

OpenVLA开源项目的跑通记录

一个生活比喻

你可以把操作系统想象成"厨房"：**Windows** 是精装修的品牌厨房，好看易用，但布局固定，改造困难。**Linux** 是毛坯厨房，给你全套工具和图纸，你可以随意改造成任何样子。

对于普通用户，精装修更省心；对于开发者和研究者，毛坯房的自由度更高。

为什么开发者经常使用这个linux

WSL2 的本质

回到你要做的事情：WSL2 相当于在你 Windows 电脑的"身体"里，**额外装了一个 Linux 灵魂**，让它们共存。两个"灵魂"共享同一套硬件（CPU、显卡、内存），但各自有独立的文件系统和运行环境。

卧槽这个东西，真是精彩至极啊

本地小规模验证，目的是，输入文本加图像，输出这个action序列

结构

```
1 windows 11 (主系统)
2   └─ wsl 2 (微软的Linux兼容技术)
3     └─ ubuntu 22.04 (你选择的Linux发行版)
4       └─ Conda 环境 (openvla_wsl)
5         └─ OpenVLA 项目
```

生活比喻

GitHub = 代码的共享平台（存放代码）

HuggingFace = 模型的共享平台（存放训练好的"大脑"）

就像你去 GitHub 下载别人写好的代码一样，你去 HuggingFace 下载别人训练好的模型

可以AI界的github

在做量化减少这个开销的时候，系统的自动调度（accelerate硬件分配），和这个.to(device)的矛盾

这个错误困扰了我一个小时

收到：你现在只差把 `/act` 的真实输入 JSON 搞清楚并成功打一次请求拿到动作输出，就完成 OpenVLA baseline 闭环。下面我直接把“最后一公里”按最小闭环 3 条命令给你（不需要猜 schema）。

1) 一条命令：从 OpenAPI 反推 `/act` 的真实 Request body (最快、最准)

在 WSL 里跑：

bash

Copy code

```
curl -s http://localhost:8000/openapi.json | python -m json.tool | sed -n '/"\\"/act"/,/^ }/p'
```

你重点看这几块（在输出里搜索）：

- `"requestBody"` 下面的 `"schema"` / `"properties"`：会列出所有字段名
- 以及 `"required"`：哪些是必填
- `"content"`：看是 `application/json` 还是 `multipart/form-data`

如果这里能看到字段名，那就完全不需要进代码，直接进入第 3 步发请求。

这个act，相当于一个项目说明书，规范着我们实际的运行

```
1 python - <<'PY'
2 import requests, numpy as np
3 from PIL import Image
4
5 img_path = "/mnt/d/02614db1fa77ac2655db5be6048dc599.jpg" # 换成你的图片路径
6 img = np.array(Image.open(img_path).convert("RGB"), dtype=np.uint8)
7
8 payload = {
9     "instruction": "pick up the red screw",
10    "image": img.tolist(),    # 关键：用 JSON 能传的 list
11 }
12
13 r = requests.post("http://localhost:8000/act", json=payload, timeout=120)
14 print("status:", r.status_code)
15 print("text:", r.text[:500])
16 print(r.headers)
17 print("server:", r.headers.get("server"))
18
19 PY
20
```

```
⑧ text head: {"__numpy__": "AIAImFo47z4QKc099RZ+P9rh4YHMxYw/ONPxndn99kL+gCB1LseuIvyAS6QzQLJe/AAAAAAAAAAA=", "dtype": (openvla_wsl) ubuntu@LAPTOP-A9ON60MQ:~/projects/openvla$ python test_decode_action.py
HTTP 状态码: 200
解码后类型: <class 'numpy.ndarray'>
解码后action: [ 1.48869192e-05  7.34611318e-03  1.40491464e-02 -1.61037365e-02
 -1.21682979e-02 -2.26318844e-02  0.00000000e+00]
action 维度: 7
(openvla_wsl) ubuntu@LAPTOP-A9ON60MQ:~/projects/openvla$
```

太难了，这个结果出来真不容易，我决定稍后学习当下最先进的工具，掌握一整套开发流程

```
time.sleep(0.2)
open("/tmp/openvla_baseline_5runs.json","w").write(json.dumps({
    "ts": time.time(),
    "instruction": instruction,
    "runs": outs
}, ensure_ascii=False, indent=2))
print("saved -> /tmp/openvla_baseline_5runs.json")
PY
0 [1.4886919189882608e-05, 0.014468829232103624, 0.005994562820038369, -0.028237026877263108, -0.029584780253031684, 0.07891139427236468, 0.0]
1 [1.4886919189882608e-05, 0.014468829232103624, 0.005994562820038369, -0.028237026877263108, -0.029584780253031684, 0.07891139427236468, 0.0]
2 [1.4886919189882608e-05, 0.014468829232103624, 0.005994562820038369, -0.028237026877263108, -0.029584780253031684, 0.07891139427236468, 0.0]
3 [1.4886919189882608e-05, 0.014468829232103624, 0.005994562820038369, -0.028237026877263108, -0.029584780253031684, 0.07891139427236468, 0.0]
4 [1.4886919189882608e-05, 0.014468829232103624, 0.005994562820038369, -0.028237026877263108, -0.029584780253031684, 0.07891139427236468, 0.0]
saved -> /tmp/openvla_baseline_5runs.json
(openvla_wsl) ubuntu@LAPTOP-A90N60MQ:~/projects/openvla$
```

22:8 CRLF UTF-8

终于出结果了

不同模糊程度的实验结果的对比

```
open("/tmp/openvla_ambiguity_pair.json","w").write(json.dumps({
    "ts": time.time(),
    "results": results
}, ensure_ascii=False, indent=2))
print("saved -> /tmp/openvla_ambiguity_pair.json")
PY
pick up the screw -> [0.003146326296469725, 0.0021659560519105767, 0.002357008954619655, 0.003054090336257184, -0.004129921381964441, 0.06762880775624616, 0.0]
pick up the red screw -> [-0.0028928782168556808, 0.04069337469514669, 0.02626934355942626, 0.05861178824597689, 0.014626290417769333, -0.04842065355240138, 0.0]
saved -> /tmp/openvla_ambiguity_pair.json
(openvla_wsl) ubuntu@LAPTOP-A90N60MQ:~/projects/openvla$
```

目前完成的任务

你到目前为止完成的任务	你产出的“可交付物”	这件事在科研里叫什么	你现在立刻能做什么（基于它）	对你“模糊指令消歧”课题的意义
把 OpenVLA 服务跑通 (/act 稳定返回 7 维 action, 可解码)	/act 可用的推理服务 + action_7d 可读解析 (delta_xyz / delta_rot / gripper)	Baseline 推理闭环 (能跑、能输出)	给任意图+指令，稳定拿到 action，并把 action 写成“人类可读”的格式	你有了“机器人动作输出”的统一接口，后面消歧策略只需要改变输入文本/流程
做“同图同指令重复多次”验证稳定性 (repeats)	输出里带 latency、重复结果 (N 次 action)	稳定性验证 (repeatability check)	衡量同一输入多次输出是否一致，粗看模型是否抖动	消歧实验必须先保证“模型不乱抖”，否则你分不清是消歧有效还是模型随机性
做“模糊 vs 明确”对照实验，并落盘成标准 JSON	outputs/pair_scene_01.json (meta + results)	最小对照实验 (controlled comparison)	同一张图，换两种指令，得到两组动作输出并保存，可复现	这是“模糊指令”研究的最小实验单元：你已经能观察“语言变清楚 → 动作如何变化”
把零散命令整理成脚本 (run_pair / run_suite)	run_pair.py、run_suite.py	实验脚本化 (reproducible runner)	一条命令跑一对/一批测试样例，自动保存结果	你后面每次改消歧策略，都能用同一套脚本重复跑、对比结果
固定最小评测集配置 (suite.json)	data/suite.json (图像路径 + prompts + repeats 等)	最小评测集 (mini benchmark)	快速扩展到 10-30 张图、20-50 条指令的固定测试集	这是科研最关键的“对照公平性”：同一套输入才能比较不同消歧方法

你到目前为止完成的任务	你产出的“可交付物”	这件事在科研里叫什么	你现在立刻能做什么（基于它）	对你“模糊指令消歧”课题的意义
加入最小量化指标（L2/稳定性等）并输出 CSV	<code>metrics.py</code> + <code>metrics.csv</code> (或 <code>results.csv</code>)	量化评估 (quantitative evaluation)	自动出表：模糊 vs 明确的 L2(action)、最大差异维度、稳定性	你已经能写“实验结果表格”，不是口头描述；为论文实验章节打底
一键化运行 (Makefile pipeline)	<code>Makefile</code> (<code>serve/run_suite/metrics/pipeline</code>)	可复现流水线 (one-command pipeline)	新机器/新同学照着 <code>make pipeline</code> 复现整套结果	复现从“个人工程”变成“团队可复用资产”，后续迭代省很多时间
写清楚复现流程 (README)	<code>README.md</code> (Quick Reproduction)	复现报告雏形 (deliverable reproduction)	别人 clone 仓库，2-3 条命令拿到同样格式输出	你已经具备“可以交付”的 baseline，后面做消歧就是在这个底座上叠加方法

1. L2 (L2 Norm / Euclidean Distance) —— “差异度打分器”

在你的代码里，`L2` 指的是 L2 范数（欧几里得距离）。

- 通俗理解：** 它是用来衡量**“两个动作之间到底差了多少”**的一把尺子。
- 在 OpenVLA 里的作用：** 你之前的实验里，让机器人针对“拿螺丝”（模糊指令）和“拿红螺丝”（清晰指令）分别输出了两个 7维向量（那 7 个数字）。
 - 如果不算 L2，你很难肉眼看出 `[0.1, 0.2...]` 和 `[0.11, 0.19...]` 到底有多大区别。
 - L2 就是把这 7 个数字的差异算成一个总分。**
 - 公式：** $\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots}$

怎么看结果？

- L2 很大：** 说明机器人认为“拿螺丝”和“拿红螺丝”是完全不同的两件事（动作差异大）。
- L2 很小（接近 0）：** 说明机器人觉得这两句话没啥区别，动作基本一致。

在你截图 `image_b5ab1e.png` 第 176 行：`row['l2_mean_amb_vs_clear']` 这就是在计算：“模糊指令下的动作” VS “清晰指令下的动作”到底有多大的距离。这是验证你

总体上的差异度打分器

这个脚本可以当做记录实验结果的表格

总结：你的 `metrics.py` 在干嘛？

这个脚本的工作流是：

1. 读取 JSON：拿到你之前保存的那些“7维动作向量”。
2. 计算 L2：算出向量之间的差异距离（看看机器人是不是真的听懂了指令的区别）。
3. 写入 CSV：把算出来的分数保存成表格文件，作为你的最终实验结论。

关于项目部分代码的理解

这段代码本质上是在做一件很“工程但关键”的事：把你的任务指令（instruction）包装成LLM最熟悉的“聊天对话格式”，让它更容易稳定地产生“动作 token”。你可以把它理解成：给大模型写一张标准化的“提问卡片”。

这一点和规划上下文不谋而合

```
---  
4. 构建Prompt  
  
prompt_builder = self.get_prompt_builder()  
prompt_builder.add_turn(role="human", message=f"What action should the robot take to {instruction.lower()}?")  
prompt_text = prompt_builder.get_prompt()  
  
解释：将用户指令包装成对话格式。例如：  
USER: What action should the robot take to pick up the red block?  
ASSISTANT:  
这样LLM就会"回答"动作token。
```

- 往对话里加一轮：**人类说的话** (`role="human"`)。
- `message` 是一个固定问句模板：
 - “机器人应该采取什么动作来完成 {你的指令}？”
- `instruction.lower()`：把指令转成小写，属于“文本归一化（normalization）”
 - 目的：减少大小写差异带来的无谓变化，让模型输入更一致（比如“Pick up...”和“pick up...”）。

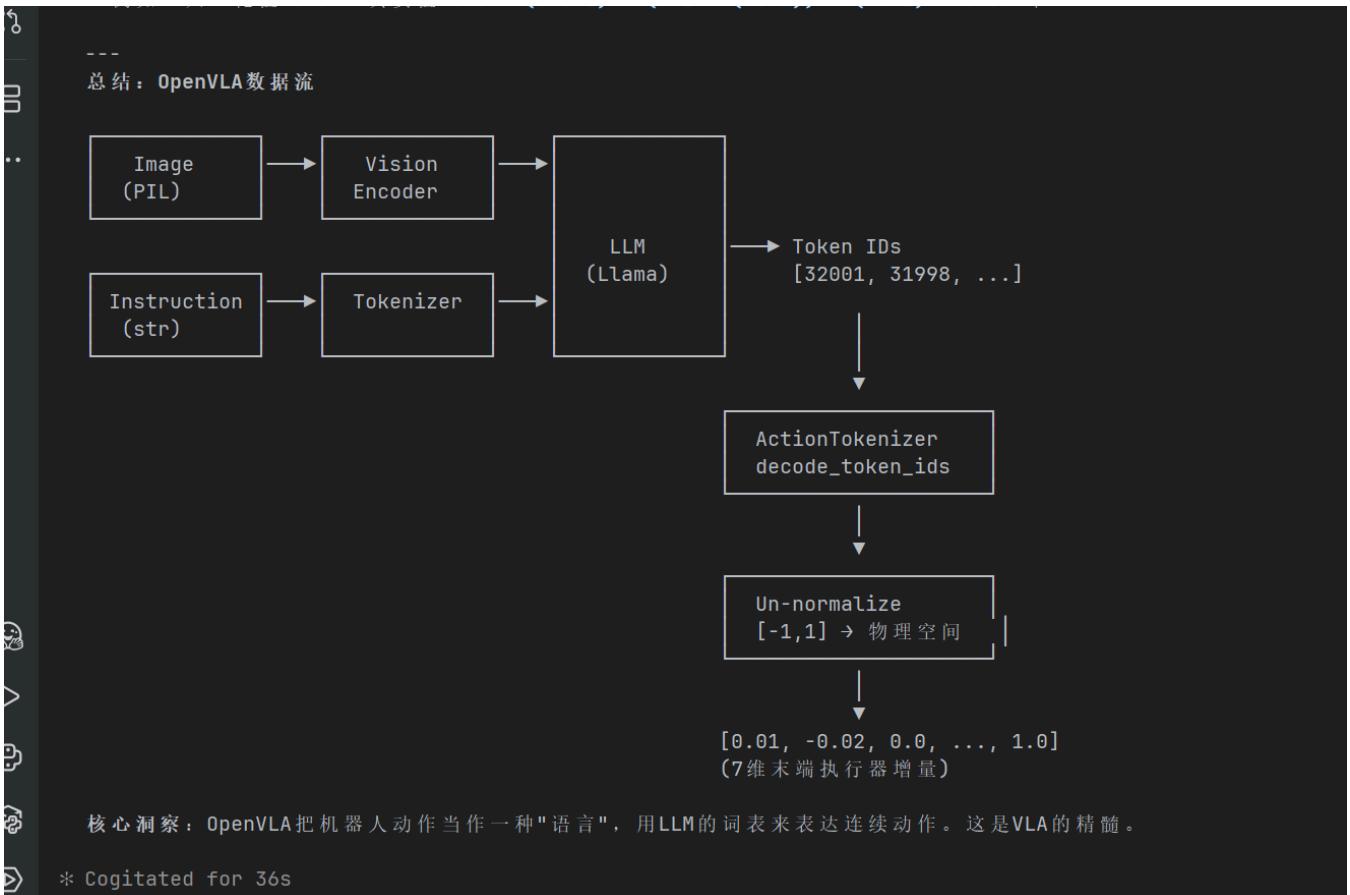
python

 Copy code

```
prompt_text = prompt_builder.get_prompt()
```

- 把当前累计的对话轮次，**拼成最终的 prompt 字符串**。
- 这个字符串后面会被 `tokenizer` 变成 `input_ids`，送进模型。

有意思，文本对话，最终才进行这个执行



直接在项目里给我讲解，这个claude code skill真是强大

1. 类定义：核心思想

```

class ActionTokenizer:
    def __init__(self, tokenizer: PreTrainedTokenizerBase, bins: int = 256, min_action: int = -1, max_action: int = 1) -> None:
        ...

```

解释：

参数	默认值	含义
tokenizer	-	基础LLM的分词器（如LlamaTokenizer）
bins	256	将连续动作离散化为256个档位
min_action	-1	归一化动作下界
max_action	1	归一化动作上界

核心思想：把 [-1, 1] 的连续值切成256份，每份对应一个token。

读了这么多，你应该能够感觉到这个东西的合理性

- 1 | self.action_token_begin_idx: int = int(self.tokenizer.vocab_size - (self.n_bins + 1))
- 2 | 这个冒号是这个类型注解的意思，表示的是这个前面这个东西是这个int类型，可以触发这个静态编译



3. 关键设计：复用词表末尾的 token

```
self.action_token_begin_idx: int = int(self.tokenizer.vocab_size - (self.n_bins
```

解释：

LLaMA词表（32000个token）：



为什么这样做？

- BPE词表中，末尾的token几乎不会出现在自然语言中
- 直接复用这些"死"token表示动作，无需扩展词表
- 巧妙地将动作嵌入到语言模型的输出空间

非常符合工程直觉，并且很多东西都是一个意义的赋予，而不是这个本来的原貌

因为模型不能直接理解字符串，它只能处理数字。

2) 什么叫 token id? (中文：词元编号 / token 的索引)

token id 就是：

“每一个 token 在词表 (vocabulary) 里的编号 (ID) ”。

你可以把它类比成：

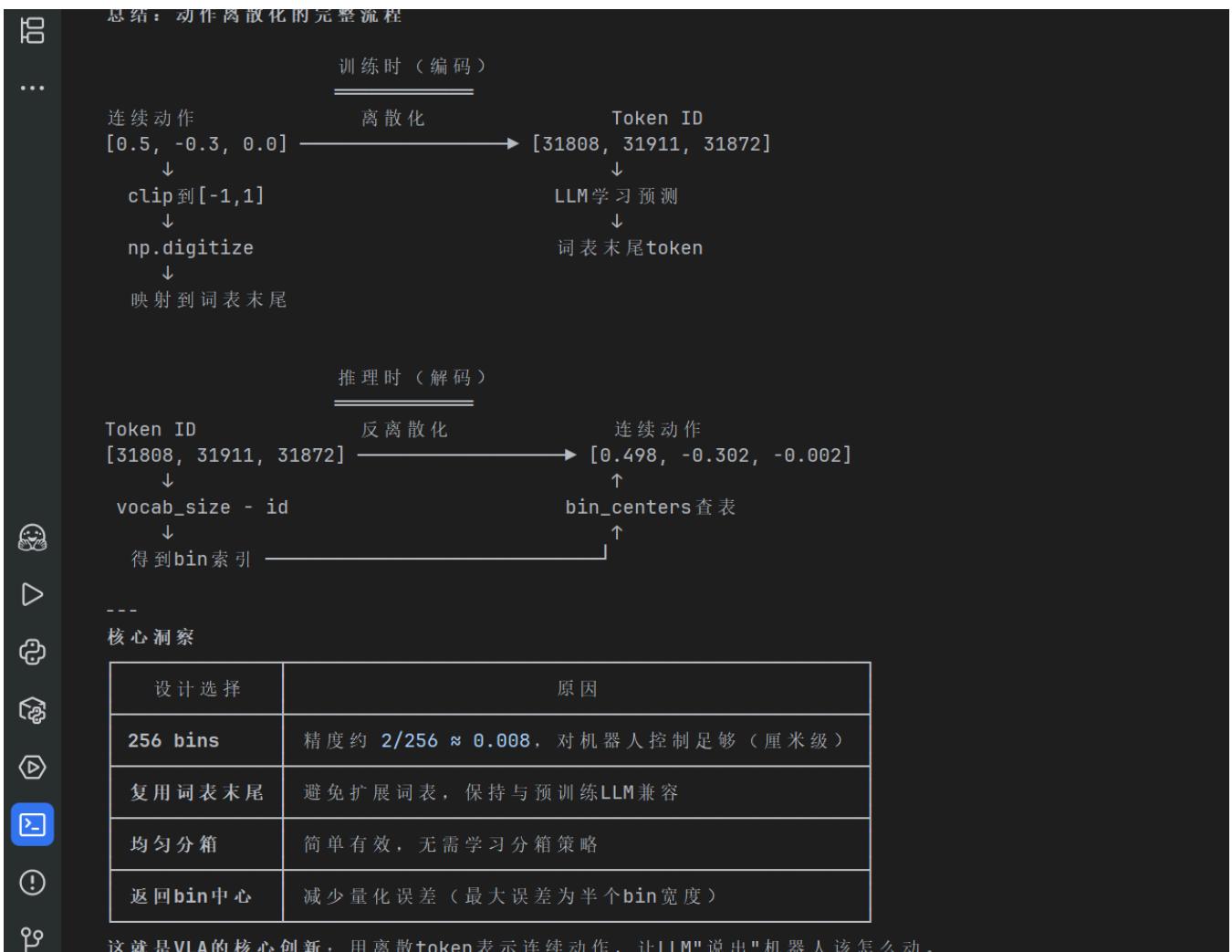
- 词表 = 一本字典
- token = 字典里的某个词条
- token id = 这个词条在字典里的“页码/编号”

比如（举例，不代表真实 LLaMA）：

- "robot" → id 1234
- "pick" → id 5678
- "up" → id 910

模型吃进去的不是 "robot pick up"，而是 [1234, 5678, 910] 这串数字。

好一个词元编号，真是一个精彩至极的设计，所以我们是对这个tokenid进行解码



到这个tokenid进行这个学习预测，到这个1-1的归一化进行这个输出具体连续的动作序列

2) Projector (MLP Adapter) : 负责“对齐维度/对齐表征”，让视觉特征能喂给LLM

它做什么？

Vision Backbone输出的向量维度是 D_v ，但 LLaMA 期望的 token 向量维度是 D_h (LLaMA hidden size)。

Projector 就是一个 MLP (多层感知机) 把：

$$\mathbb{R}^{D_v} \rightarrow \mathbb{R}^{D_h}$$

每个视觉 token 都做一次这样的映射。

你可以把它理解成：

- 视觉模型说的是“视觉方言”(维度、分布、语义空间都不同)

好一个对齐维度，对齐表征，充当了一个翻译官的作用

两个有着不同强项的目标检测

DINO 像“做视觉表征底座”，希望每一块都学得细；

SigLIP 像“做语义对齐”，希望整图语义对齐文本更稳。

在 OpenVLA 这种结构里

如果视觉 backbone 是 “DinoSigLIP” 这种组合/融合，通常也是想要：

- DINO 提供更强的空间结构与局部可用性
- SigLIP 提供更强的语义对齐能力（尤其与语言 token 对齐）

The screenshot shows a Jupyter Notebook interface with the following content:

解释：VLM由三部分组成：

```
PrismaticVLM
Vision Backbone (DinoSigLIP) → Projector (Adapter) (MLP) → LLM Backbone (LLaMA)
```

2. Projector初始化：视觉→语言的桥梁

```
if arch_specifier == "linear":  
    self.projector = LinearProjector(vision_backbone.embed_dim, llm_backbone.embed_dim)  
elif arch_specifier.endswith("fused-gelu-mlp"):  
    self.projector = FusedMLPProjector(vision_backbone.embed_dim, llm_backbone.embed_dim)  
elif arch_specifier.endswith("gelu-mlp"):  
    self.projector = MLPPProjector(vision_backbone.embed_dim, llm_backbone.embed_dim)
```

解释：

Projector类型	结构	说明
linear	单层线性	最简单
gelu-mlp	Linear→GEGLU→Linear	默认，效果好
fused-gelu-mlp	融合版MLP	性能优化

作用：将视觉特征维度（如1024）映射到LLM维度（如4096）

从1024维度，到这个4096维度，这个多层感知机解决这个维度和其他的东西不匹配的问题

5. 训练策略: `freeze_backbones` (重要)

```
def freeze_backbones(self, stage: str) -> None:
    if stage == "align":
        self.vision_backbone.requires_grad_(False)      # 冻结
        self.llm_backbone.requires_grad_(False)          # 冻结
        self.projector.requires_grad_(True)              # 只训练Projector

    elif stage in {"finetune", "vla-train"}:
        self.vision_backbone.requires_grad_(False)      # 冻结
        self.llm_backbone.requires_grad_(True)           # 训练
        self.projector.requires_grad_(True)              # 训练

    elif stage in {"full-finetune", "vla-full-train"}:
        self.vision_backbone.requires_grad_(True)         # 全部训练
        self.llm_backbone.requires_grad_(True)
        self.projector.requires_grad_(True)
```

解释：VLM的两阶段训练策略（来自LLaVA）：

阶段	Vision	Projector	LLM	目的
align	冻结	训练	冻结	对齐视觉-语言表征
finetune	冻结	训练	训练	学习任务能力
full-finetune	训练	训练	训练	端到端微调

4. 核心前向传播 `forward` (最重要)

4.1 视觉特征提取

有意思，这个冻结体现在这个维度的关闭的部分false部分

输入：
pixel_values: [1, 3, 224, 224] 图像
input_ids: [1, 50] 文本token IDs



Vision Backbone (DinoSigLIP)
[1, 3, 224, 224] → [1, 256, 1024]



Projector (MLP)
[1, 256, 1024] → [1, 256, 4096]



Text Embedding
[1, 50] → [1, 50, 4096]



Concatenate: [<BOS>] + [256 patches] + [49 text tokens]
→ [1, 306, 4096]



LLM Backbone (LLaMA)
[1, 306, 4096] → logits [1, 306, 32000]
→ loss (if labels provided)



这就是数据格式不断的转换吗，真是精彩

图像需要进行两次转换，一次是特征提取，一次是维度或者说格式对齐

文本只需要进行一次这个文本嵌入，相当于是图片的第二步骤

第三步骤，进行这个文本和这个图片相同格式的对齐，发给这个LLM大脑，推理出这个动作，可以在这个地方建立损失函数进行降低损失，反向传播，进行数据优化

然后最终结果进行解码为动作，记录数据

核心洞察机制

核心洞察

设计	原因
图像插在之后	LLM先“看”图，再处理文本，符合因果注意力
图像位置label=-100	不要求模型重建图像，只利用图像做条件
两阶段训练	先对齐表征空间，再学任务，稳定高效
Projector用MLP	非线性变换比线性效果好，参数量小

这就是VLM的骨架，OpenVLA在此基础上加了ActionTokenizer，让LLM能“说出”机器人动作。

* Churned for 55s

● deploy.py 关键要点

1. 核心架构

- C/S架构：FastAPI服务器 + REST API（/act端点）
- 输入：{"image": np.ndarray, "instruction": str}
- 输出：{"action": np.ndarray}（7维机器人动作）

2. 模型加载策略（显存优化关键）

```
BitsAndBytesConfig(  
    load_in_4bit=True,           # 4-bit 量化  
    bnb_4bit_quant_type="nf4",   # NormalFloat4 量化  
    bnb_4bit_compute_dtype=torch.float16  
)  
device_map="auto"             # 自动CPU/GPU分配  
max_memory={0: "5GiB", "cpu": "32GiB"} # 显存不足时offload到CPU
```

这个C/S架构还是非常有意思的

- 扣分 (Loss): 0.105 (非常低, 几乎满分)

学生 2: 学渣 (犹豫不决, 蒙对了)

- 回答: 呃... A 的概率是 0.3, B 是 0.3, C 是 0.4... (虽然 A 概率不高, 但它是正确答案里最高的, 所以算蒙对了)。
- 老师 (交叉熵) 评价: 虽然你选了 A, 但你心里没底啊。你需要更坚定一点!
- 扣分 (Loss): 1.20 (扣分中等, 惩罚它的不自信)

学生 3: 捣乱分子 (自信地选错了)

- 回答: 我敢打赌! A 的概率只有 0.01, D 的概率是 0.99!
- 老师 (交叉熵) 评价: 你不仅错了, 还错得这么离谱! 我要狠狠罚你!
- 扣分 (Loss): 4.60 (扣分极其巨大!)

核心直觉: 交叉熵不仅仅看你“选没选对”, 它更看重**“你对正确答案的预测概率有多高”**。

- 你对正确答案预测的概率越接近 1, 交叉熵 (Loss) 就越接近 0。
- 你对正确答案预测的概率越接近 0, 交叉熵 (Loss) 就会爆炸式增长 (趋向无穷大)。

交叉熵, 好一个对正确的预测概率有多高, 这也是损失函数的一种, 越低意味着预测的越稳定和精准

1. 分布式 (Distributed) —— FSDP 分片

截图里写着: FSDP 分片, 支持 64+ GPU 跨节点训练。

• 这是什么?

- 背景: OpenVLA 是一个 7B (70亿参数) 的巨型模型。如果想要全量训练它, 显存需求大得惊人, 单张甚至单台服务器 (8张显卡) 都装不下。
- FSDP (Fully Sharded Data Parallel): 这是一种“化整为零”的技术。它不让一张显卡死记硬背整个模型, 而是把模型切成几十份 (分片/Sharding)。
- 怎么工作: 比如有 64 张显卡, 每张卡只存 1/64 的模型参数。当需要计算时, 显卡之间疯狂交换数据, 拼凑出临时的完整参数。

好一个分布式, 真是符合我们的日常策略直觉

Open X-Embodiment Dataset (开放 X-具身智能数据集)。

openvla 正是因为吃了这么多数据集, 才能表现出如此强大的泛化能力