenizer) 和图像处理器 (image_processor)
3. 调用 `model.generate` 生成文本
4. 使用 `batch_decode` 将生成的 token 解码为可读文本

图片	提示 (Prompt)	生成 (Generation)
	What is this?	In the picture I can see the pink color bed sheet. I can see two cats lying on the bed sheet.
	What is the woman doing?	Here in the middle she is performing yoga

结论

在本篇博客中，我们介绍了什么是视觉语言模型（VLM），详解了 nanoVLM 的架构设计，并深入剖析了训练与推理的完整流程。

通过保持代码库的轻量 and 可读，nanoVLM 既能作为学习工具，也能作为你后续开发的基础。无论你是想了解多模态输入如何对齐，还是希望在自己的数据集上训练 VLM，这个仓库都能帮助你快速入门。

3 视觉大模型VLM实战：代码解读与模型训练

此代码库，可以通过本机安装运行，也可以使用google colab来运行。

3.1 环境安装（本机运行）

因为是纯pytorch的实现，安装比较简单

```
git clone https://github.com/huggingface/nanoVLM.git
cd nanoVLM
uv init --bare --python 3.12
uv sync --python 3.12
source .venv/bin/activate
uv add torch numpy torchvision pillow datasets huggingface-hub transformers wandb
```

3.2 评估与训练

```
# 激活nv运行环境
source .venv/bin/activate
# 推理
python generate.py
# 模型训练（单GPU）
torchrun --nproc_per_node=1 train.py
```

3.3 数据介绍

训练数据采用的是：

https://huggingface.co/datasets/HuggingFaceM4/the_cauldron

下面是数据集中的一个样本，其中包含：

- 一张图片
- 问题：对应 user 字段
- 回答：对应 assistant 部分



```
[
{
  "user": "Question: What do respiration and combustion give out\nChoices:\nA. Oxygen\nB. Carbon dioxide\nC. Nitrogen\nD. Heat\nAnswer with the letter.",
  "assistant": "Answer: B",
  "source": "AI2D"
}
```

3.4 代码解读：模型架构

代码解读部分，列出了VLM最主要的核心部分，主要包括 VisionLanguageModel, Vision Encoder, projector, decoder

VisionLanguageModel

https://github.com/huggingface/nanoVLM/blob/1ce10cf53ac23af8693a462e7ae83efe9a8211ef/models/vision_language_model.py#L20

```
class VisionLanguageModel(nn.Module):
    def __init__(self, cfg: VLMConfig, load_backbone=True):
        if load_backbone: # 【使用预训练的 ViT 和 LLM】
            print("Loading from backbone weights")
            self.vision_encoder = ViT.from_pretrained(cfg)
            self.decoder = LanguageModel.from_pretrained(cfg)
        else: # 【完全初始化模型，包括encoder, LLM decoder】
            self.vision_encoder = ViT(cfg)
            self.decoder = LanguageModel(cfg)
        self.MP = ModalityProjector(cfg) # 【Projector】

    def forward(self, input_ids, image, attention_mask=None, targets=None):
        image_embd = self.vision_encoder(image)
        image_embd = self.MP(image_embd)

        token_embd = self.decoder.token_embedding(input_ids)

        combined_embd = torch.cat((image_embd, token_embd), dim=1) # Concatenate image embeddings to token embeddings

        logits = self.decoder(combined_embd, attention_mask)
```

Vision Encoder

https://github.com/huggingface/nanoVLM/blob/1ce10cf53ac23af8693a462e7ae83efe9a8211ef/models/vision_transformer.py#L156


```

def __init__(self, cfg):
    self.patch_embedding = ViTPatchEmbeddings(cfg)
    self.blocks = nn.ModuleList([ViTBlock(cfg) for _ in range(cfg.vit_n_blocks)])

def forward(self, x):
    x = self.patch_embedding(x) # Patch embedding的处理
    x = self.dropout(x)
    for block in self.blocks:
        x = block(x)

    if self.cls_flag:
        x = self.layer_norm(x[:, 0])
    else:
        x = self.layer_norm(x)
    #x = x.mean(dim=1)

```

Projector

https://github.com/huggingface/nanoVLM/blob/1ce10cf53ac23af8693a462e7ae83efe9a8211ef/models/modality_projector.py#L12

```

class ModalityProjector(nn.Module):
    def __init__(self, cfg):
        # projector 为单层 linear
        self.proj = nn.Linear(self.input_dim, self.output_dim, bias=False)

    def forward(self, x):
        x = self.pixel_shuffle(x) # pixel shuffle的处理
        x = self.proj(x)

```

LLM Decoder

https://github.com/huggingface/nanoVLM/blob/1ce10cf53ac23af8693a462e7ae83efe9a8211ef/models/language_model.py#L235

```

class LanguageModel(nn.Module):
    def __init__(self, cfg):
        self.token_embedding = nn.Embedding(cfg.lm_vocab_size, cfg.lm_hidden_dim)
        self.rotary_embd = RotaryEmbedding(cfg)
        self.blocks = nn.ModuleList([
            LanguageModelBlock(cfg) for _ in range(cfg.lm_n_blocks)
        ])
        self.norm = RMSNorm(cfg) # Final Norm
        self.head = nn.Linear(cfg.lm_hidden_dim, cfg.lm_vocab_size, bias=False)

    def forward(self, x, attention_mask=None):
        x = self.token_embedding(x) # Only embed the inputs when using tokens
        B, T, _ = x.size()
        # Create position ids [0, 1, 2, ..., seq_len-1]
        position_ids = torch.arange(T, device=x.device).unsqueeze(0).expand(B, -1)
        cos, sin = self.rotary_embd(position_ids) # Get rotary position embeddings
        for block in self.blocks:
            x = block(x, cos, sin, attention_mask)
        x = self.norm(x)
        x = self.head(x)

```

模型训练相关配置

<https://github.com/huggingface/nanoVLM/blob/main/models/config.py>

3.5 图像处理过程

通过本教程修改后的代码，你可以非常方便的确认一张图片是如何被处理，并被最终送到LLM decoder中进行模型建模的。

你可以运行 nanoVLM_augudao_v1.ipynb (jupyter, colab) 或者 nanoVLM_augudao_v1.py 来进行调试。

具体调试过程，可以参考介绍视频： ? ? ? ?

将下面代码更新，用来打印各步处理后数据的维度变化。

```
models/vision_language_model.py Line:33
def forward(self, input_ids, image, attention_mask=None, targets=None):
    print("VLM - image shape:", image.shape) #torch.Size([12, 3, 224, 224])
    image_embd = self.vision_encoder(image)
    print("VLM - after vision_encoder:", image_embd.shape) #torch.Size([12, 196, 768])
    image_embd = self.MP(image_embd)
    print("VLM - after MP:", image_embd.shape) #torch.Size([12, 49, 576])

    token_embd = self.decoder.token_embedding(input_ids)
    print("VLM - after token_embedding (token_embd):", token_embd.shape) #torch.Size([12, 79, 576])

    combined_embd = torch.cat((image_embd, token_embd), dim=1) # Concatenate image embeddings to token embeddings
    print("VLM - after token_embedding (combined_embd):", combined_embd.shape) #torch.Size([12, 128, 576])

models/modality_projector.py Line:40
def forward(self, x):
    print("MP - Before pixel_shuffle:",x.shape) #torch.Size([12, 196, 768])
    x = self.pixel_shuffle(x)
    print("MP - After pixel_shuffle:",x.shape) #torch.Size([12, 49, 3072])
    x = self.proj(x)
    print("MP - After proj:",x.shape) #torch.Size([12, 49, 576])
    return x
```

程序执行后输出的维度变化

```
VLM - image shape: torch.Size([12, 3, 224, 224])
VLM - after vision_encoder: torch.Size([12, 196, 768])
MP - Before pixel_shuffle: torch.Size([12, 196, 768])
MP - After pixel_shuffle: torch.Size([12, 49, 3072])
MP - After proj: torch.Size([12, 49, 576])
VLM - after MP: torch.Size([12, 49, 576])
VLM - after token_embedding (token_embd): torch.Size([12, 79, 576])
VLM - after token_embedding (combined_embd): torch.Size([12, 128, 576])
```

从图片到token向量处理流程介绍：

1. 原始图片张量

```
image.shape # torch.Size([12, 3, 224, 224])
```

- Batch 大小 12;
- 3 个通道 (RGB) ;
- 空间分辨率 224×224。

2. 经过视觉编码器 (patch embedding + Transformer)

```
image_embd = self.vision_encoder(image)
image_embd.shape # torch.Size([12, 196, 768])
```

- 224×224 的图片被切成 14×14 个 patch，因此序列长度 14×14=196。
- 每个 patch 被映射到 768 维的向量，得到 [batch, num_patches, embed_dim]。

3. Modality Projector (下采样 + 投影)

```
image_embd = self.MP(image_embd)
image_embd.shape # torch.Size([12, 49, 576])
```

- 这里的 MP 通常做两件事：
 - a. **token 下采样**：把 14×14 的 patch token 用类似 pixel-shuffle、patch-merging 或可学习的聚合，从 196 降到 7×7=49;
 - b. **维度投影**：映射到 576 维，以便后续和语言侧保持同一 hidden size。
- 结果是 [batch, 49, 576]，即更少的视觉 token，但每个 token 的向量里承载更多的信息。

4. 语言侧 token embedding + 拼接

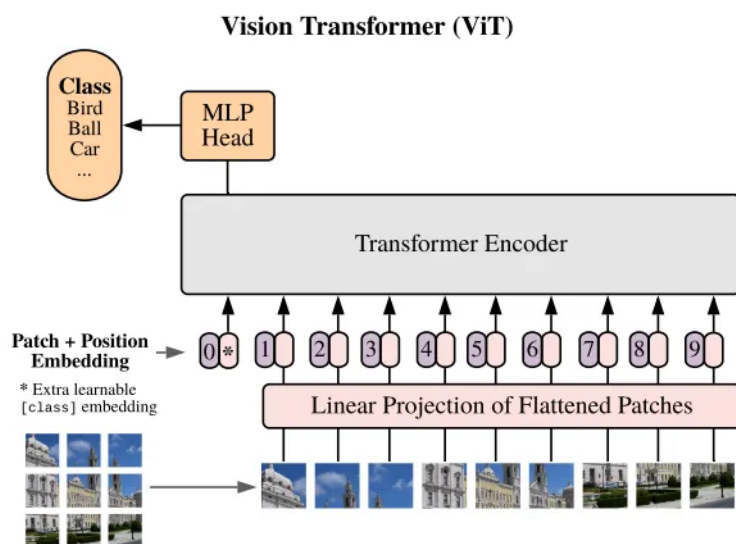
```
token_embd = self.decoder.token_embedding(input_ids)
token_embd.shape # torch.Size([12, 79, 576])

combined_embd = torch.cat((image_embd, token_embd), dim=1)
combined_embd.shape # torch.Size([12, 128, 576])
```

- `input_ids` (长度假设是 79) 被映射到同样的 576 维空间, 得到 `[batch, 79, 576]`。
- 把视觉部分的 49 token 和语言部分的 79 token 在序列维 (`dim=1`) 上拼接, 就得到了 `[batch, 49+79=128, 576]`。
- 最终这个长度为 128、维度为 576 的混合序列, 就可以输入到 Decoder 完成图文联合建模。

VIT 处理示意图

通过此图, 可以帮助我们理解什么是patch embedding: 此图中图片被拆分为16个小块, 然后按顺序排列后, 被ViT分别进行处理。



图片来源 (其中有动画演示) :

<https://medium.com/@fernandopalominocobo/demystifying-visual-transformers-with-pytorch-understanding-patch-embeddings-part-1-3-ba380f2aa37f>