一、 深度学习科研代码管理和任务管理面临的问题

深度学习模型结构变化程度大、参数多，如何高效地进行代码和任务管理?

1. **环境管理**：如何保证代码运行环境可复现；
2. **模块管理**：如何划分模型的核心和任务，以及一些辅助的目录；
3. **分支管理**：如何划分 git 分支；
4. **开发流程**：如何规划开发流程；
5. **大文件管理**：如何管理数据集、模型文件等大型二进制文件；
6. **参数管理**：如何管理训练参数；
7. **调试模式**：如何实现小数据集的调试模式，避免执行用时过长；
8. **断点处理**：如何避免 Python 中的错误中断，以及如何进行断点训练；
9. **进度管理**：如何可视化训练进度、训练用时；
10. **结果管理**：如何管理日志、输出数据和任务结果；
11. **性能测量**：如何测量代码执行的效率，找出瓶颈和优化点；
12. **图表管理**：如何管理论文中可能用到的文档图表；
13. **文档管理**：如何编写和管理文档，包括笔记、报告、论文和 slides。

二、 环境管理

使用 `requirements.txt` 和 miniconda[1] 来管理项目的环境。

conda 新建环境：

```
conda create -n envName python=3.9
```

conda 查看所有环境：

```
conda env list
```

conda 删除环境：

```
conda env remove -n envName
```

生成 `requirements.txt` 文件：

```
pip freeze > requirements.txt
```

安装 `requirements.txt` 依赖：

[1] https://docs.conda.io/projects/miniconda/en/latest/

```
pip install -r requirements.txt
```

使用 conda 安装 `requirements.txt` 依赖：

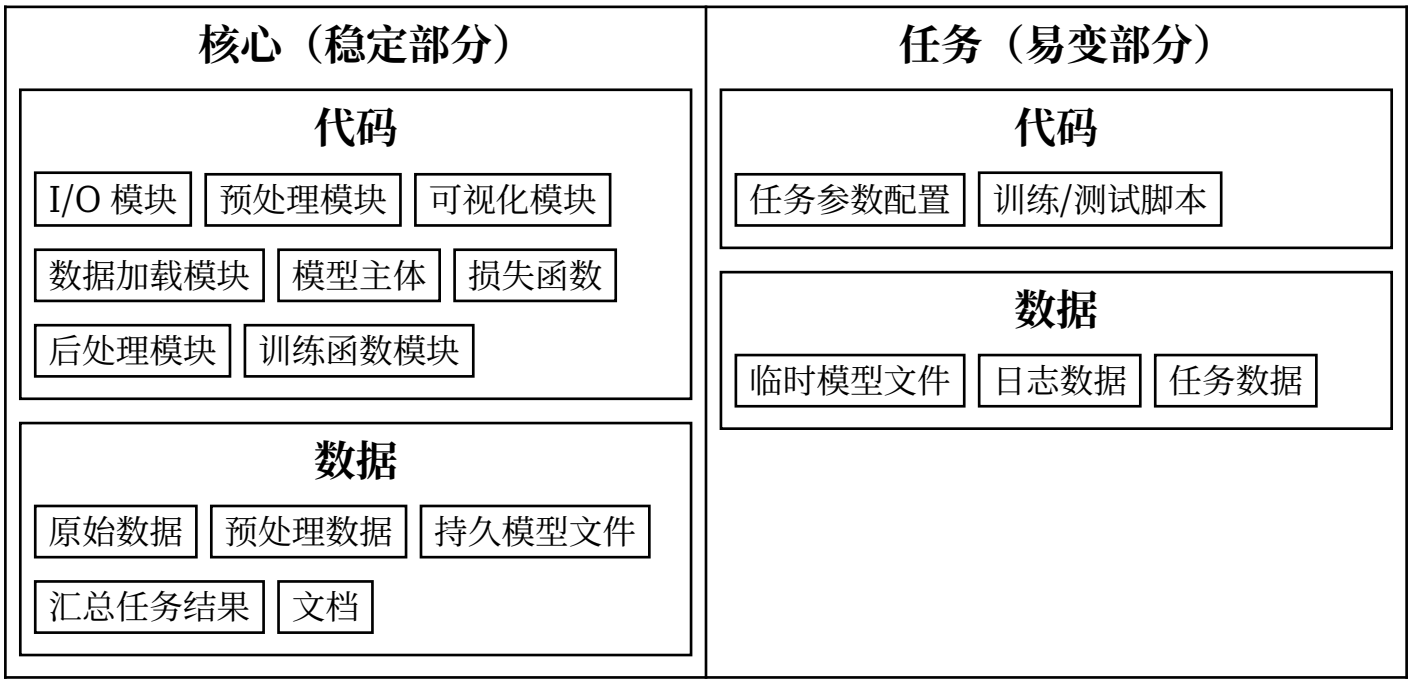
```
conda install --yes --file requirements.txt
```

使用 conda 安装 `requirements.txt` 依赖（失败时用 pip 重试）：

```
while read requirement; do conda install --yes $requirement || pip install
$requirement; done < requirements.txt
```

三、 模块管理

良好的目录划分和模块划分是可维护性的关键。一般根据代码的易变程度可以分为这两大块：



核心（稳定部分）不易变，我们可以通过「版本号」来唯一标识，因此可以直接放在根目录下，并直接依靠 git 管理。

任务（易变部分）主要是涉及到更改任务参数后的一次次的任务执行与结果记录，同一个核心版本可能会有多次不同的任务，这些任务的任务结果我们都应该记录下来，并且最后我们应该可以自动化地汇总所有的任务结果并整理任务报告。

因此一个比较好的办法就是创建一个 `tasks` 目录，并在里面创建一个「当前分支名」目录，例如对于 `main` 分支就是 `tasks/main` 目录，里面放置了模型参数文件 `config.yaml` 和训练脚本，创建任务分支时就会复制 `tasks/main` 目录到 `tasks/task-version-time-xxx` 目录，更改超

参数等工作均在这个「易变部分」的目录里完成，并且可以在外层封装一个根据当前分支名来选择执行的 Python 文件的简单脚本。

代码模块化非常重要，部分人可能会把系统写得过于耦合，例如损失函数和模型写到一起，这就导致了牵一发而动全身。我们应该把不同的模块互相解耦，例如划分数据加载、模型主体和损失函数等模块，最后在「任务」里的训练脚本和测试脚本里把这几个模块组装起来，这样既可以让 Yaml 配置文件可以有不同的结构，也可以使用 `importlib` 动态选择导入具体的模型，或者导入哪一个子模型。

四、 分支管理

我们应该使用 git 分支管理管理好我们的主分支和任务分支，并且分支的命名要有一定的规则，必要的话可以通过脚本来自动化。

这里的分支管理规则参考了常见的 Git Flow[1] 开发流程（推荐 10 到 20 人共同开发），并根据深度学习科研的代码管理特性进行了一定的改良。

这里是理论上的较复杂项目的最佳实践，可以根据自己的情况进行一定程度的化简，例如去除 `dev` 分支，只留下 `main` 分支并解除相应限制。

4.1 main 主分支

长期分支，应该始终保持为一个最新的可用版本，即版本发布状态。

`main` 主分支需要有最严格的权限管理，不能直接提交 commit 到 `main` 分支，只能合并其他分支，因此最好开启分支保护。只有在 `dev` 分支上的代码达到一个稳定状态之后，才能合并到 `main` 分支。

简单的 `hotfix-xxx` 分支（数行代码的修改）可以在确认无误后合并到 `main` 分支（也需要同步到 `dev` 分支）。除了 `dev` 和 `hotfix-xxx` 分支以外的分支都不能合并到 `main` 分支。

合并命令：

```
git checkout main
git merge --squash dev
```

这里使用 squash merge 可以将 `dev` 分支新增的 commits 压缩为一个 commit 并加入 `main` 中。这样做能够让 `main` 分支的 commit 记录保持简洁。

但是这样 main 的 commit 记录提交者就变成了执行 merge 操作的那个人，所以最好是由特性分支的开发者执行 merge。为了解决记录提交者问题，也可以考虑常规的非快进模式 merge `git --no-ff merge dev` 或 rebase merge[2]。

[1]https://www.ruanyifeng.com/blog/2012/07/git.html
[2]https://www.jianshu.com/p/ff1877c5864e

在 merge 之后还需要 `dev` 重新 merge 一下 `main` 分支的最新提交，这样才能让 `dev` 保持最新的 commit 记录，方便下次 merge。

```
git checkout dev
git merge main
```

4.2 dev 开发分支

长期分支, 用于保持最新的开发进度, 主要开发工作都应该在这个分支进行。如果你能确定 main 分支不需要那么强的稳定性，也可以去除 `dev` 分支，只使用 `main` 分支和 `feature-xxx` 分支。

由于深度学习科研中的代码常常仅由单人（或几个人）进行开发，所以应当允许在 dev 分支直接进行开发和提交 commit。

`dev` 分支可以合并任何其他分支，例如 `feature-xxx` 和 `hotfix-xxx` 分支。

4.3 feature-xxx 特性分支

临时分支，用于开发某一个特性模块，例如预处理模块、可视化模块等。

特性分支一般是在多人开发的情形下使用，能够方便地进行并行化开发，也可以用于区分不同模块的开发，让开发过程更具区分性。

一般一个特性分支专注于一个功能，而且应该只是临时的，不应该长期存在。完成后应该先合并至 `dev` 分支，确认稳定后再经由 `dev` 分支合并到 `main` 分支。

4.4 hotfix-xxx 修复分支

临时分支，仅用于 bug 的紧急修复。

`hotfix-xxx` 分支是为了应对 `dev` 分支还不稳定，不能合并到 `main` 分支，但是又需要紧急修复的情况而使用的，否则可以直接在 `dev` 分支上修复。

分支内不应该包含过多的删除或新增代码，即不应该是实现功能，而应该是修复功能。

`hotfix-xxx` 分支可以直接合并到 `dev` 和 `main` 分支。

4.5 task-version-time-xxx 任务分支

临时分支，用于修改任务参数，然后执行一次任务，并记录结果。

只能从 `main` 或其他任务分支 `task-yyy` 中分支出来（模型变更后需要先更新核心版本，并打一个 git tag），并且应该由一个自动化脚本创建，根据核心版本-时间戳-模型参数的维度命名，便于辨识。

只能修改 `tasks/task-version-time-xxx` 目录下的文件，然后执行一次任务，并记录结果。如果认为需要修改其他目录的代码，请考虑是否应该新增一个超参数用于控制，或者是否应该在其他如 `dev`、`feature-xxx` 和 `hotfix-xxx` 分支中进行修改。

记录的内容可以包括：

- 复制 `tasks/main` 目录下的文件到 `tasks/task-version-time-xxx` 目录；
- 记录代码执行的 log 日志；
- 需要持久化的模型文件；
- 记录任务的原始图表数据文件（便于后续生成图表）；
- 任务的具体输出结果；
- 也许可以保留的 checkpoint models。

由于没有代码层面的变更，执行完成后可以直接合并到 `dev` 和 `main` 分支中。

4.6 debug-version-time-xxx 调试分支

临时分支，用于在 debug 模式下跑一次任务，用于验证是否能正常地跑通训练流程。

可以从 `main` 或 `dev` 中分支出来，并且应该由一个自动化脚本创建。

该分支的变更都不应该保存到其他分支里，包括数据变更和代码变更。如果有代码的修复，可以使用 `git stash` 暂存代码并切换到 `dev` 或 `hotfix-xxx` 分支修复。

4.7 commit 提交规范

常规规范：

- `feat`：新功能（feature）
- `fix`：修复 bug
- `docs`：文档（documentation）
- `style`：代码风格（不影响代码运行的变动）
- `refactor`：重构（即不是新增功能，也不是修改 bug 的代码变动）
- `test`：增加测试
- `chore`：构建过程或辅助工具的变动

深度学习科研代码新增规范：

- `task`：合并一个 `task-version-time-xxx` 分支的任务执行结果。

在 merge 到 `main` 主分支时，merge 的 commit 信息也应该遵循这个规范。

五、 开发流程

1. **初始化**：从 `main` 分支开始，通过 `git checkout -b dev` 创建并切换到 `dev` 分支。
2. **模块开发**：
    1. 在 `dev` 分支 `git checkout -b feature-xxx` 创建并切换到 `feature-xxx` 分支。
    2. 完成 `feature-xxx` 分支的开发工作。
    3. 执行 `git checkout dev` 切换到 `dev` 分支。
    4. 执行 `git merge feature-xxx` 合并 `xxx` 模块的代码到 `dev` 分支。

3. **调试代码**:
    1. 在 `dev` 分支执行自动化脚本，实现
        1. 执行 `git checkout -b debug-version-time-xxx` 切换到调试分支。
        2. 复制 `tasks/main` 目录到 `tasks/debug-version-time-xxx` 目录。
    2. 在 `tasks/debug-version-time-xxx` 目录以 debug 模式执行一次任务，检验代码逻辑是否有误。
    3. 发现错误后，进行代码的修复，修复完后重新验证。
    4. 通过 `git add <code-files>` 和 `git stash`，仅保留代码变更，不保留日志文件等临时输出数据文件。
    5. 可以提交 commit 到 `debug-version-time-xxx` 分支，将日志文件等数据保留，或者直接 discard 遗弃。
    6. 执行 `git checkout dev` 切换到 `dev` 分支。
    7. 执行 `git stash pop` 将修改后的代码取出，并给 `dev` 分支提交 commit。
4. **合并代码**:
    1. 执行 `git checkout main` 切换到 `main` 分支。
    2. 执行 `git merge --squash dev` 或 `git merge --no-ff dev` 将 `dev` 分支的变更合并到 `main` 分支。
    3. 在 `tasks/main` 的 Yaml 配置文件中给 `VERSION` 自增版本号，并提交 commit。
    4. 执行 `git tag -a v0.1.0 -m "my version 0.1.0"` 给版本号打上 tag。
    5. 切换到 `dev` 分支并通过 `git merge main` 合并 `main` 的最新版本。
5. **执行任务**:
    1. 在 `main` 分支执行自动化脚本，实现
        1. 执行 `git checkout -b task-version-time-xxx` 切换到任务分支。
        2. 复制 `tasks/main` 目录到 `tasks/task-version-time-xxx` 目录。
    2. 修改 `tasks/task-version-time-xxx` 目录下的 Yaml 配置文件。
    3. 执行 `tasks/task-version-time-xxx` 目录下的训练或测试脚本。
    4. 将执行结果通过 commit 提交到任务分支上。
    5. 执行 `git checkout main` 切换到 main 分支。
    6. 合并任务结果到 `main` 分支。

六、 大文件管理

我们使用数据版本控制（DVC）进行大文件的管理，其允许在 Git 提交中数据和模型的版本，同时将它们存储在本地或云存储中。

我们既可以在命令行中使用[1]，也可以在 VS Code 中安装 DVC 插件来使用。

6.1 跟踪数据文件

```
dvc add data/data.xml
```

[1]https://juejin.cn/post/7057767026072223751

这个命令会将文件信息存储到名为 `data/data.xml.dvc` 的特殊文件中，并且会将原始数据放在 `.gitignore` 文件中。然后将数据移动到目录的缓存 `.dvc/cache` 中，并将其链接回工作区。

所以我们可以放心地将这些元信息文件放入 git 仓库中托管，如果数据文件发生了更改，则我们应该重新执行 `dvc add` 命令来追踪更改，这会更新 `data/data.xml.dvc` 文件，进而让我们能在 git 中记录当前的数据版本。

6.2 远程存储库同步

可以使用 `dvc push` 上传 DVC 跟踪的数据或模型文件，后续也可以通过 `dvc pull` 进行恢复。

我们需要设置一个原创存储库的地址，例如 Amazon S3、SSH、Google Drive 等。

6.3 在版本之间进行切换

当使用 `git checkout xxx` 切换了分支之后，我们可以执行

```
dvc checkout
```

来切换数据文件的版本，也即 DVC 会根据所有的 `.dvc` 文件来切换到正确的数据文件版本。

可以看出 DVC 的实现原理还是相对简单的，整体小巧而精致，但是可以很有效地和 git 共同工作，维护我们的数据文件，并且保证 git repo 的体积不会过于庞大。

七、 参数管理

什么是参数? 参数是运行一次任务所需要配置的所有输入参数。在任意时刻，一台机器上应当最多只能运行一个任务，如果需要同时运行多个任务，可以考虑增加子任务的概念，在一次任务中运行多个子任务。

命令行 input argparse 并不是必须的，这里更加推荐使用 Yaml 文件来管理参数，参数包括项目设置、模型参数等。这样方便我们给每一个任务设定一个参数 Yaml 文件，也能够在每一次 checkpoint 的时候记录一个断点 Yaml 配置，可以用于下一次断点训练或者 finetune。

参数文件里至少要包含:

- `VERSION`： 用于记录当前的核心版本，例如 `0.1.0`。
- `DEBUG`： 是否开启 debug 模式，即使用小型数据集。
- `RESUME`： 从哪个 checkpoint 开始断点训练。
- `TRAIN`： 是否是训练模式。
- `TEST`： 是否是测试模式。
- `MODEL_PATH`: 加载的模型文件路径, 指定是外层目录、当前目录或者是 checkpoint model。
- `RANDOM_SEED`: 全局的随机数种子，使得任务更具可复现性。

其他可以包含的参数可以是训练批次、学习率等超参数。

八、 调试模式

通过加入调试模式，即使用一个小数据集，进行代码逻辑的检查。

九、 断点处理

通过捕获错误避免 Python 的错误中断。

首先通过调试模式跑一遍代码。

通过 checkpoint 保留训练数据，并保存一份用于断点恢复的 Yaml 文件。

十、 进度管理

使用 tqdm、tensorboard 与 wandb 等工具进行可视化。

可以参考 PyTorch 可视化教程[1]或知乎文章[2]。

十一、 结果管理

11.1 日志管理

参考文章[3]，关于 log 日志，我们首先确认一下我们的需求：

1. 同时在终端和日志文件中进行写入 log 的能力；
2. 能在终端与 tqdm 进度条一起使用且不发生冲突，并且日志文件不输出 tqdm 的进度条；
3. 能够自定义 log 文档的格式，能够在必要的地方插入时间信息；
    1. 方便通过时间定位到训练内容等日志信息；
    2. 方便进行性能测量；
4. 对日志信息进行级别的控制。

因此，我们可以用 Python 的基础库：logging[4]。

```
import logging
from time import sleep
from tqdm import tqdm

class TqdmLoggingHandler(logging.Handler):
    def __init__(self, level=logging.NOTSET):
```

[1]https://datawhalechina.github.io/thorough-pytorch/第七章/
[2]https://zhuanlan.zhihu.com/p/484289017
[3]https://zhuanlan.zhihu.com/p/39849027
[4]https://docs.python.org/zh-cn/3/library/logging.html

```
        super().__init__(level)

    def emit(self, record):
        try:
            msg = self.format(record)
            tqdm.write(msg)
            self.flush()
        except Exception:
            self.handleError(record)

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

# 配置日志格式
formatter = logging.Formatter(
  fmt="%(asctime)s.%(msecs)3d %(levelname)-8s \
[%(filename)s %(funcName)s: %(lineno)s] %(message)s",
  datefmt="%Y-%m-%d %H:%M:%S"
)

# 添加适配 tqdm 的 stream 处理器
stream_handler = TqdmLoggingHandler()
stream_handler.setLevel(logging.DEBUG)
stream_handler.setFormatter(formatter)
logger.addHandler(stream_handler)
# 添加日志文件处理器
logfile_handler = logging.FileHandler("task.log", mode='a')
logfile_handler.setLevel(logging.INFO)
logfile_handler.setFormatter(formatter)
logger.addHandler(logfile_handler)

for i in tqdm(range(5)):
  # 五种级别
  logging.critical(f"CRITICAL {i}")
  logging.error(f"ERROR {i}")
  logging.warning(f"WARNING {i}")
  logging.info(f"INFO {i}")
  logging.debug(f"DEBUG {i}")
  sleep(0.5)
```

会输出这样的日志:

```
2023-11-22 20:00:06.947 CRITICAL [test.py <module>: 40] CRITICAL 0
2023-11-22 20:00:06.948 ERROR    [test.py <module>: 41] ERROR 0
2023-11-22 20:00:06.948 WARNING  [test.py <module>: 42] WARNING 0
2023-11-22 20:00:06.948 INFO     [test.py <module>: 43] INFO 0
2023-11-22 20:00:06.949 DEBUG    [test.py <module>: 44] DEBUG 0
```

并且会以追加模式保存到 `task.log` 文件中。

这里推荐在 VS Code 安装 Log File Highlighter 和 Filter Lines 插件用于日志文件分析。前者提供了后缀为 `.log` 的日志文件高亮；后者提供了通过正则表达式或字符串按行筛选日志文件的功能。

11.2 输出数据

对于需要在论文或报告中使用的输出数据，我们不应该只生成具体的图表，而是也记录图表的原始数据，需要用到图表时再统一生成。

由于每个任务都有一个专属的目录，因此可以放心地生成对应的数据文件，不用担心被覆盖。

十二、 性能测量

我们需要对代码进行性能测量，便于后续找出代码瓶颈，进行代码优化。

简单的分析，可以通过分析日志文件的输出时间来计算运行用时，这种做法不用嵌入过于复杂的性能测量逻辑。

如果需要更精确的分析，可以考虑使用 cProfile 或者 timeit，其中 cProfile 较为便捷，而 timeit 可以用于分析小代码片段的耗时。

十三、 图表管理

通过原始数据最后生成图表，而不是仅在代码运行过程中使用 plt 生成图表。

当然，我们有时候也需要可视化进度过程，一方面可以考虑使用 tensorboard 这样的可视化平台；另一方面，只要做好其他模块和可视化模块的分析，就可以很好地兼顾运行时生成图表与运行完毕后生成图表两种情况。

图表的可视化也可以考虑使用 Typst 的 `read()` 函数来管理，进而方便地在报告和论文里将图表可视化出来。

这里我们应该区分「任务图表」和「汇总图表」的区别。任务图表是每次任务生成的图表数据，它们不应该被直接在报告中使用，而是应该经过一个自动化处理脚本，从「任务图表」生成「汇总」图表，再最后在报告里读取并生成最终图表。

十四、 文档管理

使用 Typst 管理文档，包括但不限于笔记、报告、论文和幻灯片。Typst 是一门新兴的科学写作标记语言，有着类似 Markdown 一般的简洁语法和编译速度，以及类似 LaTeX 一般的强大排版能力，更有着这两者都没有的现代脚本语言的可编程性。这篇文档就是使用 Typst 写的。