NanoGPT：从随机权重到指令遵循

摘要

本项目基于 Andrej Karpathy 的 build-nanogpt 项目，实现了一个完整的、从零开始的 GPT-2 模型训练与部署流程。项目首先在大型文本语料库 `FineWeb-Edu` 上对模型进行预训练，使其掌握通用的语言知识；随后，利用 `dolly-15k` 指令数据集进行监督式微调，赋予模型遵循人类指令的能力。为了提升工程实用性，项目实现了断点续训、分布式训练 (DDP) 等关键功能。项目通过一个与 OpenAI API 完全兼容的高性能 Flask 应用提供服务，该应用支持流式生成、高级采样参数，并配备了简洁的 Web 交互界面。

1. 引言

1.1 当前小型语言模型 (SLM) 领域情况

小型语言模型（Small Language Models, SLM）指参数量在几十亿甚至更少的语言模型。它们不是简单地将大模型等比例缩小，而是通过更优的模型架构、更高质量的训练数据和先进的训练技术，力求在小巧的体积内实现尽可能高的性能。

当前 SLM 领域呈现出百花齐放的态势，性能已经远超当年的 GPT-2，甚至在很多特定任务上可以媲美 GPT-3.5 级别的模型。前沿模型包括微软 Phi 系列、谷歌 Gemma & PaliGemma 系列、阿里巴巴 Qwen (千问) 系列等。SLM 的应用主要包括端侧部署和专用化与垂直领域应用等，其小巧的规模为微调和部署提供了便利。

在大型语言模型日益发展的当下，模型蒸馏与人类反馈对齐技术也成为训练高性能 SLM 的重要方法。

1.2 模型架构选择

本项目期望通过回归经典模型以理解语言模型的基础工作原理，并完成端到端的全流程训练和部署实践。GPT-2 采用标准的 Decoder-only Transformer 架构，包含了现代大语言模型 (如 GPT-3，LLaMA，Qwen) 的核心元素——多头因果自注意力 (Multi-Head Causal Self-Attention) 和前馈神经网络 (Feed-Forward Network)。

本项目选用其 124M 的小型版本。该规模便于在有限的计算资源下，进行完整的预训练和微调。

1.3 本项目的主要工作

本项目的主要贡献可以归结为以下四个方面：

1. 实现了端到端的完整开源流程：提供了一套完整的从数据准备、模型预训练、指令微调到最终部署的开源解决方案，具有可复现性。
2. 强化了工程化的训练能力：显著增强了项目的工程实用性，实现了断点续训等关键功能，确保了在真实硬件环境下进行大规模、长时间训练的稳定性和效率。
3. 验证了指令微调的有效性：成功在经典的 GPT-2 架构上应用了指令微调技术，并证明了该方法能有效引导模型遵循人类指令，显著提升了其在特定任务上的表现。
4. 提供了高性能的兼容 **API**：我们开发并提供了一个与 OpenAI 标准完全兼容的 API 接口，支持流式生成等高级功能，确保了模型可以轻松集成到现有生态系统和应用中。

2. 系统设计与模型架构

本项目的核心是一个基于 GPT-2 架构的语言模型，其实现严格遵循了原始论文和 `train_gpt2.py` 中定义的核心组件。

2.1 核心模型 GPT

模型的主体是 GPT 类，它整合了嵌入层、多个 Transformer 解码器块和最终的语言模型头。

2.1.1 模型结构 GPT 类的整体结构定义在 nn.ModuleDict 中，包含了四个关键部分：

1. **wte (Word Token Embedding)**: 词嵌入层，将输入的 token 索引映射为 n_embd 维度的向量。
2. **wpe (Word Position Embedding)**: 位置嵌入层，为序列中的每个位置生成一个 n_embd 维度的向量，以提供位置信息。
3. **h (Transformer Blocks)**: 包含 n_layer 个 Block 模块的 nn.ModuleList，是模型的核心部分。
4. **ln_f (Final LayerNorm)**: 在输出到语言模型头之前应用的最终层归一化。

最终，通过一个线性层 lm_head (语言模型头)，将 Transformer 的输出映射到整个词汇表 (vocab_size) 的 logits 分布上。

2.1.2 前向传播流程 模型的前向传播 (forward 方法) 流程如下：

1. 输入处理: 接收形状为 (B, T) 的 token 索引 idx。
2. 嵌入计算:
   - 通过 wte 获得 token 嵌入 tok_emb。
   - 通过 wpe 获得位置嵌入 pos_emb。
   - 将两者相加得到融合了内容和位置信息的输入表示 x。
3. **Transformer** 堆叠处理: 输入 x 依次通过 n_layer 个 Block 进行深度特征提取。
4. 最终输出:
   - 通过最终的 ln_f 进行归一化。
   - 通过 lm_head 计算得到 (B, T, vocab_size) 形状的 logits。
   - 如果提供了 targets，则计算并返回交叉熵损失。

2.1.3 权重初始化与共享

- 权重共享: 为了节约参数并遵循 GPT-2 的设计，lm_head 的权重与 wte 的词嵌入权重是共享的。
- 权重初始化: _init_weights 方法对模型参数进行初始化。线性层权重采用均值为 0、标准差为 0.02 的正态分布初始化，而嵌入层也采用类似的正态分布。特别地，对于残差连接路径上的投影层 (c_proj)，其权重标准差会根据 n_layer 进行缩放 (std *= (2 * self.config.n_layer) ** -0.5)，这是维持训练稳定性的关键技巧，可以防止残差累积过程中的数值爆炸。

2.2 Transformer 核心模块 Block

Block 是构成 GPT 模型的基本单元，其结构采用了 **Pre-LN (预归一化)** 形式，这种结构将 LayerNorm 置于子模块（自注意力和 MLP）之前，有助于稳定训练过程中的梯度。

每个 Block 的数据流如下：

```
# 第一个子层：多头因果自注意力
x = x + self.attn(self.ln_1(x))

# 第二个子层：前馈神经网络
x = x + self.mlp(self.ln_2(x))
```

残差连接使得信息和梯度能够更顺畅地在深层网络中流动。

2.3 关键子模块

2.3.1 `CausalSelfAttention` (因果自注意力) 此模块是实现序列信息交互的核心。

- **QKV** 统一计算: 使用单个线性层 `c_attn` 将输入 `x` 从 `n_embd` 维投影到 `3 * n_embd` 维，然后一次性分割出 Q (Query), K (Key), V (Value) 三个张量，提高了计算效率。
- 多头机制: 将 Q, K, V 的嵌入维度 `C` 分割成 `n_head` 个头，每个头的维度为 `hs = C / n_head`。这使得模型能从多个不同的表示子空间并行学习信息。
- **Flash Attention**: 核心的注意力计算直接调用 `F.scaled_dot_product_attention(is_causal=True)`。`is_causal=True` 参数会自动应用因果掩码，确保每个 token 只能关注其自身及之前的位置，这是语言模型生成文本的必要条件。底层实现利用了 Flash Attention，这是一种内存高效且计算速度更快的注意力算法，避免了显式构造 `(T, T)` 注意力矩阵带来的内存瓶颈。
- 输出投影: 多头注意力的输出被重新组合，并通过一个最终的线性层 `c_proj` 投影回 `n_embd` 维度。

2.3.2 `MLP` (前馈神经网络) 这是模型中另一个关键的非线性处理单元。

- 扩展-收缩结构: 采用两层全连接网络。第一层 `c_fc` 将维度从 `n_embd` 扩展到 `4 * n_embd`，第二层 `c_proj` 再将其收缩回 `n_embd`。这种结构为模型提供了强大的特征变换能力。
- **GELU** 激活函数: 中间使用 `GELU` (Gaussian Error Linear Unit) 激活函数，其平滑的非线性特性相比 ReLU 更有利于模型的训练和性能。代码中使用了 `approximate="tanh"` 的 GELU 近似版本，以获得更好的计算性能。

2.4 优化器配置 `configure_optimizers`

`configure_optimizers` 方法展示了精细的优化策略：

- 参数分组: 将模型参数分为两组：
  1. 需要权重衰减 **(Weight Decay)** 的参数: 所有维度大于等于 2 的张量（主要是矩阵乘法中的权重 `weight` 和嵌入层 `nn.Embedding`）。
  2. 无需权重衰减的参数: 维度小于 2 的张量（如偏置 `bias` 和层归一化 `LayerNorm` 的参数）。这种做法可以防止对偏置和归一化参数施加不必要的惩罚，是现代深度学习训练中的标准实践。
- **AdamW** 优化器: 使用 AdamW 优化器，它将权重衰减与梯度更新解耦，通常能带来比标准 Adam 更好的性能。
- **Fused AdamW**: 在 CUDA 环境下，代码会自动检测并使用 `fused` 版本的 AdamW，该版本将多个计算核心操作融合，能显著提升 GPU 上的训练速度。

3. 数据集与数据处理

高质量的数据是成功训练语言模型的基础。本项目涉及三种不同类型的数据集，每种都通过专门的脚本进行处理。

3.1 预训练数据: FineWeb-Edu

- 来源: HuggingFaceFW/fineweb-edu，一个经过高质量过滤和去重的教育内容网络文本数据集。
- 处理脚本: `data_prep/fineweb.py`
- 处理流程:
  1. 下载: 使用 `datasets` 库下载 `sample-10BT` 配置的数据集。
  2. **Tokenization**: 使用 `tiktoken` 的 `gpt2` 编码器进行并行化 tokenization。每个文档前都添加 `<|endoftext|>` (EOT) 特殊 token 作为分隔符。

3. 分片 **(Sharding)**: 将 tokenized 数据流切分为大小为 1 亿 token 的分片 (`.npy` 文件)，以便在训练时高效加载。第一个分片被指定为验证集 (`val`)，其余为训练集 (`train`)。这种策略确保了训练和验证数据来自同一分布，但又严格分离。

3.2 指令微调数据**: Dolly-15k**

- 来源: databricks/databricks-dolly-15k，一个由 Databricks 员工编写的高质量人工指令数据集。
- 处理脚本: `data_prep/prepare_dolly.py`
- 处理流程:
  1. 格式化: 每个样本统一格式化为包含 `### Instruction:`，`### Context:` (如有) 和 `### Response:` 的结构化字符串。这种清晰的结构有助于模型学习指令遵循的模式。
  2. **Tokenization**: 与预训练数据类似，使用 `gpt2` 编码器处理格式化后的字符串，并在每个样本的末尾添加 EOT token。
  3. 切分与保存: 在 tokenization 之前，整个数据集按比例（默认为 10%）随机切分为训练集和验证集。然后，每个子集被独立地 tokenized 并保存为分片文件。

3.3 自定义数据扩展

- 来源: 用户提供的 `.jsonl` 文件，每行包含 `instruction`, `context` (可选), `response` 字段。
- 处理脚本: `data_prep/prepare_custom_dataset.py`
- 处理流程: 该脚本提供了一个通用框架，用于处理任何符合特定格式的自定义指令数据集。它执行与 `prepare_dolly.py` 类似的步骤：读取、随机打乱、按比例切分、格式化、tokenization 并保存。这极大地增强了项目的可扩展性，允许用户使用自己的数据对模型进行特定领域的微调。

4. 模型训练与微调

本项目的训练分为两个核心阶段：预训练和指令微调。两个阶段共享了许多工程实践，但在数据加载和损失计算方面存在关键差异。

4.1 预训练阶段

- 脚本: `model/train_gpt2.py`
- 目标: 在大规模无标签文本上训练模型，使其学习通用的语言规律、语法和世界知识。
- 数据加载: `DataLoaderLite` 在此阶段将数据视为一个连续的 token 流。它按顺序读取数据分片，并从中切分出大小为 `(B, T)` 的批次，其中 `B` 是批次大小，`T` 是上下文长度。这种方法对于语言建模任务是最高效的。
- 训练循环:
  1. 学习率调度: 采用带预热 (warm-up) 的余弦退火衰减策略 (`get_lr` 函数)，在训练初期线性增加学习率，然后在剩余的训练步骤中余弦退火平滑衰减。
  2. 梯度累积: 为了在有限的显存下模拟大批次训练，脚本通过 `grad_accum_steps` 参数累积多个 micro-batch 的梯度，然后执行一次优化器步骤。
  3. 损失计算: 对整个上下文窗口内的所有 token 计算交叉熵损失，促使模型学习预测序列中的下一个词。

4.2 指令微调阶段

- 脚本: `model/finetune_dolly.py`
- 目标: 在小规模有标签的指令数据上进行微调，教会模型理解并遵循指令，生成有用的回答。

- 数据加载: 重新设计 `DataLoaderLite` 以处理独立的指令样本。首先将 token 流按 EOT 分割成单独的样本，然后在每个批次中随机选择 `B` 个样本，并将其填充 (padding) 到该批次中最长样本的长度。
- 损失遮罩 **(Loss Masking)**: 这是指令微调的核心优化。为了让模型专注于学习生成回答，损失函数只在指令的 `### Response:` 部分计算。输入给模型的 `targets` 张量中，非回答部分的 token 被设置为一个特殊的忽略值 (`-100`)，这样 `F.cross_entropy` 在计算损失时会自动忽略它们。这极大地提升了微调的效率和效果。
- 检查点策略: 微调脚本清晰地区分了两种检查点加载方式:
  - `--resume 'auto'`: 从最新的微调检查点恢复完整的训练状态（包括优化器和学习率）。
  - `--pretrained_checkpoint`: 仅加载预训练模型的权重，并从头开始微调。

4.3 关键技术实现

- 分布式数据并行 **(DDP)**: 两个训练脚本都通过 `torchrun` 支持 DDP。在 DDP 模式下，每个 GPU 进程都拥有模型的完整副本，并处理一部分数据。在反向传播后，梯度会在所有进程间进行 all-reduce 同步，确保所有模型副本的权重更新保持一致。这使得训练能够水平扩展到多张 GPU，显著缩短训练时间。
- 断点续训: 通过精心设计的 `save_checkpoint` 和 `load_checkpoint` 函数，项目实现了稳健的断点续训能力。检查点不仅保存了模型权重，还保存了优化器状态、数据加载器位置以及所有相关的随机数生成器状态 (PyTorch, NumPy, Python random)。这确保了从中断处恢复的训练与原始训练过程完全一致。

5. 实验与性能评估

5.1 标准化评估: Hellaswag

- 评估工具: `eval/hellaswag.py`
- 评估任务: Hellaswag 是一个常识推理任务，要求模型在给定上下文后，从四个选项中选择最合乎逻辑的句子结尾。
- 评估方法: 在预训练过程中，模型会定期在 Hellaswag 验证集上进行评估。评估方法是计算每个选项（上下文 + 结尾）的平均困惑度（通过交叉熵损失实现）。损失最低的选项被认为是模型的预测。
- 结果: 预训练日志显示，在训练结束时（约 19072 步），模型的 Hellaswag 准确率（`acc_norm`）达到了约 **29.55%**。这个结果与公开的、同等规模的 GPT-2 模型在该任务上的基线性能基本一致，验证了我们预训练的有效性。

5.2 定性分析

指令微调的成功与否，最终需要通过模型生成的具体例子来检验。`finetune_dolly.py` 中的 `generate_samples` 函数在训练过程中定期生成示例文本，让我们得以观察模型的进化。

微调前 (仅预训练): 当被问及 “What is machine learning?” 时，模型可能会生成类似以下不连贯或不相关的文本:

```
Machine learning is a type of AI that uses artificial intelligence to create machine-like models that
How does machine learning work?
Machine learning is a type of AI that uses artificial intelligence to make decisions based on data th
Is machine learning a bad thing?
Machine learning is a type of AI that is designed to improve the performance of tasks that require hu
```

微调后: 在经过 Dolly-15k 数据集的微调后，模型对相同问题的回答变得结构化且切题：

```
Machine learning is a branch of artificial intelligence which helps us to identify patterns in data t
What are the benefits of machine learning?
The benefits of machine learning include:

-   Data-driven decision making
-   Data-driven decision making
-   Predictive analytics
-   Machine learning algorithms
-   Machine learning algorithms
-   Machine learning applications
-   Data analytics
-   Data-driven decision making
-   Data
```

这个对比鲜明地展示了指令微调的有效性。模型已经学会了遵循 “指令-回应” 的结构，能够提供一个切题的定义，并尝试列出相关的益处。这与微调前生成的、不知所云的文本形成了巨大反差。

然而，这个样本也暴露了模型的局限性。在定义部分出现了 “make better decisions and make better decisions” 这样的直接重复，而在列举优点时，重复问题则更为严重，多个要点（如 “Data-driven decision making”）被多次列出，且 “Data” 这样的单词条目意义不大。

因此，我们可以得出结论：指令微调成功地引导了模型的基本行为，使其从生成无意义的文本转变为提供有结构、相关的回答。但要实现更高层次的语义连贯性和内容多样性，还需要更优质、更多样化的微调数据或更先进的训练技术。

5.3 API 性能

项目通过以下方式优化 API 的性能：

- **Gevent Worker**: `gunicorn_config.py` 中使用 `gevent` 作为 worker 类型。Gevent 基于协程，能以非阻塞方式高效处理大量并发的 I/O 密集型请求（如网络连接），这对于一个需要同时服务多个用户的 API 至关重要。
- 模型缓存: `web/app.py` 实现了一个简单的模型缓存 (`MODEL_CACHE`)。一旦模型被加载到内存中，后续对同一模型的请求将直接从缓存中获取，避免了昂贵的磁盘 I/O 和模型初始化开销。
- 流式生成: API 支持流式响应 (`stream=True`)，允许客户端在第一个 token 生成后立即开始接收内容，极大地改善了用户感知的响应速度。值得注意的是，对流式生成的支持是与 LobeChat 等 AI 应用集成的重要条件。非流式 API 需要等待模型生成全部内容后才一次性返回结果，这个过程可能长达数十秒。对于 LobeChat 这样的实时应用来说，长时间的等待很容易触发网络超时而导致连接失败。相比之下，流式生成会像打字一样，将生成的内容逐字逐句地持续 “推送” 给客户端，不仅能立即看到响应，也保持了连接的持续活跃，从而完美规避了超时问题。

6. 应用与部署

本项目的最终交付物是一个功能齐全、可部署的 Web 应用，它通过与 OpenAI 兼容的 API 提供模型能力。

6.1 Web 应用接口

- 前端: `web/templates/index.html` 提供了一个基础的交互界面，用户可以在网页上直接与模型进行对话。
- 后端: `web/app.py` 是一个 Flask 应用，作为整个服务的核心。它负责处理 HTTP 请求，并调用推理逻辑。

6.2 OpenAI 兼容 API

这是本项目的核心亮点之一，确保了与现有 AI 生态系统的无缝集成（如 LobeChat、各种编程语言的 OpenAI 库等）。

- 端点实现: `web/app.py` 精心实现了以下关键端点：
  - `/v1/models`: 列出所有在 `MODELS_DIR` 中找到的可用模型检查点。
  - `/v1/chat/completions`: 核心的聊天端点。它接收与 OpenAI API 格式完全相同的 JSON 请求体。
- 功能支持:
  - 认证: 通过 `require_api_key` 装饰器实现简单的 Bearer Token 认证。
  - 参数兼容: 支持 `model`, `messages`, `stream`, `temperature`, `top_k`, `top_p`, `presence_penalty`, `frequency_penalty` 等常用参数，并对数值范围进行了合理约束。
  - 流式响应: 当请求中 `stream=True` 时，端点返回一个 `text/event-stream` 响应，逐个 token 地推送生成的内容，并以 `data: [DONE]` 结束。

6.3 生产环境部署

- 服务器: 使用 Gunicorn 作为生产环境的 WSGI 服务器。
- 配置: `gunicorn_config.py` 提供了生产环境的配置。
  - **Worker** 类型: 明确指定 `worker_class = 'gevent'`，以利用其高效的并发处理能力。
  - 模型加载: 通过 `post_fork` 服务器钩子，确保每个 Gunicorn worker 进程在启动时独立加载模型。这种模式避免了在主进程中加载模型然后 fork 导致的潜在问题，并确保了 worker 间的隔离。

7. 结论与展望

7.1 结论

本项目成功地完成了一次构建小型语言模型的端到端实践。从一个基础的 GPT-2 Pytorch 实现出发，项目通过严谨的工程实践，构建了一个包含数据处理、大规模预训练、指令微调、评估和生产级部署的完整流程，验证了即使是像 GPT-2 这样的经典架构，在经过高质量数据和指令微调的塑造后，也能展现出显著的行为改善，从生成无意义的重复文本转变为提供结构化、与指令相关的回答。

然而，定性分析清晰地显示了仅通过基础监督式微调（SFT）所能达到的上限。模型虽然学会了遵循指令格式，但其生成内容的连贯性和多样性仍有不足，表现出明显的重复性。这说明，要实现更高质量的生成效果，单纯依赖现有数据集进行 SFT 是不够的。

尽管存在这些局限，本项目最终产出的 OpenAI 兼容 API，使得这个小巧的模型能够轻松地融入现代 AI 应用生态，是一次成功的实践。

7.2 展望

基于本次实践的发现，未来的工作可以从以下几个方面展开，以克服当前模型的局限性：

- 提升数据质量与多样性: 这是解决内容重复问题的关键。可以引入更多样化的指令数据集，或者对现有数据进行清洗和增强，剔除低质量、重复性高的样本，增加数据在风格、主题和复杂性上的变化。
- 探索高级微调技术: 监督式微调只是第一步。为了让模型更好地对齐人类偏好，减少不合逻辑或重复的输出，可以引入直接偏好优化 (DPO) 或人类反馈强化学习 (RLHF) 等更先进的对齐技术。
- 模型架构升级: 探索更现代的 SLM 架构，如 LLaMA 或 Mistral 的变体。它们通常包含 Grouped-Query Attention (GQA)、Sliding Window Attention (SWA) 和 SwiGLU 等优化，这些结构上的改进本身就能在同等参数规模下提升模型的性能和生成质量。
- 量化与性能优化: 对训练好的模型进行量化（如 4-bit 或 8-bit），以减小模型体积和显存占用，并提升在 CPU 或端侧设备上的推理速度。
- 更全面的评估: 引入更多维度的评估基准，如 MMLU、TruthfulQA 等，以更全面地衡量模型的综合能力，并为未来的改进提供更精确的指导。