Lecture 3: Collect Reconstructions: From Procedural Programming to OOP

Xiao-Xin Li

Zhejiang University of Technology

Revision: *2023/03/19*

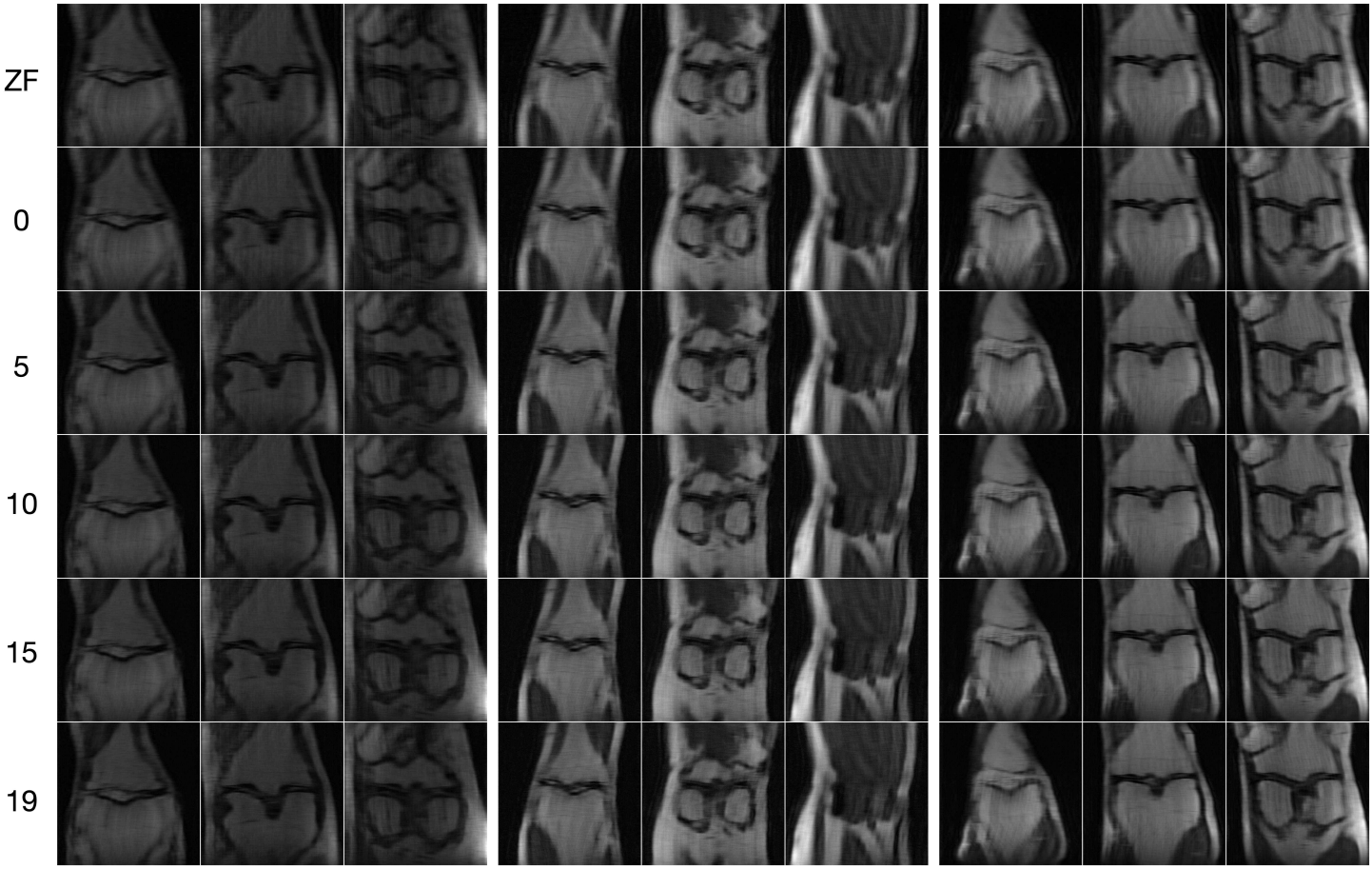
---

1. 主要问题

我们的系统非常复杂，从哪里开始呢？正如我们在 Lecture 2: Course Project Introduction 的最后一节所介绍的，我们的系统是通过多次迭代（多个Epoch，每个Epoch多个Batch）优化网络模型的，从训练过程所产生的Loss曲线，从验证过程所产生的 PSNR/SSIM 曲线，我们看到网络模型的确是在逐渐“优化”了的，那么，经过迭代，对欠采样图像的重建是否真的变得越来越好了呢？我们需要“看到”才能确信这一点。具体的，我们需要看到在迭代过程中所重构的MRI图像的演进过程。

然而，我们有多个 Epoch，多个 MRI 数据，每个 MRI 数据又有多个“切片”。如果要呈现所有 Epoch 下所有 MRI 数据的重构结果，显然是不现实的。我们只能在迭代过程中收集一些我们感兴趣的数据的部分切片在部分 Epoch 上的重建结果。

2. 重构过程的可视化展示



使用代码收集训练过程中所关心的那些重构结果：Collect Reconstructions

3. 程序设计

3.1 在哪里添加代码?

在哪里添加代码，这是一项重要的技能。在面对复杂系统时，我们需要迅速定位问题，这有赖于我们对这个项目代码的“全局观”，有赖于我们对问题和问题的解决思路的理解，而与我们学了多少程序语言、语法和写过多少代码关系并不大。

让我们再来审视下主要问题：搜集**训练过程**中所产生的**中间重构结果**，而我们知道，网络是一边训练，一边验证的。审视代码，我们会发现：

1. 训练器（Trainer）会通过测试器（Tester）进行模型验证。但他们之间不是直接调用关系，而是通过他们的上级代码进行消息传递的：

   `TrainAction.__call__` → `trainer(epoch)`
   → `TestAction.__call__`

```
class TrainAction(UserAction):
    def __call__(self):
        print('===> Build trainer ...\n')
        trainer = Trainer(
            *self['focus_str, model, train_dataloaders, max_epoch, loss, opti
```

```
6
7           print('===> Build validators ...\n')
8           validators = TestAction(self.args, run_mode='val')
9
10          """ 之前的模型已经经过了 last_epoch 轮训练，因此，从 last_epoch+1 开始，继
11          for epoch in range(self['last_epoch'] + 1, self['max_epoch']):
12              trainer(epoch)
13              validators(epoch, trainer.model)
```

2. 再来看 `TestAction.__call__`，其主要是通过 `val_model()` 方法进行模型验证的。但其中的 `tester(epoch, model)` 并没有返回任何重构结果。

```
1  # TestAction
2  class TestAction(UserAction, ReportResults):
3      def val_model(self, epoch, model):
4          print('Running in val mode ...')
5          best_epochs = set()
6          for tester in self._tester_pool:
7              tester(epoch, model)
8              best_epochs.add(tester.best_epoch)
9
10         print()
11         self._delete_redundant_checkpoints(best_epochs)
12         self._best_epochs = best_epochs
13         self._save_best_checkpoint(epoch, model)
14         return
15
16     def test_model(self):
17         pass
18
19     def __call__(self, epoch=None, model=None):
20         if self.in_test_mode:
21             self.test_model()
22             return
23         self.val_model(epoch, model)
```

3. 进入 `Tester.__call__` 查看重构过程。其关键代码在 `predict(self, epoch, model)` 中，其中有一行：`y = model(ref_image, query_image, dc)`，这个 `y` 就是重构结果了。这是否意味着我们可以直接在这一行代码的下方添加收集重构结果的代码？答案是否定的。因为收集重构结果的代码非常复杂，将如此复杂的代码嵌入到重构过程中，会造成重构代码的复杂性，并降低其可读性。因此，需要写专门的代码进行处理。

```
class Tester(Runner):
    def predict(self, epoch, model):
        with torch.no_grad():
            model.eval()
            for batch in self.dataloader:
                dc, ref_image, query_image, query_label, query_max_value, que
                    self._batch_parser(batch)

                y = model(ref_image, query_image, dc)

                y = y.abs()
                self._cr(epoch, query_files, query_image, y, dc.unnorm)
                # 我们需要在此处追加收集重构结果的代码

                label = query_label
                max_val = query_max_value

                self._vp(y, label, max_val, query_files)

        return self._vp.mean_psnr_ssim

    def __call__(self, epoch, model):
        self._vp.clear_all()
        psnr, ssim = self.predict_zf() if model is None else self.predict(epc

        if ssim > self.max_ssim:
            self.best_epoch = epoch
            self.max_ssim = ssim
            self.max_psnr = psnr

        self.log(f'{self.dataset}: PSNR/Epoch', psnr, epoch)
        self.log(f'{self.dataset}: SSIM/Epoch', ssim, epoch)
        self.print_formatted_psnr_ssim(epoch, psnr, ssim)
```

3.2 面向过程的程序设计

3.2.1 接口设计

1. 简化框架代码：

```
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
        recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
```

2. 了解相关背景知识、术语和数据结构：
    - 背景知识：整个训练过程需要迭代多个周期（epochs），而每个周期（epoch）需要迭代多个批次（batch），这是一个双重循环。
    - **Query.** query 本身是“疑问、询问”之意，在计算机视觉领域中，常用 query 来表示待处理的目标图像。例如，对于人脸识别系统，query 表示“待识别的人脸图像”；对于我们的 MRI 重构系统，query 表示“待重建的 MRI 图像”。
    - `queries` 和 `recons` 的数据结构：`B × C × H × W`。
3. 编写初步接口

```python
def collect_reconstructions(
    req_epochs: list,
    req_indices: list,
    epoch: int,
    files: list,
    queries: torch.Tensor,  # B × C × H × W
    recons: torch.Tensor  # B × C × H × W
    ):
    """Collects reconstructed MRI slices
    Args:
        req_epochs: request epochs, i.e., the epochs users concern
        req_indices: request indices, i.e., the slice indices users concern
        epoch: current epoch
        files: query files used to produce under-sampled MRI images (query images) in the k-th batch
        queries: query images or under-sampled images input to CNN in the k-th batch
        recons: reconstruction images output by CNN in the k-th batch
    """
    pass
```

    a. 接口参数的类型声明：是必要的吗？
    b. 接口注释（详见这篇博文）
        i. 使用两个“"""”；
        ii. 使用“Args:”、“Returns:”、“Raises:”等关键字，注意，这里的动词用的都是单数第三人称；
    c. 接口的函数体：`pass`

3.2.2 编写粗糙的代码，梳理主要逻辑，进一步修正接口

```python
def collect_reconstructions(req_epochs, req_indices, epoch, files, queries, reco
    req_recons = []  # Python 中的 list，一个重要的数据结构
    if epoch in req_epochs:
        for recon_slice in recons:
            # 注意，此时还须检查 recon_slice 是否是在 req_indices 中，
            #  但我们先简化处理，集中在更重要的逻辑上
            req_recons += recons
    return req_recons
```

如果把上面的代码放到整个框架中，立即会发现其中有致命的逻辑错误：`req_recons` 只能存储一个 `batch` 的重构结果。

```python
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
        recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
        # req_recons 每次迭代都会更新！！
        req_recons = collect_reconstructions(req_epochs, req_indices, epoch, fil
```

怎么解决这个问题呢？我们有如下几种方案：

1. 在框架代码中维护 `req_recons`，如下所示。但这样做的主要问题是框架代码需要知道更多的关于 `collect_reconstructions()` 所返回的 `req_recons` 的数据结构，这样，框架代码与被调用函数的“耦合”成都就更高了，一旦被调用函数对其所返回的数据的存储结构做了调整，框架代码就必须随之更新了，这样框架代码就会变得越来越复杂了，其功能也越来越不“单纯”了。

   ```python
   req_recons = []  # 初始化一个 list
   for epoch in range(max_epoch):  # 进行多轮训练
       for files, queries in dataloader:  # 加载一个 batch 的数据
           recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
           # 每次迭代，都将搜集到的结果追加到 req_recons 中
           req_recons += collect_reconstructions(req_epochs, req_indices, epoch,
   ```

2. 在 `collect_reconstructions()` 中维护 `req_recons`：将 `req_recons` 作为一个全局变量。但是这样做的主要问题是：`req_recons` 非常不安全！谁都可以修改它、破坏它。

   ```python
   req_recons = []  # 将 req_recons 作为一个全局变量
   def collect_reconstructions(req_epochs, req_indices, epoch, files, queries, r
       ~~req_recons = []~~
       if epoch in req_epochs:
   ```

```
        for recon_slice in recons:
            req_recons += recons
    return req_recons
```

3. 一种较好的方案：`req_recons` 传入-传出，具体如下。但这种方案仍然存在很大的安全隐患。实际上，不管设计者的意愿如何，对 `req_recons` 的维护实际上是不单一的，它至少需要框架代码的配合，如果框架代码不慎修改了 `req_recons` 呢?

```
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
        recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
        # req_recons 每次迭代都会更新！！
        req_recons = collect_reconstructions(req_recons, req_epochs, req_indi
```

```

def collect_reconstructions(req_recons, req_epochs, req_indices, epoch,
files, queries, recons):
    if epoch in req_epochs:
        for recon_slice in recons:
            req_recons += recons
    return req_recons
```

3.2.3 审视新接口：冗长的参数列表、重复的参数传递、潜在的安全隐患

```
def collect_reconstructions(
    req_recons: list,
    req_epochs: list,
    req_indices: list,
    epoch: int,
    files: list,
    queries: torch.Tensor,  # B × C × H × W
    recons: torch.Tensor  # B × C × H × W
    ):
    """Collects reconstructed MRI slices
    Args:
        req_recons: reconstructions of the request slices on request files and
request epochs
        req_epochs: request epochs, i.e., the epochs users concern
        req_indices: request indices, i.e., the slice indices users concern
        epoch: current epoch
```

```
        files: query files used to produce under-sampled MRI images (query images) in the k-th batch
        queries: query images or under-sampled images input to CNN in the k-th batch
        recons: reconstruction images output by CNN in the k-th batch
    """
    pass
```

现在来看下新接口存在的问题。比较明显的问题有3个：

1. 冗长的参数列表，而且，这个参数列表可能随着需求的进一步提升而变得更加复杂；
2. 重复的参数传递：用户所关注的`epochs`（`req_epochs`）和切片索引（`req_indices`），是固定的，每次调用，都需要重复传递这两个参数；
3. `req_recons`的设计非常蹩脚，需要不断传入、传出，存在安全隐患。

3.2.4 使用`lambda`表达式，避免传递重复参数

```
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
        recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
        req_recons = collect_reconstructions(req_recons, req_epochs, req_indices
```

`lambda`表达式，即：匿名函数，在主流的编程语言（如Java、Matlab等）中都有使用，常常和一些需要使用函数对象的功能函数搭配使用，如`reduce` / `filter` / `map` / `sorted`。这些函数具有大致相同的调用格式：

```
# reduce 函数在 Python 3 中被移出了全局函数，需要从functools模块中导入，
#   所以，reduce 的语法颜色与其它函数不相同
reduce(function, iterable)
filter(function, iterable)
map(function, iterable)
sorted(iterable, cmp[key])
```

`lambda`表达式仅仅是创建函数的一个特殊方法，它只包含一条语句，并自动返回这条语句的结果：

```
fun = lambda x : op(x)
```

在这里，我们可以用`lambda`表达式简化参数传递：

```
collect_recons = lambda req_recons, epoch, files, queries, recons:
                     collect_reconstructions(req_recons, req_epochs, req_indices
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
        recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
        req_recons = collect_recons(req_recons, req_epochs, req_indices, epoch,
```

但，你感觉到了吗？我们的代码仍然显得那么啰嗦、蹩脚，不像是一个高手所为。而这，当然也不是`lambda`表达式的正宗用法，PyCharm等开发环境会提示你写一个新的函数来代替`lambda`表达式。

3.2.5 不是我们的方法错了，是努力的方向错了

问题层出不穷，这似乎在暗示我们：再怎么努力设计接口，也无法做到尽善尽美。这说明，不是我们的方法错了，而是我们努力的方向错了。事实上，这些问题都可以用面向对象的程序设计（OOP）思想来完美、轻松地解决！

3.3 OOP

3.3.1 从“接口”抽象出“类”

```
def collect_reconstructions(
    req_recons: list,
    req_epochs: list,
    req_indices: list,
    epoch: int,
    files: list,
    queries: torch.Tensor,  # B × C × H × W
    recons: torch.Tensor  # B × C × H × W
    ):
    indices = get_indices(recons)
    if epoch in req_epochs:
        for recon_slice in recons:
            req_recons += recons
    return req_recons
```

从“接口”抽象出“类”。主要方法：

1. 使用接口的名字来命名一个新的类：`collect_reconstructions` → `CollectReconstructions`，注意：
   a. 类的命名规范：`Class names should use CamelCase convention`.
   b. 类名一般多使用“名词”，而方法名一般多用动词和助动词；但如果你囿于这个习惯性思维，你就很难写走出面向过程的编程思维了，你需要告诉自己，类的名字也可以是一个“动词”。
2. 将重复传递的参数放入类的“构造函数”或“初始化函数”：`__init__(self, ...)`，我们的第一个魔法函数；

```
class CollectReconstructions:
    # 有没有觉得 Python 的构造函数的命名方式很奇怪？写的很啰嗦？
    # 实际上，__init__ 是一个魔法函数（Magic Function）
    # 凡是魔法函数，都是无须明确调用的，它们都有一种潜在的调用方式，我们渐渐就会明白。
    def __init__(self, req_epochs, req_indices):
        self.req_epochs = req_epochs
        self.req_indices = req_indices
```

3. 将需要维护的主要数据也放入类的“构造函数”

```
class CollectReconstructions:
    def __init__(self, req_epochs, req_indices):
        self.req_recons = []
        self.req_epochs = req_epochs
        self.req_indices = req_indices
```

4. 改造接口的实现

```
class CollectReconstructions:
    def __init__(self, req_epochs, req_indices):
        self.req_recons = []
        self.req_epochs = req_epochs
```

```
        self.req_indices = req_indices
        return

    def collect_reconstructions(self, epoch: int, files: list,
                queries: torch.Tensor, recons: torch.Tensor):
        if epoch in self.req_epochs:
            for recon_slice in recons:
                self.req_recons += recons
        return req_recons
```

5. 现在，我们来看看在框架代码中如何使用这个类。是不是已经精简很多了？框架代码几乎恢复到了最初的模样。但是，还有一点蹩脚的地方：类名与类所提供的主要方法的名字是不是太像了？调用起来是不是也太哆嗦了？

```
collect_recons = CollectReconstructions(req_epochs, req_indices)
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
        recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
        collect_recons.collect_reconstructions(epoch, files, queries, recons)
```

6. 用**魔法函数** `__call__()` 重定义类的主方法：

```
class CollectReconstructions:
    def __init__(self, req_epochs, req_indices):
        self.req_recons = []
        self.req_epochs = req_epochs
        self.req_indices = req_indices

    def __call__(self, epoch: int, files: list,
                queries: torch.Tensor, recons: torch.Tensor):
        indices = get_indices(recons)
        if epoch in self.req_epochs:
            for recon_slice in recons:
                self.req_recons += recons
        return
```

7. 现在，再来看主程序如何使用 `CollectReconstructions`：

```
collect_recons = CollectReconstructions(req_epochs, req_indices)
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
```

```
4           recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
5           collect_recons.collect_reconstructions(epoch, files, queries, recons)
```

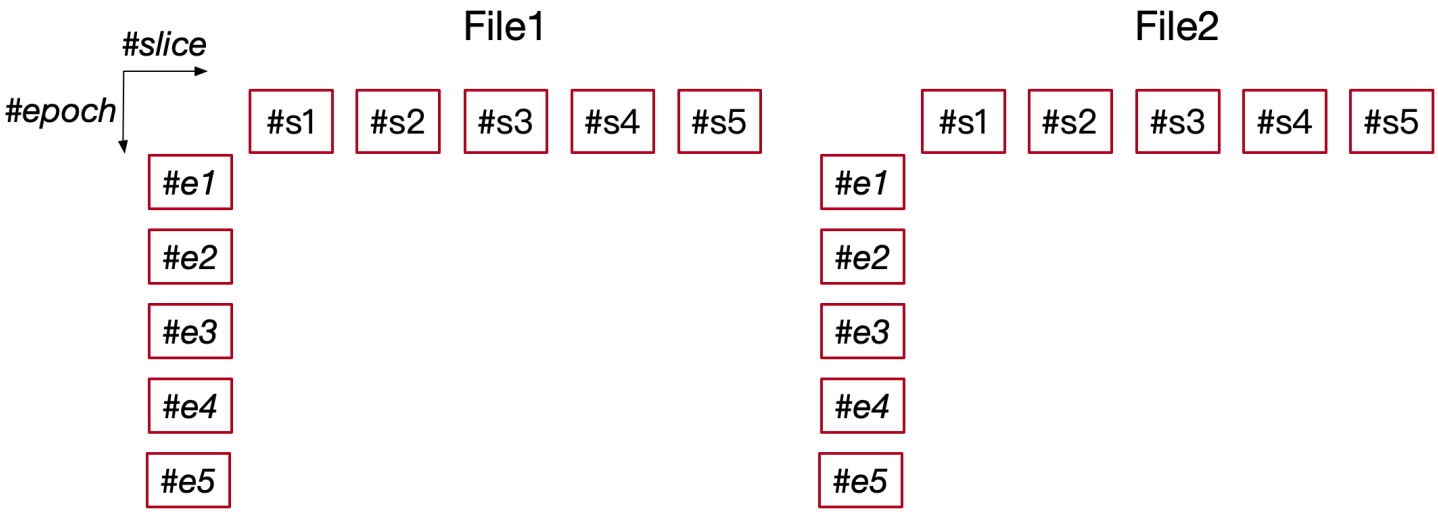
3.4 类的主要方法的设计

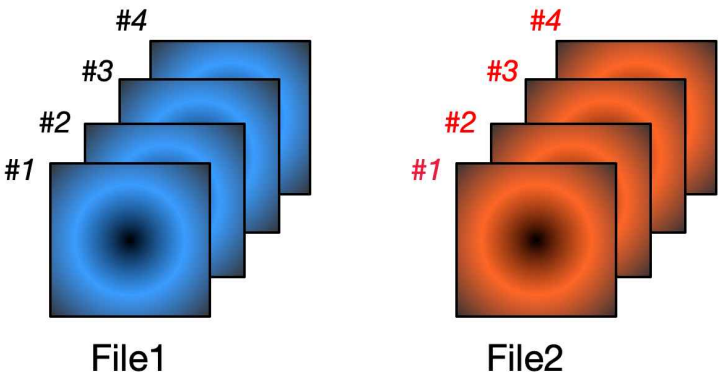
在设计了类的成员和主要接口之后，现在，我们需要结合着数据和我们的需求，全盘考虑下类的主要接口具体该如何实现。

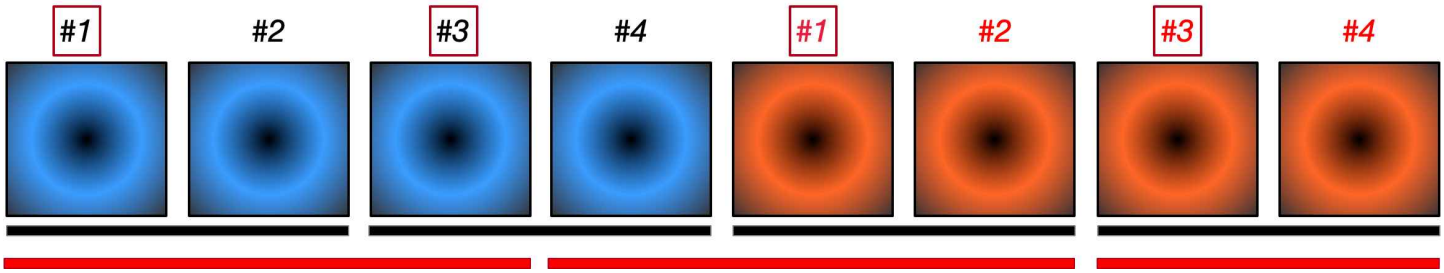
我们希望最终所展示的可视化结果正如我们在第2节所看到的那样：每一列展示的是某个File的某个slice，而每一行展示的是某个epoch上的重构结果，不同的File之间以一定的间距隔开。

值得注意的是，最初，我们是没有第2节中所展示的图的，这个图只存在于我们的想象中，也就是，我们希望能以那样的方式对重构结果进行可视化展示。

有了期望的结果，我们需要考虑“数据结构”，也就是，以什么样的数据结构来存储这样的阵列数据呢？








```
collect_recons = CollectReconstructions(req_epochs, req_indices)
for epoch in range(max_epoch):  # 进行多轮训练
    for files, queries in dataloader:  # 加载一个 batch 的数据
        recons = cnn_model(queries)  # 调用 CNN 模型，进行重构
        collect_recons(epoch, files, queries, recons)
```

```
class CollectReconstructions:
    def __call__(self, epoch: int, files: list,
                 queries: torch.Tensor, recons: torch.Tensor):
        if epoch in self.req_epochs:
            for recon_slice in recons:
                # 我们不是要缓存来自于 recons 的所有 slice 的
                #  还需要判断当前的 recon_slice 是否属于 self.req_indices
                self.req_recons += recons
        return
```

```
def collect_reconstructions(req_epochs, req_indices, epoch, files, queries, reco
    req_recons = []
    indices = get_indices(recons)
    if epoch in req_epochs:
        for idx, recon_slice in zip(indices, recons):
```

```
            if idx in req_indices:
                req_recons += recons
    return req_recons
```