



🚀 BasicSR: 图像、视频超分复原增强开源库
中文解读文档

July 15, 2022



<https://github.com/XPixelGroup/BasicSR>

Notes

本文档的最新版可以从 <https://github.com/XPixelGroup/BasicSR-docs/releases> 下载
欢迎大家一起来帮助查找文档中的错误，完善文档

👉点击快速跳转到文档主目录

Authors

感谢为 BasicSR 代码与文档贡献的每一位朋友

下面仅列出了编撰此文档初稿并持续维护的作者们

- 王鑫涛 (Xintao Wang)  xintao.alpha@gmail.com  <https://xinntao.github.io/>
- 谢良彬 (Liangbin Xie)   <https://liangbinxie.github.io/>
- 陈翔宇 (Xiangyu Chen)  chxy95@gmail.com  <https://chxy95.github.io/>
- 张文龙 (Wenlong Zhang)  wenlong.zhang@connect.polyu.hk  <https://wenlongzhang0517.github.io/>
- 刘翼豪 (Yihao Liu)   Google Scholar
- 孔祥涛 (Xiangtao Kong)  kxtv587@gmail.com  <https://xiangtaokong.github.io/>
- 顾津锦 (Jinjin Gu)  hellojasongt@gmail.com  <https://www.jasongt.com/>
- 何静雯 (Jingwen He)   Google Scholar
- 董超 (Chao Dong)  chao.dong@siat.ac.cn  Google Scholar

XPixel Metaverse



图 0.1: XPixel 的元宇宙。你可以在[官网](https://xpixel.group/)查看更高清的版本。

XPixel Metaverse 是 XPixel Group (官网: <https://xpixel.group/>) 集体智慧的结晶，它体现了 XPixel 所有成员对科研的坚持，对艺术的追求，对生活的热爱，和对世界的责任。下面就请跟随文字向导，一起来巡游 XPixel 的历史和现在。

设计理念

为了完整而有特色地展示 XPixel Group 的各项科研成果，我们采用了“地图”的形式，通过山川、河流、建筑等元素，自然地将一项项科研成果与之对应，最终构成了 XPixel Metaverse。目前地图的大陆被三片海洋包围，中间的大陆则遍布 XPixel 从2014年到2022年来经典的科研成果，完美地将科技和艺术结合在了一起。

海洋

海洋以 XPixel 的组内文化: Love (奉献) 、Focus (专注) 、Balance (平衡) 命名，象征着大陆上众多科研成果是在这种优秀的文化中孕育而成。而大陆的河流最终流入这三片汪洋之中，又象征着小组的工作成果最后又回归并加强了这种科研文化。

大陆

贯彻大陆的河流由左上角的高山发源，以此山代表深度学习超分辨率的开山之作 SRCNN，象征着这项工作重若泰山的源头意义。由此发源的河流上段命名为 Image Super Resolution (SR)，此段周围的建筑与地形均是传统单图像超分的重要成果。同时在上段出现了最早的支流 Blind，象征着图像盲超分这一分支领域的出现。

顺主流而下到达一处大坝，大陆地形由高地形转为海拔更低更平坦的地形，象征着 XPixel 在早期工作的基础上，随着更多优秀同学的加入，科研工作的开展比以往更加顺利。在大坝下方形成的名为 Low-Level Vision 的湖泊分出三条河流，分别是：

- 象征交互式可调节复原的：Interactive Modulation

- 象征超分网络可解释性的: Interpretation
- 象征视频处理和复原的: Video Processing

Interpretation 所流经的地图上方，其沙漠地貌与其他地方产生了强烈反差。这是因为底层视觉可解释性领域的工作非常稀少，且进展尤为不易，所以地貌较其他领域更加严酷。而这份不易对应的，是隐藏在沙漠背后未被探索的广袤空间。

在大陆的沿海处有一片码头区域，是超分领域最大的开源代码库 BasicSR。这是 XPixel 科研工作的重要载体和精华所在，同时也在国际上获得了广泛的使用与认可，与码头的作用高度契合。

文字介绍及其他元素

地图的左上角是作品的名称: XPixel Metaverse。地图正上方是 XPixel 的使命愿景: **Our mission is to make the world look clearer and better!** 地图右上角是 XPixel 的 logo，地图中沙漠和沿海的两处小凤凰是 XPixel 的吉祥物。

大陆上各个地形、建筑均以 XPixel 的科研成果命名，并添加文字介绍。文字介绍最多四行，从上到下：

- 某领域的开创性成果，以斜体标明该新领域
- 工作的简称
- 工作发表的期刊及年份
- 工作的荣誉在最下方特别标明

同时为了能更轻松地辨识各个领域的成果，各领域成果均以地图右上角 XPixel 的 Logo 图案中的一个颜色进行着色。

XPixel Metaverse 承载着我们的历史，也呼唤着美好的未来！愿世界各地优秀的学者与我们一起，让世界变得更清晰，更美好！

Contents

1 概述	8
1.1 本文档说明	8
1.2 BasicSR 介绍	8
1.3 使用方式与场景	9
1.3.1 本地克隆代码	9
1.3.2 basicsr 作为 Python package	9
1.4 单元测试	10
2 安装	11
2.1 环境依赖	11
2.2 BasicSR 安装	12
2.2.1 本地 clone 代码	12
2.2.2 pip 安装	13
2.2.3 验证 BasicSR 是否安装成功	13
2.3 PyTorch C++ 编译算子	13
2.4 常见安装问题	15
3 入门	18
3.1 目录解读	18
3.2 训练流程	20
3.2.1 代码的入口和训练的准备工作	20
3.2.2 Dataset 和 Model 的创建	22
3.2.3 训练过程	28
3.3 测试流程	29
3.4 推理流程	30
4 代码主体结构	31
4.1 整体框架	31
4.2 动态实例化与 REGISTER 注册机制	32
4.2.1 REGISTER 注册机制	32
4.2.2 自动扫描并 import 注册的类/函数	35
4.3 配置(Options)	38
4.3.1 实验命名与 debug 模式	38
4.3.2 配置文件简要说明	39
4.3.3 命令行修改配置	46
4.4 数据 (Data Loader 和 Dataset)	47
4.4.1 basic/data 目录介绍	48

4.4.2	Data loader 和 Dataset 的创建	49
4.4.3	Dataset 示例讲解	49
4.4.4	Dataset prefetch 说明	51
4.5	模型 (Model)	52
4.5.1	basic/models 目录介绍	52
4.5.2	Model 的创建	53
4.5.3	Base Model 和 Model 示例讲解	53
4.5.4	保存模型、训练状态 和 Resume	57
4.5.5	模型 validation	57
4.5.6	EMA 介绍	58
4.6	网络结构 (Architecture)	58
4.6.1	basic/arch 目录介绍	58
4.6.2	网络结构 arch 的创建	59
4.7	损失函数 (Loss)	59
4.7.1	basic/losses 目录介绍	60
4.7.2	loss 的创建	60
4.7.3	loss 在日志中的添加	60
4.8	算子 (Ops)	61
4.8.1	什么是算子?	61
4.8.2	BasicSR 中的自定义算子	62
4.8.3	BasicSR 中算子的编译、安装、使用	62
4.9	日志系统 (Logger)	63
4.9.1	log 文件记录与解读	63
4.9.2	tensorboard logger 记录及解读	65
4.9.3	Wandb 记录及解读	67
5	指标	68
5.1	概述	68
5.2	PSNR	69
5.3	SSIM	71
5.4	NIQE	72
5.5	如何使用指标	72
5.5.1	通过配置文件指定	72
5.5.2	使用脚本计算	73
6	如何添加与修改	74
6.1	添加修改 Dataset	74
6.2	添加修改模型	75
6.3	添加修改网络结构	76
6.4	添加修改损失函数	77
6.5	添加修改指标	78
7	数据准备	80
7.1	常见用法	80
7.2	数据存储格式	81
7.2.1	LMDB 具体说明	81
7.3	meta 文件介绍	83

7.3.1 现有 meta 文件说明	83
7.4 File Client 介绍	83
7.5 常见数据集介绍与准备	83
7.5.1 图像数据集 DIV2K 与 DF2K	83
7.5.2 视频帧数据集 REDS	84
7.5.3 视频帧数据集 Vimeo90K	85
8 部署	86
9 脚本介绍	87
10 BasicSR-examples 模板	88
11 经验	89

第 1 章

概述

本章节对本文档 (第1.1小节)、BasicSR (第1.2小节) 以及使用方式与场景 (第1.3小节) 做一个简略的概述。

§1.1. 本文档说明

本文档旨在完整地介绍 BasicSR 的设计和框架，为入门者提供一份上手指南，为使用者提供一份日常参考。

本文档不涉及具体函数和代码的介绍。如果需要具体函数和代码的介绍，请查阅 BasicSR 的在线 API 文档。我们更推荐读者直接查看代码，这样可以更加完整、细致的了解实现细节。

🔔 BasicSR API 文档

BasicSR 的 API 文档是实时更新的，并且发布在 [readthedocs.io](#) 网站上：

<https://basicsr.readthedocs.io/en/latest/>

国内可能访问速度缓慢，后续我们会考虑导出 PDF 文档作为附录。

本文档也不涉及超分和复原的专业入门。我们先挖一个大坑：后面我们会推出超分复原入门与经典工作解读的文档。

§1.2. BasicSR 介绍

BasicSR 是一个开源项目，旨在提供一个方便易用的图像、视频的超分、复原、增强的工具箱。我们希望它能够：

- 使入门者更快上手；
- 使研究者更方便实验；
- 使更多人更容易使用更先进的算法。

BasicSR 代码库从2018年4月20日开始第一个提交，然后随着做研究、打比赛、发论文，逐渐发展与完善起来。它从最开始的针对超分辨率算法到后来拓展到其他更多复原增强相关的算法，因此，BasicSR 中 SR 的涵义也从 Super-Resolution 延拓到 Super-Restoration。

2022年5月9日，BasicSR 迎来新的里程碑，它加入到 XPixel 大家庭中，和更多的小伙伴们一起致力于把 BasicSR 建设得更好！

BasicSR 是一个开源项目，我们欢迎更多的人一起来维护、建设 :-)

XPixel 团队介绍:

XPixel 团队是一个学术组织，它的愿景是让世界看得更清晰、更美好 (Make the world look clearer and better!)

官网地址: <https://xpixel.group/>

§1.3. 使用方式与场景

BasicSR 主要有两种使用方式:

1.3.1. 本地克隆代码

在这个方式下，我们把整个 BasicSR 的代码都 copy/clone 下来，也就可以方便地查看 BasicSR 的完整代码，修改并使用。

当我们尝试复现、开发方法的时候，我们比较推荐这个方式，因为可以更好地看到代码全貌，方便调试。

它的弱点是：

1. 整个仓库中有很多不需要使用的代码。因为 BasicSR 提供了很多方法的实现，在你自己的实验中，大部分的代码并不需要。但是，当你了解完 BasicSR 的代码框架后，你就放心地知道，它们并不影响。它们基本都是独立存在的
2. 当你 release 你自己新开发的方法（假设叫 NBCNN）的代码时，会遇到麻烦：你必须要 release 包含整个 BasicSR 的代码，而不能专注于 NBCNN 的核心代码。遇到这种情况，我们就可以用下面的使用场景：把 basicsr 当作一个 Python package

本地克隆代码仓库的安装方法

参见章节2.2.1：安装。

1.3.2. basicsr 作为 Python package

BasicSR 也有一个单独的 Python package — basicsr，发布在 [pypi](#) 上。它可以通过 pip 安装，提供了训练框架、流程、BasicSR 中已有的函数和功能。你可以基于 basicsr 方便地搭建你自己的项目。例如，

基于 basicsr 搭建的 [Real-ESRGAN](#);
基于 basicsr 搭建的 [GFPGAN](#)。

它们都使用了 basicsr 里面已有的函数和功能，因此只要专注于新方法的功能即可。

一般来说，深度学习项目都可以分为以下几个部分：

- data: 定义了训练数据，来喂给模型的训练
- arch (architecture): 定义了网络结构和 forward 的步骤

- model: 定义了在训练中必要的组件（比如 loss）和一次完整的训练过程（包括前向传播，反向传播，梯度优化等），还有其他功能，比如 validation 等
- training pipeline: 定义了训练的流程，即把数据 dataloader，模型，validation，保存 checkpoints 等等串联起来

我们开发一个新的方法时，我们往往在改进 data, arch, model 这几块内容。而很多流程、基础的功能其实是共用的。

BasicSR 就把很多相似的功能都独立出来，我们只要关心 data, arch, model 的开发即可。而 basicsr 这个 Python package，则可以进一步将已有函数和功能封装起来，我们只要关心新方法的开发即可。

■ 把 basicsr 当作一个 Python package 的安装方法

参见章节2.2.2: 安装。

■ 如何基于 basicsr Python package 来开发呢？

我们建立了一个模板: BasicSR-examples。

开发方法参见：https://github.com/xinntao/BasicSR-examples/blob/master/README_CN.md。

当然，直接使用 basicsr Python package 也有缺点：它会调用 BasicSR 里面的函数，如果里面的函数不能满足需求，或者有 bug 的情况，我们往往难以修改。（需要进入安装 pip package 的地方进行修改）。

为了应对这种情况，我们一般是在本地克隆仓库的情况下开发，然后等到 release 新方法的时候，再基于 BasicSR-examples 新建一个仓库，使用 basicsr 的 pip package。

§1.4. 单元测试

我们使用单元测试主要保证输入输出 shape 的正确性，以及一些流程的正确性。一般来说，我们较少使用。BasicSR为了完备，将单元测试也加入进来。

在 tests 目录的函数会被单元测试执行。单元测试需要 GPU CUDA 环境。

■ 单元测试命令

```
1 python -m pytest tests/
```

第 2 章

安装

本章节首先介绍安装 BasicSR 所需的环境依赖 (第2.1小节)，随后介绍安装 BasicSR 的两种方式：本地 clone 源代码安装和 pip 安装 basicsr 包 (第2.2小节)。对于需要在项目中使用 PyTorch C++ 编译算子的情况，我们也提供了相应的安装方式 (第4.8.3小节)。最后，我们将安装过程中常见的问题进行了汇总 (第2.4小节)。

§2.1. 环境依赖

由于 BasicSR 是基于 Python 语言和 PyTorch 深度学习框架进行开发的，因此在安装 BasicSR 之前，需要在电脑或者服务器上安装 Python 环境以及各种相关的 Python 库；如果想要在 GPU 上运行程序的话，也需要先在电脑上配置相应的 CUDA 环境。以下我们分别对 CUDA 和相应的 Python 库进行简要说明。

1. NVIDIA GPU + CUDA: GPU (Graphics Processing Unit) 由于其高效的并行能力，目前被广泛用于深度学习的计算中；CUDA (Compute Unified Device Architecture) 是 NVIDIA 推出的可以让 GPU 解决复杂计算问题的运算平台。如果需要训练 BasicSR 中的模型，需要使用 GPU 并配置好相应的 CUDA 环境
2. Python 和 Python 库 (对于 Python 库，我们提供了相应的安装脚本):
 - a) Python ≥ 3.7 (推荐使用 Anaconda 或者 Miniconda)
 - b) PyTorch ≥ 1.7 : 目前深度学习领域广泛使用的深度学习框架

当配置好 Python 环境和 CUDA 环境之后，可以直接运行以下的脚本一次性安装 BasicSR 中调用的各种 Python 库：

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

Windows 环境

BasicSR 也支持 Windows 环境。

更多注意事项，参见第2.4小节 Q1 问题。

§2.2. BasicSR 安装

在安装好上述的环境依赖后，此时就可以进行 BasicSR 的安装了。

本小节的安装默认不适用 PyTorch C++ 编译算子，若需要，则参考第 4.8.3 小节进行安装。

🔔 BasicSR 安装方式

根据不同的需求，我们提供了两种安装 BasicSR 方式，**两种方式只能选择一种安装，否则容易产生冲突**。

- 如果希望查看 **BasicSR** 中的细节或者需要对其进行修改，推荐通过本地 clone 代码的方式进行安装
- 如果仅仅是将 BasicSR 作为一个 **Python** 包进行使用（比如项目 **GFGAN** 和 **Real-ESRGAN**），推荐直接从 PyPI 安装 BasicSR，这样可以使得自身项目的代码结构更加简洁

2.2.1. 本地 clone 代码

要通过本地 clone 安装 BasicSR，需要在终端上依次进行以下3个步骤。

1. 克隆项目：

```
git clone https://github.com/XPixelGroup/BasicSR.git
```

2. 安装依赖包：

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

3. 在 BasicSR 的根目录下安装 BasicSR：

```
python setup.py develop
```

如果希望安装的时候指定 CUDA 路径，可使用如下指令：

```
CUDA_HOME=/usr/local/cuda \  
CUDNN_INCLUDE_DIR=/usr/local/cuda \  
CUDNN_LIB_DIR=/usr/local/cuda \  
python setup.py develop
```

2.2.2. pip 安装

对于使用 pip 安装 BasicSR，在终端上运行以下指令即可：

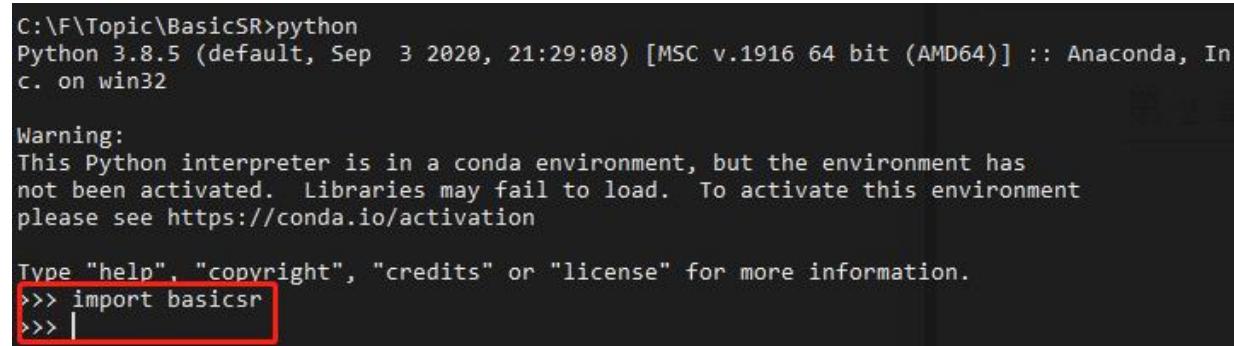
```
pip install basicsr
```

如果希望安装的时候指定 CUDA 路径，可使用如下指令：

```
CUDA_HOME=/usr/local/cuda \
CUDNN_INCLUDE_DIR=/usr/local/cuda \
CUDNN_LIB_DIR=/usr/local/cuda \
pip install basicsr
```

2.2.3. 验证 BasicSR 是否安装成功

当选择了上述两种方式中的一种方式安装 BasicSR 后，我们可以通过图2.1的方式来判断是否成功安装 BasicSR：



The screenshot shows a terminal window with the following text:

```
C:\F\Topic\BasicSR>python
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] :: Anaconda, In
c. on win32

Warning:
This Python interpreter is in a conda environment, but the environment has
not been activated. Libraries may fail to load. To activate this environment
please see https://conda.io/activation

Type "help", "copyright", "credits" or "license" for more information.

>>> import basicsr
>>> |
```

图 2.1: 验证成功安装 BasicSR

如果此时没有报错，则说明 BasicSR 安装成功，此时便可以基于 BasicSR 进行开发啦 ~~~~

§2.3. PyTorch C++ 编译算子

考虑到某些项目中会需要使用 PyTorch C++ 编译算子，我们在这个小节针对这种情况也提供了相应的 BasicSR 安装方式。如果不需要使用相关 C++ 编译算子，则此小节可以跳过。

对于项目中需要使用以下的 PyTorch C++ 编译算子时，比如：

- 可变性卷积 DCN (如果安装的 Torchvision 版本 $\geq 0.9.0$ ，会自动使用 TorchVision 中提供的 DCN，故不需要安装此编译算子)，比如：EDVR 中的 DCN
- StyleGAN 中的特定的算子，比如：upfirdn2d, fused_act

由于第2.2小节所提到的安装方式不支持 PyTorch C++ 编译算子，为了能够使用 PyTorch C++ 编译算子，此时需要一些特定的修改 (有以下两种方式可供选择)：

1. 安装的时候对 PyTorch C++ 编译算子进行编译: 此时需要将原先的安装指令进行修改, 其中 `BASICSR_EXT=True` 中的 `EXT` 是单词 Extension 的缩写。

a) 对于通过本地 clone 代码安装 BasicSR 的方式, 此时修改指令:

```
python setup.py develop --> BASICSR_EXT=True python setup.py develop
```

b) 对于通过 pip 安装 BasicSR 的方式, 此时修改指令:

```
pip install basicsr --> BASICSR_EXT=True pip install basicsr
```

进行了上述的修改之后, 如果我们需要运行 StyleGAN 的测试代码 (需要用到 PyTorch C++ 编译算子) (代码位于 `inference/inference_stylegan2.py`), 此时直接输入指令即可:

```
python inference/inference_stylegan2.py
```

2. 每次在跑程序的时候即时加载 (JIT) PyTorch C++ 编译算子: 如果我们选择了这种方式, 此时不需要修改 BasicSR 的安装指令。依然拿 StyleGAN 的测试代码举例, 在这种情况下, 如果想要运行 StyleGAN 的测试代码, 此时需要输入的指令是:

```
BASICSR_JIT=True python inference/inference_stylegan2.py
```

关于上述提到的两种使用 PyTorch C++ 编译算子方式之间的优劣和场景对比如表2.1所示:

选项	优点	缺点	适用场景	具体安装指令
安装编译 C++ 算子	运行代码的时候, 能够快速加载编译算子	配置环境的时候, 需要更多的依赖, 碰到的问题可能更多	需要多次训练或多次测试模型	在安装的时候, 设置 <code>BASICSR_EXT=True</code>
即时加载 C++ 算子	有着更少的依赖, 碰到的问题可能更少	每次运行代码的时候, 都需要花费几分钟重新编译算子	仅仅是进行测试	在跑程序的时候, 设置 <code>BASICSR_JIT=True</code>

表 2.1: 安装编译算子和即时加载编译算子的对比。

■ 注意

- 对于需要在安装的时候就编译 PyTorch C++ 算子, 需要确保: gcc 和 g++ 版本 ≥ 5 。
- `BasicSR_JIT` 有最高的优先级。即使在安装的时候已经成功编译了 C++ 编译算子, 若在运行代码指令中设置了 `BasicSR_JIT=True`, 此时代码仍旧会即时加载 C++ 编译算子。
- 在安装的时候, 不能设置 `BasicSR_JIT=True`。

§2.4. 常见安装问题

1. Q1: Windows 下是否可以使用?

经过验证, Windows 下可以通过上述的两种安装方式安装 BasicSR。如果需要使用 CUDA, 需要指定 CUDA 路径。另外需要注意的是如果需要在 Windows 环境中使用环境变量, 需要使用以下方式:

```
set BASICSR_EXT=True
```

由于 BasicSR 项目是在 Linux (Ubuntu) 环境下进行开发的, 因此推荐在 Linux 环境下基于 BasicSR 进行项目的开发。

2. Q2: BASICSR_EXT 和 BASICSR_JIT 在什么环境下才能执行?

如果在加入 BASICSR_EXT 和 BASICSR_JIT 环境变量之后运行报错, 此时需要检查 gcc 版本。BasicSR 在已被验证在 gcc5 ~ gcc7 版本下可以成功编译 C++ 编译算子。

3. Q3: 安装路径混淆的问题

很多问题都是由于安装路径混淆, 其主要原因是本地 clone 代码和 pip 安装包两个方式被同时执行。

具体而言, 如果先通过 pip 安装了 BasicSR, 随后又使用本地 clone 的方式进行安装, 此时项目中调用的 BasicSR 路径还是 pip 安装的 BasicSR; 反过来, 如果先使用本地 clone 的方式进行安装, 随后又使用 pip 安装, 此时项目中调用的 BasicSR 路径还是本地 clone 下的 BasicSR (分别如图2.4和图2.5所示)。

a) 通过本地 clone 安装成功的时候, 此时使用 `pip list` 命令查看 basicsr 路径:

Package	Version	Location
absl-py	0.11.0	
addict	2.4.0	
async-generator	1.10	
attrs	20.3.0	
autopep8	1.5.4	
av	9.1.1	
basicsr	1.3.5	c:\f\topic\basicsr
beautifulsoup4	4.9.3	
blurhash	1.1.4	

图 2.2: 本地 clone 安装成功时的 basicsr 路径显示

b) 通过 pip 安装成功的时候, 此时使用 `pip list` 命令查看 basicsr 路径:

Package	Version	Location
absl-py	0.11.0	
addict	2.4.0	
async-generator	1.10	
attrs	20.3.0	
autopep8	1.5.4	
av	9.1.1	
basicsr	1.3.5	
beautifulsoup4	4.9.3	
blurhash	1.1.4	

图 2.3: pip 安装成功时的 basicsr 路径显示 (如果指向 anaconda 下的路径, 也是正常的)

- c) 如果先通过 pip 安装, 随后通过本地 clone 安装, 此时使用 pip list 命令查看 basicsr 路径:

Package	Version	Location
absl-py	0.11.0	
addict	2.4.0	
async-generator	1.10	
attrs	20.3.0	
autopep8	1.5.4	
av	9.1.1	
basicsr	1.3.5	
beautifulsoup4	4.9.3	
blurhash	1.1.4	

图 2.4: basicsr 路径并未指向本地 clone 的 BasicSR

- d) 如果先通过本地 clone 安装, 随后通过 pip 安装, 通过 pip list 命令查看此时 basicsr 路径:

Package	Version	Location
absl-py	0.11.0	
addict	2.4.0	
async-generator	1.10	
attrs	20.3.0	
autopep8	1.5.4	
av	9.1.1	
basicsr	1.3.5	c:\f\topic\basicsr
beautifulsoup4	4.9.3	
blurhash	1.1.4	

图 2.5: basicsr 路径并未指向 python 环境下 (或者 anaconda) 下的 BasicSR

对于上述的两种错误情况 (图2.4和图2.5), 此时正常的解决方式为: 先将安装的 BasicSR 进行卸载, 随后再根据项目的需要重新选择一种方式安装 BasicSR。

```
pip uninstall basicsr
```

4. Q4: 如何更新最新版本的 BasicSR?

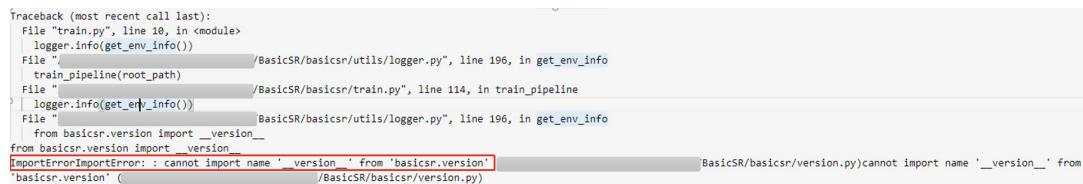
[回到主目录](#)

- a) 对于通过本地 clone 进行安装的方式，需要将本地的 BasicSR 项目代码与[远端的 BasicSR 项目代码](#)进行同步。
- b) 对于通过 pip 安装的方式，

```
pip install basicsr --upgrade
```

5. Q5: 如何解决运行代码时出现的 version 问题？

有时候在运行代码的时候，会出现类似于如下的问题：



```
Traceback (most recent call last):
  File "train.py", line 10, in <module>
    logger.info(get_env_info())
  File "/.../BasicSR/basicssr/utils/logger.py", line 196, in get_env_info
  File ".../train_pipeline(root_path)
  File ".../BasicSR/basicssr/train.py", line 114, in train_pipeline
    logger.info(get_env_info())
  File ".../BasicSR/basicssr/utils/logger.py", line 196, in get_env_info
    from basicsr.version import _version_
ImportError: cannot import name '_version_' from 'basicsr.version'
'basicsr.version' (/.../BasicSR/basicssr/version.py)
```

图 2.6: version 问题示例

此时，可以尝试：

- a) 重新运行安装 BasicSR 的指令。
- b) 将涉及到 version 的代码进行注释。

 如果小伙伴们在安装过程中还遇到其它的问题，可以在我们的 BasicSR 微信群、QQ 群（可从[BasicSR 项目主页](#)中获取）、github 的 issue 上面进行反馈，我们会持续将一些常见的问题更新到这个小节当中。

第 3 章

入门

本部分为 BasicSR 方法的入门部分，主要涉及的有目录解读，训练、测试和快速推理的流程。这个部分的主要目的是希望读者能够快速入门 BasicSR 整体框架。

§3.1. 目录解读

所谓“看书先看目录”。我们首先来看一下 BasicSR 仓库的基本结构，先来整体地把握一下。根据仓库的目录层级，第一部分为仓库的整体概览。这部分主要包括算法核心文件和代码基础配置文件。具体的目录结构如下。

其中，

红色 表示和跑实验直接相关的文件，即我们平时打交道最多的文件；

蓝色 表示其他与 BasicSR 强相关的代码文件；

黑色 表示配置文件。

BasicSR 根目录

·github/workflows	GitHub 的自动 workflows, 比如 PyLint、PyPI Publish 等
·vscode	VSCode 配置, 用于统一格式
LICENSE	使用的其他代码的 LICENSE 和 Acknowledgement
assets	存放仓库中展示使用的图片
basicsr	BasicSR 核心代码
colab	Google Colab 的 Notebook, 提供方便的 inference demo
datasets	“存放”使用的数据集, 推荐 soft link, 做到代码、数据的分离
docs	使用和说明文档
experiments	实验 checkpoints 保存路径
└ pretrained_models	预训练模型存放路径
inference	快速推理, 主要用于得到 demo 结果
options	训练和测试的配置文件
scripts	功能脚本, 包含数据集制作, 指标测试和数据集下载等
test_scripts	一些用于手动单元测试的脚本
tests	PyTest 自动单元测试
·gitignore	Git 忽略文件的配置
·pre-commit-config.yaml	Pre-commit Hook 的配置文件
·readthedocs.yaml	自动触发 basicsr.readthedocs.io 的配置文件
MANIFEST.in	发布 basicsr 时, 额外需要包含进去的文件

_ README.md	说明文档
_ README_CN.md	说明文档中文版
_ VERSION	版本文件
_ requirements.txt	安装依赖包文件
_ setup.cfg	格式配置文件，比如 flake8, yapf 和 isort
_ setup.py	安装文件

在 BasicSR 仓库中，核心代码在 basicsr 这个文件夹中。这个部分主要为深度学习模型常用的代码文件，比如网络结构，损失函数和数据加载等，具体目录如下。

其中，**红色** 表示我们在开发中主要修改的文件。

basicsr	
_ archs	定义网络结构和 forward 的步骤
_ data	定义 Dataset 来喂给模型的训练，Dataloader 的定义也在这里
_ losses	定义损失函数
_ metrics	定义评价指标，比如 PSNR, SSIM, NIQE 等
_ models	定义一次完整训练，比如前向、反向传播，梯度优化，Validation 等
_ ops	定义需要编译的算子，例如 StyleGAN 中用到的算子等
_ utils	定义基础工具，例如 file client, logger, registry, image process, matlab function 等
_ test.py	定义测试流程，是测试文件的主文件、入口
_ train.py	定义训练流程，是训练文件的主文件、入口

由于在算法设计和开发中，还需要用到一些脚本，比如数据的预处理、指标计算等，相关的文件位于 scripts，目录如下：

scripts	
_ data_preparation	准备数据
_ matlab_scripts	基于 MATLAB 语言的数据处理脚本
_ metrics	计算指标的脚本，例如 PSNR, SSIM, NIQE 等
_ model_conversion	模型转换的脚本，主要是 .pth 文件的 keys 转换
_ dist_test.sh	方便的分布式测试启动脚本
_ dist_train.sh	方便的分布式训练启动脚本
_ download_gdrive.py	从 Google Drive 下载文件的脚本
_ download_pretrained_models.py	从 Google Drive 批量下载预训练模型的脚本
_ publish_models	发布模型的脚本，包括添加 SHA 等

至此，我们对 BasicSR 的整体框架便有了一定的了解啦～

§3.2. 训练流程

在对目录结构有了初步的了解之后就可以进行训练了。我们希望 BasicSR 即方便使用，又清晰易懂，降低使用者的门槛。但随着 BasicSR 代码库逐渐抽象和复杂起来，很多刚接触的同学不知道程序入口在哪里，数据、模型和网络是在哪里定义的，流程又是在哪里控制的，那么我们就通过一个例子简要地说一下。

本节的目的是希望能够初步地让读者了解到训练的基本流程和代码逻辑流，具体的细节我们会采用引用的方式来供读者查阅。我们强烈建议你跟着下面的流程和实际代码，走一遍训练的流程。遮掩可以对 **BasicSR** 整体的框架有一个全面理解。

训练流程是从 `basicsr/train.py` 开始的。

3.2.1. 代码的入口和训练的准备工作

我们以训练超分辨率模型 MSRResNet 为例，首先需要在终端输入命令来开始训练：

```
python basicsr/train.py -opt  
→ options/train/SRResNet_SRGAN/train_MSResNet_x4.yml
```

其中 `options/train/SRResNet_SRGAN/train_MSResNet_x4.yml` 为 yml 配置文件，主要设置实验相关的配置参数。参数具体说明参见章节4.3.2.1：代码主体结构。

它从 `basicsr/train.py` 的 `train_pipeline` 函数作为入口：

```
basicsr > train.py > ...  
210     if __name__ == '__main__':  
211         root_path = osp.abspath(osp.join(__file__, osp.pardir, osp.pardir))  
212         train_pipeline(root_path)
```

图 3.1: 函数 `train_pipeline` 作为 `basicsr/train.py` 的入口

root_path 作为参数传进去：这里为什么要把 `root_path` 作为参数传进去呢？是因为，当我们把 `basicsr` 作为 package 使用的时候，需要根据当前的目录路径来创建文件，否则程序会错误地使用 `basicsr` package 所在位置的目录了。

`train_pipeline` 函数会做一些基础的事，比如：

1. 解析配置文件 option file，即 yml 文件
2. 设置 distributed training 的相关选项，设置 random seed 等
3. 如果有 resume，需要 load 相应的状态
4. 创建相关文件夹，拷贝配置的 yml 文件
5. 合理初始化日志系统 logger

我们对着代码一一讲解，如图3.2所示：

具体的子函数，大家可以对着代码点进去查看，这里我们着重说几点。

```

basicsr > train.py > train_pipeline
91  def train_pipeline(root_path):      You, 5 months ago • Major: Add Registry mechanism (#385)
92  ...# parse_options, set_distributed_setting, set_random_seed
93  ...opt, args = parse_options(root_path, is_train=True) 解析配置的yml文件。还有其他操作,
94  ...opt['root_path'] = root_path          比如设置dist training, 设置random seed等
95
96  ...torch.backends.cudnn.benchmark = True
97  ...# torch.backends.cudnn.deterministic = True
98
99  ...# load_resume_states_if_necessary    如果配置文件或者命令行有resume的指令，则会load相应的
100 ...resume_state = load_resume_state(opt).state 文件，用于resume
101 ...# mkdir for experiments and logger
102 ...if resume_state is None:
103 ...    make_exp_dirs(opt) 创建用于存放实验的一系列文件夹。如果resume则不需要创建
104 ...    if opt['logger'].get('use_tb_logger') and 'debug' not in opt['name'] and opt['rank'] == 0:
105 ...        mkdir_and_rename(osp.join(opt['root_path'], 'tb_logger', opt['name']))
106
107 ...# copy the yml file to the experiment root 把yml文件copy到实验文件夹里，方便后续查看
108 ...copy_opt_file(args.opt, opt['path']['experiments_root'])
109 【特别注意】在这以上的代码(及调用函数)中，不能使用 get_root_logger，否则会导致logger初始化错误
110 ...# WARNING: should not use get_root_logger in the above codes, including the called functions
111 ...# Otherwise the logger will not be properly initialized
112 ...log_file = osp.join(opt['path'][‘log’], f"train_{opt[‘name’]}_{get_time_str()}.log")
113 ...logger = get_root_logger(logger_name='basicsr', log_level=logging.INFO, log_file=log_file)
114 ...logger.info(get_env_info())           初始化日志系统logger，会有文件和屏幕的logger
115 ...logger.info(dict2str(opt))         在logger中输出环境信息，以及实际的且更加完善的配置信息
116 ...# initialize wandb and tb loggers
117 ...tb_logger = init_tb_loggers(opt) 根据配置需要，初始化tensorboard logger 和 wandb logger
118
119 ...# create train and validation dataloaders
120 ...result = create_train_val_dataloader(opt, logger)
121 ...train_loader, train_sampler, val_loader, total_epochs, total_iters = result

```

图 3.2: 函数 train_pipeline 的基础准备工作

- 我们在命令行中的参数输入，在哪里完成解析呢，即 argparse 在哪里？答：是在 parse_options 这个函数中。我们截取一部分来看一下。

从图3.3中，我们看到命令行的参数不多。我们一一讲解一下。

- opt，配置文件的路径，一般采用这个命令配置训练或者测试的 yml 文件。
- launcher，用于指定 distributed training 的，比如 pytorch 或者 slurm。默认是 none，即单卡非 distributed training。
- auto_resume，是否自动 resume，即自动查找最近的 checkpoint，然后 resume。详见章节4.5.4.1：代码主体结构。
- debug，能够快速帮助 debug。详见章节4.3.1.2：代码主体结构。
- local_rank，这个不用管，是 distributed training 中程序自动会传入。
- force_yml，方便在命令行中修改 yml 中的配置文件。详见章节4.3.3：代码主体结构。

- 每个实验创建的文件夹

每个实验都会在 experiments 目录中创建一个以配置文件中的 name 为名字的文件夹，里面的文件如图3.4所示。log 的内容参见章节4.9：代码主体结构。在实验文件夹中有把配置文件也 copy 一份，还会额外添加 copy 的时间和运行使用的具体命令，方便事后检查和复现。

```

basicsr > utils > options.py > parse_options
82 def parse_options(root_path, is_train=True):      You, 3 months ago * add debug in cmd args
83     parser = argparse.ArgumentParser()
84     parser.add_argument('--opt', type=str, required=True, help='Path to option YAML file.')
85     parser.add_argument('--launcher', choices=['none', 'pytorch', 'slurm'], default='none', help='job launcher')
86     parser.add_argument('--auto_resume', action='store_true')
87     parser.add_argument('--debug', action='store_true')
88     parser.add_argument('--local_rank', type=int, default=0)
89     parser.add_argument('--force_yaml', nargs='+', default=None, help='Force to update yaml files. Examples: train:ema_decay=0.999')
90     args = parser.parse_args()
91
92
93     # parse yaml to dict
94     with open(args.opt, mode='r') as f:
95         opt = yaml.load(f, Loader=ordered_yaml()[0])
96
97     # distributed settings
98     if args.launcher == 'none':
99         opt['dist'] = False
100        print('Disable distributed.', flush=True)
101    else:
102        opt['dist'] = True
103        if args.launcher == 'slurm' and 'dist_params' in opt:
104            init_dist(args.launcher, **opt['dist_params'])
105        else:
106            init_dist(args.launcher)
107        opt['rank'], opt['world_size'] = get_dist_info()
108
109    # random seed
110    seed = opt.get('manual_seed')
111    if seed is None:
112        seed = random.randint(1, 10000)
113    opt['manual_seed'] = seed
114    set_random_seed(seed + opt['rank'])
115
116    # force to update yaml options
117    if args.force_yaml is not None:
118        for entry in args.force_yaml:
119            ....下面还有其他的解析和设置, 比如文件路径, debug设置, train/test设置等

```

图 3.3: 函数 parse_options 解析参数输入

```

(pt17) -> 001_MSResNet_x4_f64b16_DIV2K_1000k_B16G1_wandb git:(master) 11
total 44K
drwxr-xr-x 3 user user 4.0K Sep 21 17:51 models  存放网络参数 .pth 文件
-rw-r--r-- 1 user user 280 Sep 21 17:51 train_001_MSResNet_x4_f64b16_DIV2K_1000k_B16G1_wandb_20210921_170513.log  log文件
drwxr-xr-x 1 user user 2.0K Sep 21 17:51 copy  配置文件copy
drwxr-xr-x 2 user user 4.0K Sep 21 17:51 training_states  存放用于resume的optimizer, scheduler等状态 .state文件
drwxr-xr-x 2 user user 4.0K Sep 21 17:51 visualization  存放validation的可视化图片等

```

图 3.4: 实验过程中创建的文件

3.2.2. Dataset 和 Model 的创建

当训练准备工作结束后，我们接下来就要看 dataset 和 model 的创建过程了。下图3.5是相对应的代码。它主要包括：

1. 训练和 validation 的 data loader 创建，下面会展开
2. model 的创建，下面会展开
3. logger 的初始化，这块详见章节4.9: 代码主体结构 的相关内容
4. 还有 dataset prefetch 的内容，这块详见章节4.4.4: 代码主体结构 的相关内容

这里我们着重讲解两块，dataloader 的创建和 model 的创建。

1. **dataloader 创建**。首先我们看调用的 create_train_val_dataloader 函数 (如图3.6所示)。

里面主要就是两个函数，build_dataset 和 build_dataloader。无论是 train 还是 val 的 dataloader 都是这两个函数构建的。创建 dataloader 要靠 build_dataloader，其中又要用到 dataset。而 dataset 是由 build_dataset 创建的。dataloader 其实大家都是共用的。当我们

```

basicsr > train.py > train_pipeline.py
117     tb_logger = init_tb_loggers(opt)
118
119     # create train and validation dataloaders
120     result = create_train_val_dataloader(opt, logger)
121     train_loader, train_sampler, val_loader, total_epochs, total_iters = result
122
123     # create model
124     model = build_model(opt)
125     if resume_state: # resume training
126         model.resume_training(resume_state) # handle optimizers and schedulers
127         logger.info(f'Resuming training from epoch: {resume_state["epoch"]}, iter: {resume_state["iter"]}')
128         start_epoch = resume_state['epoch']
129         current_iter = resume_state['iter'] - handle optimizers and schedulers
130     else:
131         start_epoch = 0
132         current_iter = 0
133
134     # create message logger (formatted outputs)
135     msg_logger = MessageLogger(opt, current_iter, tb_logger) 这个用来控制并format logger输出的信息
136
137     # dataloader prefetcher
138     prefetch_mode = opt['datasets'][train].get('prefetch_mode')
139     if prefetch_mode is None or prefetch_mode == 'cpu':
140         prefetcher = CPUPrefetcher(train_loader)
141     elif prefetch_mode == 'cuda':
142         prefetcher = CUDAPrefetcher(train_loader, opt)
143         logger.info(f'Use {prefetch_mode} prefetch dataloader')
144         if opt['datasets'][train].get('pin_memory') is not True:
145             raise ValueError('Please set pin_memory=True for CUDAPrefetcher.')
146     else:
147         raise ValueError(f'Wrong prefetch_mode {prefetch_mode}. Supported ones are: None, "cuda", "cpu".')
148
149     # training
150     logger.info(f'Start training from epoch: {start_epoch}, iter: {current_iter}')
151     data_timer, iter_timer = AvgTimer(), AvgTimer()

```

图 3.5: train_pipeline 中初始化 dataset 和 model

说要新写一个 dataloader，其实写的是 dataset。build_dataset 和 build_dataloader 都是定义在 basicsr/data/_init_.py 文件里。

```

basicsr > data > __init__.py
25 def build_dataset(dataset_opt):
26     """Build dataset from options.
27
28     Args:
29         dataset_opt (dict): Configuration for dataset. It must contain:
30             name (str): Dataset name.
31             type (str): Dataset type.
32     """
33     dataset_opt = deepcopy(dataset_opt)
34     dataset = DATASET_REGISTRY.get(dataset_opt['type'])(dataset_opt)
35     logger = get_root_logger()
36     logger.info(f'Dataset [{dataset.__class__.__name__}] --{dataset_opt["name"]} is built.')
37     return dataset
38
39
40 def build_dataloader(dataset, dataset_opt, num_gpu=1, dist=False, sampler=None, seed=None):
41     """Build dataloader.
42
43     Args:
44         dataset (torch.utils.data.Dataset): Dataset.
45         dataset_opt (dict): Dataset options. It contains the following keys:
46             phase (str): 'train' or 'val'.
47             num_worker_per_gpu (int): Number of workers for each GPU.
48             batch_size_per_gpu (int): Training batch size for each GPU.
49             num_gpu (int): Number of GPUs. Used only in the train phase.
50             Default: 1.
51             dist (bool): Whether in distributed training. Used only in the train
52             phase. Default: False.
53             sampler (torch.utils.data.sampler): Data sampler. Default: None.
54             seed (int | None): Seed. Default: None
55     """
56     phase = dataset_opt['phase']
57     rank, _ = get_dist_info()
58     if phase == 'train':
59         if dist: # distributed training

```

图 3.7: build_dataset 和 build_dataloader 的定义

```

basicsr > train.py > create_train_val_dataloader
29 def create_train_val_dataloader(opt, logger):
30     ... # create train and val dataloaders
31     train_loader, val_loader = None, None
32     for phase, dataset_opt in opt['datasets'].items():
33         if phase == 'train':
34             dataset_enlarge_ratio = dataset_opt.get('dataset_enlarge_ratio', 1)
35             train_set = build_dataset(dataset_opt) # Line 35 highlighted
36             train_sampler = EnlargedSampler(train_set, opt['world_size'], opt['rank'], dataset_enlarge_ratio)
37             train_loader = build_dataloader(
38                 train_set,
39                 dataset_opt,
40                 num_gpu=opt['num_gpu'],
41                 dist=opt['dist'],
42                 sampler=train_sampler,
43                 seed=opt['manual_seed'])
44
45             num_iter_per_epoch = math.ceil(
46                 len(train_set) * dataset_enlarge_ratio / (dataset_opt['batch_size_per_gpu'] * opt['world_size']))
47             total_iters = int(opt['train']['total_iter'])
48             total_epochs = math.ceil(total_iters / (num_iter_per_epoch))
49             logger.info('Training statistics:')
50             f'\n\tNumber of train images: {len(train_set)}'
51             f'\n\tDataset enlarge ratio: {dataset_enlarge_ratio}'
52             f'\n\tBatch size per gpu: {dataset_opt["batch_size_per_gpu"]}'
53             f'\n\tWorld size (gpu number): {opt["world_size"]}'
54             f'\n\tRequire iter number per epoch: {num_iter_per_epoch}'
55             f'\n\tTotal epochs: {total_epochs}; iters: {total_iters}.'
56
57         elif phase == 'val':
58             val_set = build_dataset(dataset_opt) # Line 58 highlighted
59             val_loader = build_dataloader(
60                 val_set, dataset_opt, num_gpu=opt['num_gpu'], dist=opt['dist'], sampler=None, seed=opt['manual_seed'])
61             logger.info(f'Number of val images/folders in {dataset_opt["name"]}: {len(val_set)}')
62         else:
63             raise ValueError(f'Dataset phase {phase} is not recognized.')
64
65     return train_loader, train_sampler, val_loader, total_epochs, total_iters

```

图 3.6: 初始化 train 和 valid 的 dataloader

这里面，`build_dataset` 是核心 (如图3.7)。它会根据配置文件 yml 中的 dataset 类型，比如在我们这个例子中就是 `PairedImageDataset`，创建相应的实例。核心的代码是：`DATASET_REGISTRY.get()`。这里是如何做到根据“类名”动态创建实例的，请参见章节4.2: 代码主体结构。(实例就是由类 class 创建的，具体运行的对象)。这里我们只要理解，通过这一句调用，就可以创建相应的实例了。`build_dataloader` 是比较容易理解的。它根据传入的 dataset 和其他在 yml 中的参数，构建 dataloader。

2. **model 的创建**。 `model` 的创建是通过 `build_model` 这个函数，定义在 `basicsr/models/__init__.py` 文件里，简略图参见图3.8。

```

basicsr > models > __init__.py > ...
19 def build_model(opt):
20     """Build model from options.
21
22     Args:
23         opt (dict): Configuration. It must contain:
24             model_type (str): Model type.
25
26             根据配置文yml中的 model 类型
27             opt = deepcopy(opt) 靠着MODEL_REGISTRY.get() 就可以创建相应的model的实例了
28             model = MODEL_REGISTRY.get(opt['model_type'])(opt)
29             logger = get_root_logger()
30             logger.info(f'Model [{model.__class__.__name__}] is created.')
31
32     return model

```

options > train > SRResNet_SRGAN > train_MSResNet_x4.yml
4 # general settings
5 name: 001_MSResNet_x4_f64b16_DIV2K_1000k_B16G1_wandb
6 model_type: SRModel # Line 6 highlighted
7 scale: 4
8 num_gpu: 1 # set num_gpu: 0 for cpu mode

图 3.8: 基于 `build_model` 创建 model 实例

`build_model` 会根据配置文件 yml 中的 model 类型，比如在我们这个例子中就是 `SRModel`，创建相应的实例。

接下来我们再具体地看看 `SRModel` 这个实例的创建过程吧，以便更好地理解一个模型中做了什么操作。让我们进入 `SRModel` 这个类。简略图参见图3.9。

```

basicsr > models > sr_model.py > ...
14     @MODEL_REGISTRY.register()    注册SRModel。具体的注册机制会在后续文章中说明
15     class SRModel(BaseModel): SRModel继承于BaseModel。BaseModel提供了model共用的一些函数，会在后续文章中细说
16         """Base SR model for single image super-resolution."""
17
18     >     def __init__(self, opt): ... 初始化SRModel类，比如 定义网络和load weight
19
20     >     def init_training_settings(self): ... 初始化与训练相关的，比如loss，设置optimizers和schedulers
21
22     >     def setup_optimizers(self): ... 具体设置optimizer，可以根据实际需求，对params设置多组不同的optimizer
23
24     >     def feed_data(self, data): ... 提供数据，是与dataloader (dataset) 的接口
25
26     >     def optimize_parameters(self, current_iter): ... 优化参数，即一个完整train的step。包括了forward, loss计算,
27                                backward, 参数优化等
28
29     >     def test(self): ... 测试流程
30
31
32     >     def dist_validation(self, dataloader, current_iter, tb_logger, save_img): ...
33         validation的流程，分了dist (多卡一起做validation) 和 nondist (只有单卡做validation)
34
35     >     def nondist_validation(self, dataloader, current_iter, tb_logger, save_img): ...
36
37     >     def _log_validation_metric_values(self, current_iter, dataset_name, tb_logger): ...
38         控制如何打印validation的结果
39
40     >     def get_current_visuals(self): ... 得到网络输出的结果。这个函数会在validation中用到 (实际可以简化掉)
41
42     >     def save(self, epoch, current_iter): ... 保存网络 (.pth文件) 以及训练状态 (.state文件)
43
44

```

图 3.9: SR model 类的定义

在这里我们主要关注以下几个方面，关于 model 具体的介绍，参见章节4.5: 代码主体结构。

- a) network 的创建
- b) loss 的创建
- c) optimize_parameters，即一个 iteration 的 train step
- d) metric 的使用

下面我们分别简略说明，希望大家可以抓住大致的脉络。

- a) **network 的创建**一般是在 model 的 `__init__()` 函数里面，是通过调用 `build_network()` 实现的。`__init__()` 函数一般还会加载预训练模型，并初始化训练相关的设置。如图3.10所示。

```

basicsr > models > sr_model.py > SRModel > __init__
18     def __init__(self, opt): ... You, a year ago * update to V1.0 (Merge pull request #250 from xinn...
19         super(SRModel, self).__init__(opt)
20
21         # define network
22         self.net_g = build_network(opt['network_g'])           定义网络结构
23         self.net_g = self.model_to_device(self.net_g)          根据参数，实例化网络结构
24         self.print_network(self.net_g)
25
26         # load pre-trained models
27         load_path = self.opt['path'].get('pretrain_network_g', None)
28         if load_path is not None:
29             param_key = self.opt['path'].get('param_key_g', 'params')
30             self.load_network(self.net_g, load_path, self.opt['path'].get('strict_load_g', True), param_key)
31
32         if self.is_train:
33             self.init_training_settings()                         加载预训练模型
34
35     def init_training_settings(self):
36         self.net_g.train()

```

图 3.10: 模型初始化 - 创建网络结构

`build_network` 会根据配置文件 `yml` 中的 `network` 类型，比如在我们这个例子中就是 `MSRResNet`，从 `ARCH_REGISTRY` 创建相应的实例。如图3.11所示。

The screenshot shows a terminal window with Python code and a configuration file. The code in `archs/_init_.py` contains a `build_network` function that creates a network instance based on the `network_type` from the configuration file. A red box highlights the `ARCH_REGISTRY.get(network_type)(**opt)` line. A green arrow points from this line to the `type: MSRResNet` entry in the configuration file. The configuration file also includes other parameters like `num_in_ch: 3`, `num_out_ch: 3`, and `num_feat: 64`.

```

basicsr > archs > __init__.py > ...
19 def build_network(opt):
20     opt = deepcopy(opt)
21     network_type = opt.pop('type') 根据配置文件yml中的arch类型，可创建相应的network实例
22     net = ARCH_REGISTRY.get(network_type)(**opt)
23     logger = get_root_logger()
24     logger.info(f'Network [{net.__class__.__name__}] is created.')
25     return net
26

```

图 3.11: `build_network` 说明: 根据 yml 配置文件中的网络结构类型, 创建相应实例

- b) **loss 的创建**一般是在 model 的 `init_training_settings()` 函数里面。其他先不关注, 我们主要关注 `build_loss` 这个函数。loss 就是通过调用 `build_loss()` 实现的。如果有多个 loss , 则会多次调用 `build_loss()` , 创建多个 loss 实例。如图3.12所示。

The screenshot shows the `SRModel` class definition in `sr_model.py`. The `init_training_settings` method contains logic to create losses based on configuration. Red boxes highlight the `build_loss` calls for pixel and perceptual losses. A green arrow points from the `pixel_opt` entry in the configuration file to the corresponding `build_loss` call in the code.

```

basicsr > models > sr_model.py > SRModel
35 def init_training_settings(self):
36     self.net_g.train()
37     train_opt = self.opt['train']
38
39     self.ema_decay = train_opt.get('ema_decay', 0)
40     if self.ema_decay > 0:
41
42         # define losses
43         if train_opt.get('pixel_opt'):
44             self.cri_pix = build_loss(train_opt['pixel_opt']) to(self.device) pixel loss
45             else:
46                 self.cri_pix = None
47
48         if train_opt.get('perceptual_opt'):
49             self.cri_perceptual = build_loss(train_opt['perceptual_opt']) to(self.device) perceptual loss
50             else:
51                 self.cri_perceptual = None
52
53         if self.cri_pix is None and self.cri_perceptual is None:
54             raise ValueError('Both pixel and perceptual losses are None.')
55
56         # set up optimizers and schedulers
57         self.setup_optimizers()
58         self.setup_schedulers()
59

```

图 3.12: `SR_model` 类中, 使用 `build_loss` 创建 loss

`build_loss` 会根据配置文件 yml 中的 loss 类型, 比如在我们这个例子中就是 `L1Loss` , 从 `LOSS_REGISTRY` 中创建相应的实例。如图3.13所示。

The screenshot shows the `build_loss` function in `losses/_init_.py`. It creates a loss instance from the configuration. A red box highlights the `LOSS_REGISTRY.get(loss_type)(**opt)` line. A green arrow points from the `type: L1Loss` entry in the configuration file to this line.

```

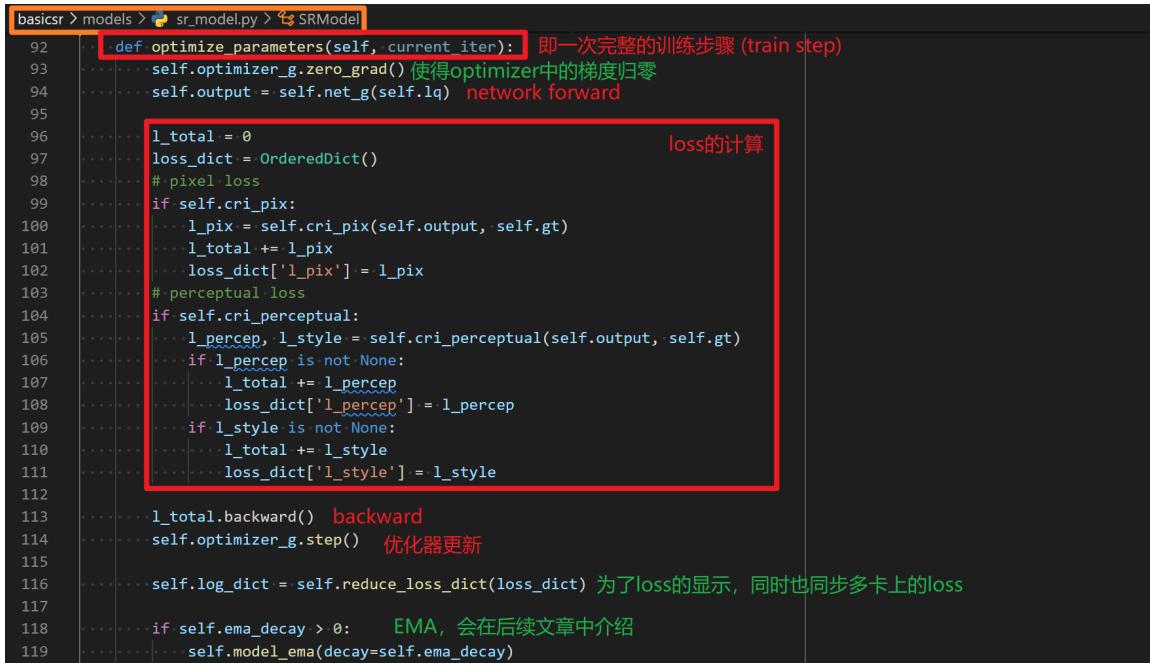
basicsr > losses > __init__.py > ...
14 def build_loss(opt):
15     """Build loss from options.
16
17     Args:
18         opt (dict): Configuration. It must constrain:
19             type (str): Model type.
20     """
21
22     opt = deepcopy(opt)
23     loss_type = opt.pop('type') 根据配置文件yml中的loss类型和参数, 实例化loss
24     loss = LOSS_REGISTRY.get(loss_type)(**opt)
25     logger = get_root_logger()
26     logger.info(f'Loss [{loss.__class__.__name__}] is created.')
27     return loss
28

```

图 3.13: `build_loss` 说明: 根据 yml 配置文件中的 loss 类型, 创建相应实例

- c) **optimize_parameter** 函数, 即一个 iteration 下的 train step 。这个函数里面主要包含了 network forward , loss 计算, backward 和优化器的更新。如图3.14所示。
- d) **metric** 的使用主要是在 validation 里面。我们来看在训练 MSRResNet 中调用的 `nondist_validation` 函数。其中核心是在 `calculate_metric` 这个函数, 它会根据配置文件 yml 中的 metrics 配置, 调用相应的函数。如图3.15所示。

`calculate_metric` 具体定义在 `basicsr/metrics/_init_()` 文件中, 它也是使用了 REGISTRY 机制: METRIC_REGISTRY。它会根据配置文件 yml 中的 metric 类型, 比如

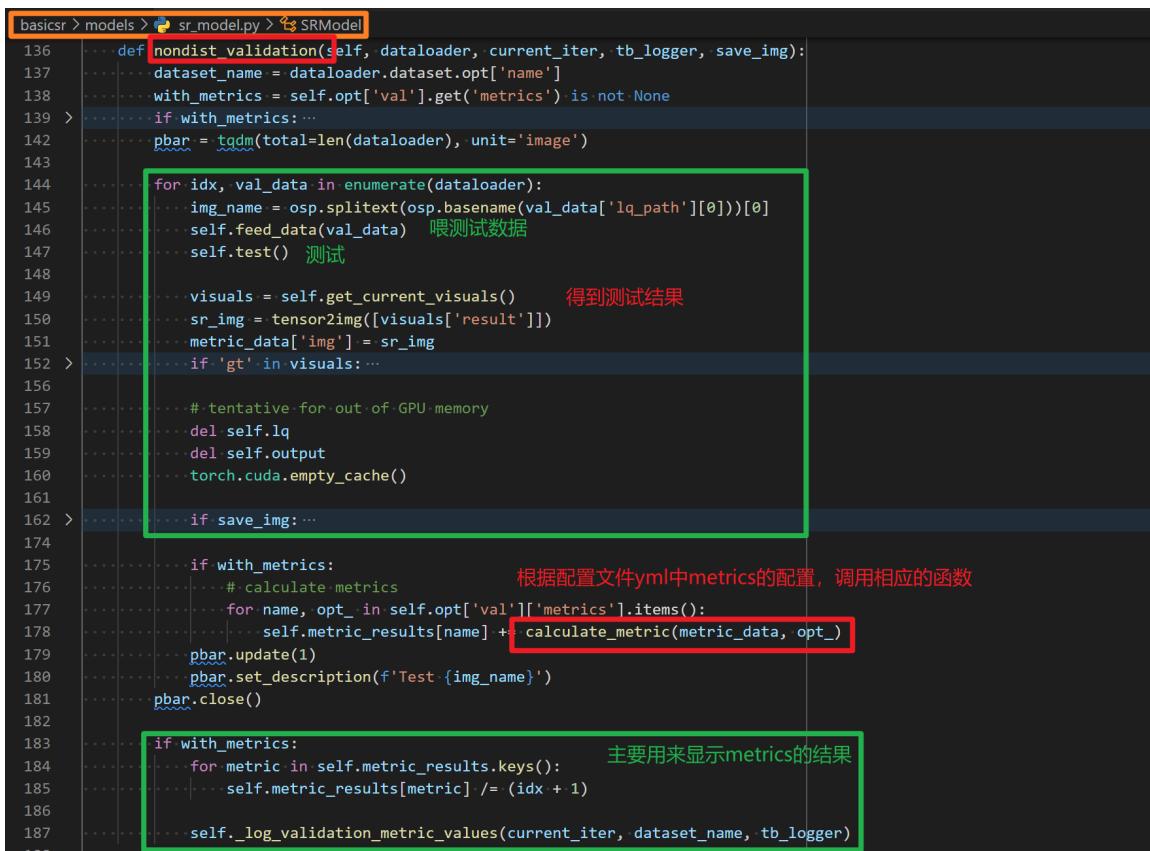


```

92     def optimize_parameters(self, current_iter):
93         self.optimizer_g.zero_grad() 使得optimizer中的梯度归零
94         self.output = self.net_g(self.lq) network forward
95
96         l_total = 0
97         loss_dict = OrderedDict()
98         # pixel loss
99         if self.cri_pix:
100             l_pix = self.cri_pix(self.output, self.gt)
101             l_total += l_pix
102             loss_dict['l_pix'] = l_pix
103         # perceptual loss
104         if self.cri_perceptual:
105             l_percep, l_style = self.cri_perceptual(self.output, self.gt)
106             if l_percep is not None:
107                 l_total += l_percep
108                 loss_dict['l_percep'] = l_percep
109             if l_style is not None:
110                 l_total += l_style
111                 loss_dict['l_style'] = l_style
112
113         l_total.backward() backward
114         self.optimizer_g.step() 优化器更新
115
116         self.log_dict = self.reduce_loss_dict(loss_dict) 为了loss的显示，同时也同步多卡上的loss
117
118         if self.ema_decay > 0: EMA, 会在后续文章中介绍
119             self.model_ema(decay=self.ema_decay)

```

图 3.14: optimize_parameter 函数: 一个 iteration 的参数优化过程



```

136     def nondist_validation(self, dataloader, current_iter, tb_logger, save_img):
137         dataset_name = dataloader.dataset.opt['name']
138         with_metrics = self.opt['val'].get('metrics') is not None
139
140         pbar = tqdm(total=len(dataloader), unit='image')
141
142         for idx, val_data in enumerate(dataloader):
143             img_name = osp.splitext(osp.basename(val_data['lq_path'][0]))[0]
144             self.feed_data(val_data) 喂测试数据
145             self.test() 测试
146
147             visuals = self.get_current_visuals() 得到测试结果
148             sr_img = tensor2img([visuals['result']])
149             metric_data['img'] = sr_img
150             if 'gt' in visuals:
151
152                 # tentative for out of GPU memory
153                 del self.lq
154                 del self.output
155                 torch.cuda.empty_cache()
156
157             if save_img:
158
159                 if with_metrics:
160                     # calculate metrics 根据配置文件yml中metrics的配置，调用相应的函数
161                     for name, opt_ in self.opt['val']['metrics'].items():
162                         self.metric_results[name] += calculate_metric(metric_data, opt_)
163
164                     pbar.update(1)
165                     pbar.set_description(f'Test {img_name}')
166                     pbar.close()
167
168                 if with_metrics:
169                     for metric in self.metric_results.keys():
170                         self.metric_results[metric] /= (idx + 1)
171
172                     self._log_validation_metric_values(current_iter, dataset_name, tb_logger)

```

图 3.15: validation 中 metric 的使用: 基于 yml 文件配置调用对应的 metric 函数

在我们这个例子中就有两个 metrics: PSNR 和 SSIM , 调用相应的函数。注意和前面 DATASET, ARCH, MODEL, LOSS 的 REGISTRY 不同, 这里返回的是函数调用, 而其他返回的是类的实例。如图3.16所示。

到此, 我们已经看到了 dataset (data loader) 的创建, 以及 model 的创建。model 的创建包含了 network architecture 和 loss 的创建, 一次完整的训练流程以及 validation 中用到的 metric 的计算。

```

basicsr > metrics > __init__.py > ...
7     __all__ = ['calculate_psnr', 'calculate_ssim', 'calculate_niqe']
8
9
10    def calculate_metric(data, opt):
11        """Calculate metric from data and options.
12
13        Args:
14            opt (dict): Configuration. It must contain:
15            type (str): Model type.
16
17        """
18
19        opt = deepcopy(opt)
20        metric_type = opt.pop('type') 根据配置文件yml中metrics设置，调用相应的函数
21        metric = METRIC_REGISTRY.get(metric_type)(**data, **opt)
22
23        return metric

```

```

options > train > SRResNet_SRGAN > train_MSResNet_x4.yml
92     metrics:
93         psnr: # metric name can be arbitrary
94             type: calculate_psnr PSNR
95             crop_border: 4
96             test_y_channel: false
97         niqe:
98             type: calculate_niqe NIQE
99             crop_border: 4

```

图 3.16: calculate_metric 函数: 基于 yml 文件配置中 metric 的类型, 调用相应的函数

3.2.3. 训练过程

当以上这些部件都被创建后, 就进入训练过程了。它就是一个循环的过程, 不断地喂数据, 然后不断执行训练步骤。整个训练过程如图3.17所示。看图中的说明基本就能明白大概啦, 这里就不赘述了。

```

basicsr > train.py > train_pipeline
149     # training
150     logger.info(f'Start training from epoch: {start_epoch}, iter: {current_iter}')
151     data_timer, iter_timer = AvgTimer(), AvgTimer()
152     start_time = time.time()
153
154     for epoch in range(start_epoch, total_epochs + 1): 虽然这里是以epoch为外层循环, 但实际我们是以
155         train_sampler.set_epoch(epoch) iteration来判断训练是否结束
156         prefetcher.reset()
157         train_data = prefetcher.next()
158
159         while train_data is not None: 当每一个epoch还有数据, 就进入一次iteration的训练
160             data_timer.record()
161
162             current_iter += 1
163             if current_iter > total_iters: 达到设置的最大训练iteration数目, 训练停止
164                 break
165             # update learning rate 更新learning rate
166             model.update_learning_rate(current_iter, warmup_iter=opt['train'].get('warmup_iter', -1))
167             # training
168             model.feed_data(train_data) 喂数据
169             model.optimize_parameters(current_iter) 一次训练过程 (train step)
170             iter_timer.record()
171             if current_iter == 1:
172                 # reset start time in msg_logger for more accurate eta_time
173                 # not work in resume mode
174                 msg_logger.reset_start_time()
175             # log
176             if current_iter % opt['logger']['print_freq'] == 0: ... log 常用信息, 比如iter数据, loss等
177
178             # save models and training states
179             if current_iter % opt['logger']['save_checkpoint_freq'] == 0:
180                 logger.info('Saving models and training states.')
181                 model.save(epoch, current_iter) 每隔一段时间, 保存模型 (.pth文件)和训练状态 (.state文件)
182
183             # validation
184             if opt.get('val') is not None and (current_iter % opt['val']['val_freq'] == 0):
185                 model.validation(val_loader, current_iter, tb_logger, opt['val']['save_img'])
186                     每隔一段时间, 做validation
187
188             data_timer.start()
189             iter_timer.start()
190             train_data = prefetcher.next()
191
192         # end of iter 一次iteration结束
193
194     # end of epoch 一个epoch结束

```

图 3.17: 完整的训练过程

上面的 for 循环结束, 整个训练过程也就结束啦。

§3.3. 测试流程

这里的测试流程指的是，使用 `basicsr/test.py` 和 配置文件 `yml` 来测试模型，以得到测试结果，同时输出指标结果的过程。

测试流程是从 `basicsr/test.py` 开始的。

在测试阶段，很多流程（比如 dataset 和 data loader 的创建、model 的创建、网络结构的创建）都和训练流程是共用的。因此我们在这里主要解释测试流程中特有的部分。

测试阶段，我们需要在终端输入命令来开始训练。

```
1 python basicsr/test.py -opt options/test/SRResNet_SRGAN/test_MSResNet_x4.yml
```

其中 `options/test/SRResNet_SRGAN/test_MSResNet_x4.yml` 为 `yml` 配置文件，主要设置实验相关的配置参数。参数具体说明参见章节4.3.2.2：代码主体结构。

下面是 `basicsr/test.py` 主要的测试流程 `test_pipeline` 函数，相比于 `basicsr/train.py` 着实简单了很多。

```
def test_pipeline(root_path):
    # 解析 yml 文件，加载配置参数
    opt, _ = parse_options(root_path, is_train=False)
    ...

    # 新建 logger 并初始化，打印基础信息
    make_exp_dirs(opt)
    log_file = osp.join(opt['path']['log'],
        f"test_{opt['name']}_{get_time_str()}.log")
    logger = get_root_logger(logger_name='basicsr', log_level=logging.INFO,
        log_file=log_file)
    logger.info(get_env_info())
    logger.info(dict2str(opt))

    # 创建测试集和 dataloader。和训练过程一样，调用 build_dataset 和
    # build_dataloader
    test_loaders = []
    for _, dataset_opt in sorted(opt['datasets'].items()):
        test_set = build_dataset(dataset_opt)
        test_loader = build_dataloader(
            test_set, dataset_opt, num_gpu=opt['num_gpu'], dist=opt['dist'],
            sampler=None, seed=opt['manual_seed'])
        logger.info(f"Number of test images in {dataset_opt['name']}:
            {len(test_set)}")
        test_loaders.append(test_loader)

    # 创建模型，和训练过程一样，调用 build_model
    model = build_model(opt)
```

```
# 测试多个测试集，调用的是 model 里面的 validation 函数
for test_loader in test_loaders:
    test_set_name = test_loader.dataset.opt['name']
    logger.info(f'Testing {test_set_name}...')
    model.validation(test_loader, current_iter=opt['name'],
                     tb_logger=None, save_img=opt['val']['save_img'])
```

可以看到，整个测试过程和训练过程大部分都是重合的，非常简洁。

§3.4. 推理流程

这里的推理流程指的是，使用 `inference` 目录下的代码，快速方便地测试结果。和测试流程（第3.3小节）的目的是不同的：

- 测试流程针对学术数据集，希望能够同时测试多个测试集，同时能够输出相应的指标
- 推理流程针对实际使用场景，提供 demo。它往往只需要一个输出结果，而不需要有 GT (Ground-Truth) 数据，也不需要有指标输出。

简而言之，推理流程方便用户快速得到一个 demo 的结果。因此我们希望 `inference` 的文件，能够尽可能少的依赖 BasicSR 框架，即可以自己读数据，创建模型。我们只需要使用 BasicSR 中的网络结构即可（而网络结构在 BasicSR 中又是相对独立的）。这样，使用者便可以根据 `inference` 文件，快速将所需要的模型“摘”出来，放到自己的应用场景里面去。

在快速推理阶段，我们只需要在终端输入命令：

```
python inference/inference_esrgan.py --input input_path --output out_path
```

`basicsr/inference/inference_esrgan.py` 提供了一个非常简洁且具有代表性的例子，相信你可以轻而易举地看懂。

第 4 章

代码主体结构

在本章节中，我们将对 BasicSR 代码框架进行一个整体介绍，主要包括以下内容：整体框架（第4.1小节）、注册器机制（第4.2小节）、配置文件（第4.3小节）、数据（第4.4小节）、模型（第4.5小节）、网络结构（第4.6小节）、损失函数（第4.7小节）、算子（第4.8小节）、日志系统（第4.9小节）等。通过阅读本章，你将对 BasicSR 有进一步的认识，理解其模块之间的相互关系、以及模块内部的核心工作原理。但我们不对具体函数和代码做具体介绍。如果需要具体函数和代码的介绍，请查阅 BasicSR 的在线 API 文档 (<https://basicsr.readthedocs.io/en/latest/>)。

§4.1. 整体框架

对于基于深度学习的算法框架，其核心的组成部分包括：数据、模型、损失函数、训练。BasicSR 框架也是大致根据以上部分撰写而成的。下图概括了 BasicSR 的整体组成框架：

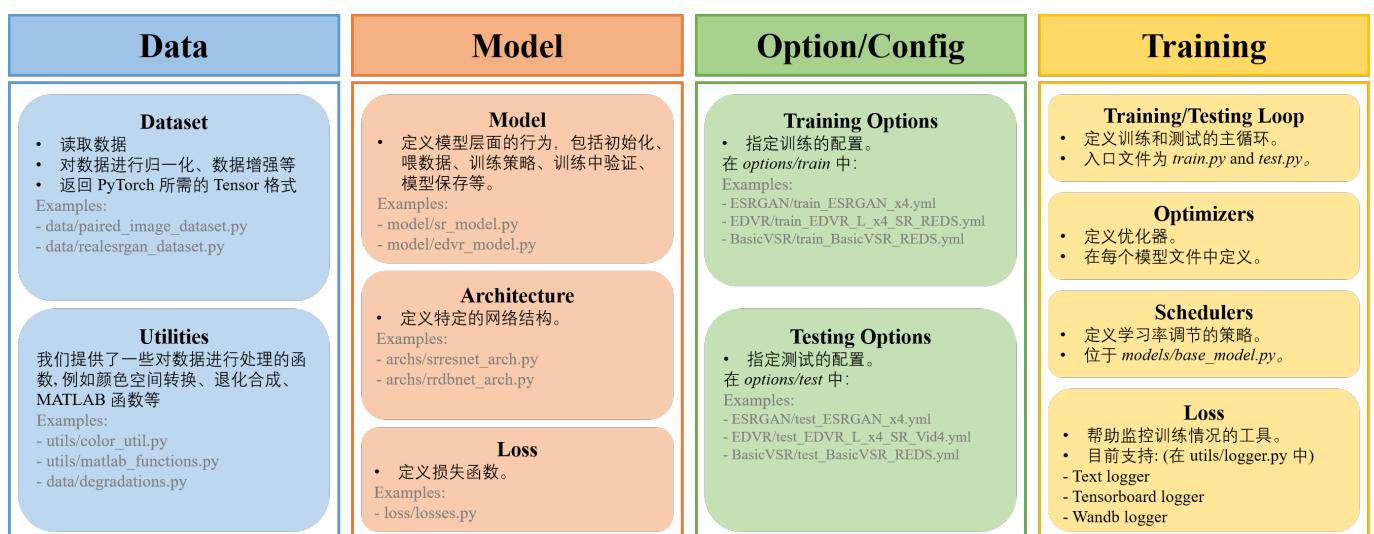


图 4.1: BasicSR 代码整体框架

- 数据 (Data): 这个部分主要定义了 Dataset 和 Data Loader 文件，放在了 `basicsr/data` 目录下。Dataset 用于读取和预处理数据，包括图像读取、归一化 (normalization)、数据增强 (augmentation) 以及封装为 PyTorch Tensor 等。同时，我们也提供了一些辅助函数，帮助使用者自定义自己的数据预处理功能，例如图像色彩空间转换、常用 MATLAB 函数的

Python 版本、常用的图像退化模型 (degradation model) 等。详细说明参见章节4.4: 代码主体结构

- 模型 (Model): 在 `basicsr/models` 目录下, 我们提供了常用的模型文件。这些模型文件主要用于定义网络结构与初始化、输入输出数据、一次 forward 的训练过程、保存加载模型等。在 `basicsr/archs` 目录下, 我们提供了常用的网络结构模型文件, 包括 SRResNet、ESRGAN、RCAN、SwinIR、EDVR、BasicVSR 等。在 `basicsr/losses` 文件夹中, 我们提供了常用的损失函数, 例如 L1/L2 loss、perceptual loss、GAN loss 等。详细说明参见章节4.5: 代码主体结构、章节4.6: 代码主体结构、章节4.7: 代码主体结构
- 配置 (Option): 配置文件放到 `option` 目录下。我们提供了常用模型的训练和测试配置文件。我们使用 `YAML` 来作为配置文件的语言。修改这些 `yml` 文件可以简易地调整训练过程中的各种超参数。详细说明参见章节4.2: 代码主体结构和章节4.3: 代码主体结构
- 训练 (Training): 这一部分主要涉及训练的策略和记录训练日志。`basicsr/train.py` 和 `basicsr/test.py` 是启动模型训练和测试的入口文件, 其中定义了训练和测试的 main loop。常见优化器 (optimizer) 的定义可以在 `models/base_model.py` 文件中的 `get_optimizer` 函数中找到。学习率的调度策略在 `models/base_model.py` 文件中的 `setup_schedulers` 函数中定义。为了方便追踪记录训练的过程, 我们提供了相应的 logger 工具, 支持直接 print 到屏幕、Tensorboard、Wandb等多种方式, 具体代码可以在 `basicsr/utils/logger.py` 中找到。详细说明参见章节4.9: 代码主体结构
- 详细的代码接口文档可以在 <http://basicsr.readthedocs.io> 查询

■ BasicSR 目录说明

你可以在章节3.1: 入门 查看到详细的 BasicSR 目录说明。

§4.2. 动态实例化与 REGISTER 注册机制

4.2.1. REGISTER 注册机制

首先, 来看我们的目的: 当我们新写了类 (Class) 或函数时, 我们希望可直接在配置文件中指定, 然后程序会根据配置文件的类名或函数名, 自动查找并实例化。以开发新的网络结构为例, 我们会做以下几件事:

1. 写具体的网络结构, 它往往是一个Class, 并且往往是一个单独的文件
2. 在配置文件中会指定我们使用哪一个网络结构, 往往是通过 Class name 指定
3. 在训练过程的某一个地方, 程序会根据配置文件指定的 Class name, 自动实例化相关的类

这里说的 REGISTER 注册机制就是来更简洁地实现上面的第三个步骤的。因为其能够根据配置文件动态地实例化所需要的类或函数, 因此这个过程被称为动态实例化 (Dynamic Instantiation)。

BasicSR 的 Register 注册机制参考了 FacebookResearch 的 `fvcore` 仓库的函数, 定义了 Registry 类。详细代码可查看 `basicsr/utils/registry.py`。它主要有两个函数: `register()` 和 `get()`。

```

1  class Registry():
2      """
3          The registry that provides name -> object mapping, to support third-party
4          ↳ users' custom modules.
5      """
6
7      def __init__(self, name):
8          self._name = name
9          self._obj_map = {}
10
11     def _do_register(self, name, obj, suffix=None):
12         ...
13         self._obj_map[name] = obj
14
15     def register(self, obj=None, suffix=None):
16         # register() 函数主要用来注册一个实现的类或函数
17         if obj is None:
18             # used as a decorator
19             def deco(func_or_class):
20                 name = func_or_class.__name__
21                 self._do_register(name, func_or_class, suffix)
22                 return func_or_class
23
24             return deco
25
26         # used as a function call
27         name = obj.__name__
28         self._do_register(name, obj, suffix)
29
30     def get(self, name, suffix='basicsr'):
31         # get() 函数主要用来根据配置文件中的类名或函数名来查找对应的实例
32         ret = self._obj_map.get(name)
33         if ret is None:
34             ret = self._obj_map.get(name + '_'+ suffix)
35         ...
36
37         return ret

```

4.2.1.1. 如何注册新的类?

在 BasicSR 中，我们定义了五个 REGISTER，相关定义在 basicsr/utils/registry.py 中：

```

1  DATASET_REGISTRY = Registry('dataset')
2  ARCH_REGISTRY = Registry('arch')
3  MODEL_REGISTRY = Registry('model')

```

```

4 LOSS_REGISTRY = Registry('loss')
5 METRIC_REGISTRY = Registry('metric')

```

需要注册的时候，我们

1. import 相关的注册器，比如 ARCH_REGISTRY
2. 使用 Python 装饰器，即在类/函数前面加上 @ARCH_REGISTRY.register()

以网络结构 RRDBNet 为例：

```

1 from basicsr.utils.registry import ARCH_REGISTRY # import 相关的注册器
2 from .arch_util import default_init_weights, make_layer, pixel_unshuffle
3
4 @ARCH_REGISTRY.register() # 使用 Python 装饰器
5 class RRDBNet(nn.Module):
6     def __init__(self):
7         super(RRDBNet, self).__init__()
8         ...

```

这样 RRDBNet 这个类就被注册上了。

4.2.1.2. 如何使用已注册的类？

当我们需要使用的时候，只需要在配置文件中配置，相关代码会自动实例化所需的类。

还是以使用 RRDBNet 网络结构为例。我们在配置文件中指定了网络结构的类型为 RRDBNet。

```

1 # network structures
2 network_g:
3     type: RRDBNet
4     num_in_ch: 3
5     num_out_ch: 3
6     num_feat: 64
7     num_block: 23

```

模型文件 `basicr/models/sr_models.py` 中的函数 `build_network(opt['network_g'])` 便会根据配置文件 build 网络结构。

```

1 class SRModel(BaseModel):
2     """Base SR model for single image super-resolution."""
3
4     def __init__(self, opt):
5         super(SRModel, self).__init__(opt)

```

```

6
7     # define network
8     self.net_g = build_network(opt['network_g'])  # 调用建构网络结构的函数
9     self.net_g = self.model_to_device(self.net_g)
10    self.print_network(self.net_g)

```

而 `build_network` 函数就会从 `ARCH_REGISTRY` 中找到已经被注册的 `RRDBNet` 进行实例化。`build_network` 函数所在位置 `basicsr/archs/__init__.py`

```

1 def build_network(opt):
2     opt = deepcopy(opt)
3     network_type = opt.pop('type')
4     net = ARCH_REGISTRY.get(network_type)(**opt)  # 实例化的核心函数。从
      → ARCH_REGISTRY 找到已被注册的类进行实例化
5     logger = get_root_logger()
6     logger.info(f'Network [{net.__class__.__name__}] is created.')
7     return net

```

4.2.2. 自动扫描并 import 注册的类/函数

上面讲了 `REGISTER` 注册机制，但还有一个问题：这个类/函数还需要被 Python 程序感知到。目前的类/函数只是在一个文件中写了，但是 Python 程序并没有 `import` 进来。

这个问题的解决方法：一般是在 `__init__.py` 文件中写 `import` 语句，比如在 `mmediting` 中定义了网络结构后，还需要在 `sr_backbones/__init__.py` 中添加这样的语句：

```

1 from .basicvsr_net import BasicVSRNet
2 from .edsr import EDSR
3 from .edvr_net import EDVRNet
4 from .rrdb_net import RRDBNet
5 from .sr_resnet import MSRResNet

```

它需要在 `__init__.py` 中把写好的网络结构显示地 `import` 进来。但这样很麻烦：每写一个新的网络结构，我们就要去修改 `__init__.py`，很多时候容易忘记。最早开发 `mmediting` 的时候就感觉这个很繁琐。本着偷懒的原则，在 `BasicSR` 中就找了一个省事的办法。

但我们新建一个文件，就以 `_arch.py` 结尾。通过扫描特定文件的方式来进行自动 `import`。比如，对于网络结构，我们在 `basicsr/archs/__init__.py` 中定义了：

```

1 # automatically scan and import arch modules for registry
2 # scan all the files under the 'archs' folder and collect files ending with
      → '_arch.py'
3 arch_folder = osp.dirname(osp.abspath(__file__))
4 arch_filenames = [osp.splitext(osp.basename(v))[0] for v in
      → scandir(arch_folder) if v.endswith('_arch.py')]

```

```

5  # import all the arch modules
6  _arch_modules = [importlib.import_module(f'basicsr.archs.{file_name}') for
→   file_name in arch_filenames]

```

我们可以看到，程序会自动扫描以 `_arch.py` 的文件，然后 `import` 相关文件。这样就把所有注册的网络结构类都 `import` 进去啦。

类似的，`DATASET`，`ARCH`，`MODEL`，`LOSS` 都在相关的 `__init__.py` 文件中定义了扫描并自动 `import` 的操作。

总结一下，当我们在新开发网络结构时（其他模块也类似），只要做两件事，修改两个文件就好了。`BasicSR` 背后的动态实例化和 `REGISTRY` 机制会帮你完成剩下的事。

1. 写一个单独的网络结构文件（以 `_arch.py` 结尾）。在写好的 Class 前加上 `@ARCH_REGISTRY.register()` 装饰器
2. 在配置文件中指定使用哪一个网络结构，即上面的 Class name

4.2.2.1. 文件后缀名的约定

Model	Register	File Suffix	Example
data	DATASET_REGISTRY	_dataset.py	basicsr/data/paired_image_dataset.py
arch	ARCH_REGISTRY	_arch.py	basicsr/archs/srresnet_arch.py
model	MODEL_REGISTRY	_arch.py	basicsr/models/sr_model.py
loss	LOSS_REGISTRY	_loss.py	basicsr/losses/gan_loss.py

表 4.1: 文件后缀名的约定

注意

1. 上面的文件后缀只用在需要的文件中，其他文件命名尽量避免使用以上的后缀
2. 注册的类名或函数名不能重复，否则会报错

METRIC 稍有特殊

我们定义了 `METRIC_REGISTRY`，它在用法上和其他类一样，但是它是根据 **函数名** 来调用相对应的函数。

因为 `metric` 我们相对改动少，所以我们没有采用自动扫描文件再 `import` 的方式，而是保留了在 `basicsr/metrics/__init__.py` 中 `import` 的方式

4.2.2.2. 避免类名/函数名出现重名问题

Register 机制会自动检测出重名的类/函数，然后抛出错误：An object named xxx was already registered in yyy registry!。这是特意设计的，以减少 bug。因为如果重名的类，在实例化的时候，就不能确定程序到底实例化的是哪一个类。

但是有有一种情况下，Register 的重名检查机制反而会掣肘开发。

当我们在 BasicSR 里面定义了一个类，比如 basicsr/data/realesrgan_dataset.py 中的 RealESRGANDataset 类。而在 Real-ESRGAN GitHub repo 中，我们是把 basicsr 当作一个 package 来使用，然后又定义了一遍 RealESRGANDataset 类。这个时候，原来 BasicSR 代码中的类和后面开发的 Real-ESRGAN 代码中的类就有重名了。

这个情况下，我们约定在 BasicSR 对应的类中的注册器中，传入 basicsr 的参数，以指示这个类是 BasicSR 中定义的，以示区分。其中这里的 basicsr 是关键词，注意使用其他词是不会被识别的。

```

1 @DATASET_REGISTRY.register(suffix='basicsr') # 我们在类名中传入后缀 suffix,
    ↳ 以指示这是在 BasicSR repo 中定义的
2 class RealESRGANDataset(data.Dataset):
3     """Dataset used for Real-ESRGAN model:
4     """
5     ...

```

这个具体在 basicsr/utils/registry.py 的代码如下：

```

1 def get(self, name, suffix='basicsr'):
2     ret = self._obj_map.get(name)
3     if ret is None:
4         ret = self._obj_map.get(name + '_' + suffix)
5         print(f'Name {name} is not found, use name: {name}_{suffix}!')
6     if ret is None:
7         raise KeyError(f"No object named '{name}' found in '{self._name}'"
7             ↳ registry!")
8     return ret

```

它实现的逻辑如下：

1. 如果 get 函数指定了 suffix，则会优先找带有 suffix 的类/函数
2. 一般我们的 get 函数都不会指定 suffix。这样的情况，程序优先找自己实现的（比如 Real-ESRGAN 代码库中的）类/函数
3. 如果在自己实现的类/函数中没有找到，则会找 BasicSR 官方库中的实现的相同名字的类

§4.3. 配置(Options)

在这个章节，我们先简单介绍一下实验命名的约定 (第4.3.1.1小节)；然后通过例子介绍训练和测试的配置文件 (第4.3.2小节)。

4.3.1. 实验命名与 debug 模式

4.3.1.1. 实验命名

我们推荐对实验名字进行有意义的命名，方便后续的实验以及进行多组实验对比。

我们以 001_MSRRResNet_x4_f64b16_DIV2K_1000k_B16G1_wandb 为例：

- 001: 我们一般给实验进行数字打头的标号，方便进行实验管理
- MSRRResNet: 模型名称，这里指代 Modified SRResNet
- x4_f64b16: 重要配置参数，这里表示放大4倍；中间feature通道数是64，使用了16个Residual Block
- DIV2K: 训练数据集是 DIV2K
- 1000k: 训练了1000K iterations
- B16G1: Batch size 为16，使用一卡 GPU 训练
- wandb: 使用了 wandb，训练过程上传到了 wandb 云服务器

4.3.1.2. Debug 模式

正式训练之前，你可以用 debug 模式检查是否正常运行。在 debug 模式下：

- 程序会在每次 iteration 下都打印日志，并且经过8次 iterations 后，便会进入 validation 阶段。这样可以快速方便地查看代码是否可以正常运行，而不用实际训练。毕竟实际训练可能很慢，等半天后，发现程序崩溃了，而原因是 validation 中有 bug
- 在 debug 模式中，并不会使用 tensorboard logger 和 wandb logger，以保证日志文件的简洁性

如何进入 debug 模式？

方式 1. 在命令行最后加入 ‘**--debug**’。比如：

```
1  python basicsr/train.py -opt
   ↳ options/train/SRResNet_SRGAN/train_MSRRResNet_x4.yml --debug
```

方式 2. 在配置文件 yml 文件的 name 中添加 ‘**debug**’ 字符。只要在实验名字中有 ‘**debug**’ 字样，则会进入 debug 模式。

4.3.2. 配置文件简要说明

在 BasicSR 中我们使用 YAML 来作为配置文件的语言。

训练的配置文件在 options/train 中，测试的配置文件在 options/test 中。通过 option 配置文件，我们可以设置实验名、选择模型、指定 GPU、指定数据路径、选择网络结构、配置训练策略等。我们在第4.3.2.1小节介绍训练配置文件例子，在第4.3.2.2小节介绍测试配置文件例子。

配置文件的解析在 basicsr/utils/options.py 的 parse_options 中实现。这个过程将 YAML 文本解析成 Python 的 dict 类型，并根据需要做出调整（比如 debug 模式下的特殊配置等）。读者可以在章节3.2.1：入门 和对应代码中找到解释和具体实现。

4.3.2.1. 训练配置文件例子

下面，我们以 train_MSResNet_x4.yml 为例，简单说明训练配置文件的每个部分。我们先把配置文件贴出来，在后面对应上解释。然后在说明框内会列举相关的要点。为方便说明，整个配置文件会被分散成不同的板块来讲解。

```
# Modified SRResNet w/o BN from:  
# Photo-Realistic Single Image Super-Resolution Using a Generative  
→ Adversarial Network  
  
# ----- Commands for running  
# ----- Single GPU with auto_resume  
# PYTHONPATH=".:$PYTHONPATH" CUDA_VISIBLE_DEVICES=0 python  
→ basicsr/train.py -opt options/train/SRResNet_SRGAN/train_MSResNet_x4.yml  
→ --auto_resume  
  
# general settings - 这块为通用设置  
name: 001_MSResNet_x4_f64b16_DIV2K_1000k_B16G1_wandb # 实验名称，若实验名字  
→ 中有debug字样，则会进入debug模式  
model_type: SRModel # 使用的 model 类型  
scale: 4 # 输出比输入的倍数，在SR中是放大倍数；若有些任务没有这个配置，则写1  
num_gpu: 1 # 指定使用的 GPU 卡数  
manual_seed: 0 # 指定随机种子
```

通用配置：

- 在配置文件的最开始，会有简单的说明，以及默认的运行命令。运行命令的 `auto_resume` 表示自动从断点接着训练（详见章节4.5.4.1：代码主体结构）
- 常见模型 (Model) 的定义在 `models` 目录中。详细说明参见章节4.5：代码主体结构
- `num_gpu: 0` 表示 使用CPU，`auto` 表示自动从可用 GPU 块数推断

```
# dataset and data loader settings  
datasets: # 这块是 dataset 的配置
```

```
train: # 训练 dataset 的配置
  name: DIV2K # 自定义的数据集名称
  type: PairedImageDataset # 读取数据的 Dataset 类
  # 以下属性是灵活的，可在相应类的说明文档中获得。新加的数据集可根据需要添加
  dataroot_gt: datasets/DF2K/DIV2K_train_HR_sub # GT (Ground-Truth) 图像
  ↳ 的文件夹路径
  dataroot_lq: datasets/DF2K/DIV2K_train_LR_bicubic_X4_sub # LQ
  ↳ (Low-Quality) 输入图像的文件夹路径
  meta_info_file: basicsr/data/meta_info/meta_info_DIV2K800sub_GT.txt # 预
  ↳ 先生成的 meta_info 文件
  # (for lmdb)
  # dataroot_gt: datasets/DIV2K/DIV2K_train_HR_sub.lmdb
  # dataroot_lq: datasets/DIV2K/DIV2K_train_LR_bicubic_X4_sub.lmdb
  filename_tmpl: '{\}' # 文件名字模板，一般LQ文件会有类似 '_x4' 这样的文件后
  ↳ 缀，这个就是来处理GT和LQ文件后缀不匹配的问题的
  io_backend: # IO 读取的 backend
    type: disk # disk 表示直接从硬盘读取
    # (for lmdb)
    # type: lmdb

  gt_size: 128 # 训练阶段裁剪 (crop) 的GT图像的尺寸大小，即训练的 label 大小
  use_hflip: true # 是否开启水平方向图像增强 (随机水平翻转图像)
  use_rot: true # 是否开启旋转图像增强 (随机旋转图像)

  # data loader - 下面这块是 data loader 的设置
  num_worker_per_gpu: 6 # 每一个 GPU 的 data loader 读取进程数目
  batch_size_per_gpu: 16 # 每块 GPU 上的 batch size
  dataset_enlarge_ratio: 100 # 放大 dataset 的长度倍数 (默认为1)。可以扩大
  ↳ 一个 epoch 所需 iterations
  prefetch_mode: ~ # 预先读取数据的方式
```

数据读取相关配置：

- 常见数据 (dataset) 的定义在 `basicsr/data` 目录中。详细说明参见章节4.4: 代码主体结构
- data loader 定义在 `basicsr/data/__init__.py` 文件中
- meta_info_file: 细节请参看章节7.3: 数据准备
- io_backend: 读取数据的方式，细节请参看章节7.1: 数据准备
- dataset_enlarge_ratio: 它代表了手工扩大 dataset 的倍率。例如，如果训练数据集有15张图，设置 dataset_enlarge_ratio 为100，那么程序会重复读取这些图片100次，这样一个 epoch 下来，便会读取1500张图 (事实上是重复读的)。这个方法经常用来加速 data loader，因为在有的机器上，一个 epoch 结束，会重启进程，导致拖慢训练
- prefetch_mode:，默认为 **None**，即 ~。cpu 表示使用 CPU prefetcher。cuda 表示使用 CUDA prefetcher。它会多占用一些GPU显存。注意：这个模式下，一定要设置 `pin_memory=True`。详情参见章节4.4.4: 代码主体结构

```

val: # validation 数据集的设置
    name: Set5 # 数据集名称
    type: PairedImageDataset # 数据集的类型
    # 以下属性是灵活的，类似训练数据集
    dataroot_gt: datasets/Set5/GTmod12
    dataroot_lq: datasets/Set5/LRbicx4
    io_backend:
        type: disk

val_2: # 另外一个 validation 数据集
    name: Set14
    type: PairedImageDataset
    dataroot_gt: datasets/Set14/GTmod12
    dataroot_lq: datasets/Set14/LRbicx4
    io_backend:
        type: disk

```

validation 配置：

- 这里使用了两个 validation sets，它们通过关键字 `val`, `val_2` 来区分。如果有更多的 validation sets，可以通过 `val_3`, `val_4` ... 来区分

```

# network structures - 网络结构的设置
network_g: # 网络 g 的设置
    type: MSRResNet # 网络结构 (Architecture) 的类型
    # 以下属性是灵活且特定的，可在相应类的说明文档中获得
    num_in_ch: 3 # 模型输入的图像通道数

```

```

num_out_ch: 3 # 模型输出的图像通道数
num_feat: 64 # 模型内部的 feature map 通道数
num_block: 16 # 模型内部基础模块的堆叠数
upscale: 4 # 上采样倍数

```

网络结构相关配置：

- 常见网络结构 (arch) 的定义在 `archs` 目录下。详细说明参见章节4.6：代码主体结构
- 如果模型需要使用多个网络，我们一般以 `network_` 打头来命名。比如我们需要一个 discriminator，命名为 `network_d`

```

# path
path: # 以下为路径和与训练模型、重启训练的设置
pretrain_network_g: ~ # 预训练模型的路径，需要以 pth 结尾的模型
param_key_g: params # 读取的预训练的参数 key。若需要使用 EMA 模型，需要改成
#<-- params_ema
strict_load_g: true # 是否严格地根据参数名称一一对应 load 模型参数。如果选择
#<-- false，那么模型对于找不到的参数，会随机初始化；如果选择 true，假如存在不对
#<-- 应的参数，会报错提示
resume_state: ~ # 重启训练的 state 路径，在
#<-- experiments/exp_name/training_states 目录下

```

模型路径相关配置：

- `resume_state` 设置后，会覆盖 `pretrain_network_g` 的设定
- 对于 `resume`，更多信息可以参考章节4.5.4.1：代码主体结构

```

# training settings
train: # 这块是训练策略相关的配置
    ema_decay: 0.999 # EMA 更新权重
    optim_g: # 这块是优化器的配置
        type: Adam # 选择优化器类型，例如 Adam
        # 以下属性是灵活的，根据不同优化器有不同的设置
        lr: !!float 2e-4 # 初始学习率
        weight_decay: 0 # 权重衰退参数
        betas: [0.9, 0.99] # Adam 优化器的 beta1 和 beta2

    scheduler: # 这块是学习率调度器的配置
        type: CosineAnnealingRestartLR # 选择学习率更新策略
        # 以下属性是灵活的，根据学习率 Scheduler 的不同有不同的设置
        periods: [250000, 250000, 250000, 250000] # Cosine Annealing 的更新周期
        restart_weights: [1, 1, 1, 1] # Cosine Annealing 每次 Restart 的权重
        eta_min: !!float 1e-7 # 学习率衰退到的最小值

```

```

total_iter: 1000000 # 总共进行的训练迭代次数
warmup_iter: -1 # warm up 的迭代次数，如是-1，表示没有 warm up

# losses - 这块是损失函数的设置
pixel_opt: # loss 名字，这里表示 pixel-wise loss 的 options
    type: L1Loss # 选择 loss 函数，例如 L1Loss
    # 以下属性是灵活的，根据不同损失函数有不同的设置
    loss_weight: 1.0 # 指定 loss 的权重
    reduction: mean # loss reduction 方式

```

训练策略相关配置：

训练策略相关的配置主要分为优化器，学习率调度器，总共训练 iterations，损失函数等。

- 关于 EMA，请参考章节4.5.6：代码主体结构
- optim_g，后缀 _g 表示和 network_g 中的 _g 一一对应起来
- lr: !!float 2e-4 中的 !!float 是 YAML 语言语法，表示以 float 解释后面的数字，不然就会以文字来进行解释
- 常见优化器 (optimizer) 的定义可以在 models/base_model.py 文件中的 get_optimizer 函数中找到
- 学习率的调度策略在 models/base_model.py 文件中的 setup_schedulers 函数中定义
- 常用的损失函数可以在 losses 目录中定义

```

# validation settings
val: # 这块是 validation 的配置
    val_freq: !!float 5e3 # validation 频率，每隔 5000 iterations 做一次
    #> validation
    save_img: false # 否需要在 validation 的时候保存图片

    metrics: # 这块是 validation 中使用的指标的配置
        psnr: # metric 名字，这个名字可以是任意的
            type: calculate_psnr # 选择指标类型
            # 以下属性是灵活的，根据不同 metric 有不同的设置
            crop_border: 4 # 计算指标时 crop 图像边界像素范围 (不纳入计算范围)
            test_y_channel: false # 是否转成在 Y(CbCr) 空间上计算
            better: higher # 该指标是越高越好，还是越低越好。选择 higher 或者
            #> lower, 默认为 higher
        niqe: # 这是在 validation 中使用的另外一个指标
            type: calculate_niqe

```

```

crop_border: 4
better: lower # the lower, the better

```

validation 相关配置：

- 关于 metrics 的介绍, 请参考章节5: 指标
- 指标在 basicsr/metrics 目录中定义
- BasicSR 支持在 validation 时使用多个指标, 只需要在配置文件中添加配置, 比如上面的 psnr 和 niqe

```

# logging settings
logger: # 这块是 logging 的配置
  print_freq: 100 # 多少次迭代打印一次训练信息
  save_checkpoint_freq: !!float 5e3 # 多少次迭代保存一次模型权重和训练状态
  use_tb_logger: true # 是否使用 tensorboard logger
  wandb: # 是否使用 wandb logger
    project: ~ # wandb 的 project名字。默认是 None, 即不使用 wandb
    resume_id: ~ # 如果是 resume, 可以输入上次的 wandb id, 则 log 可以接起来

```

训练日志相关配置：

- 关于 wandb, 目前 wandb 只是同步 tensorboard 的内容, 因此要使用 wandb, 必须也同时使用 tensorboard。更多关于 wandb, 参见章节4.9.3: 代码主体结构

```

# dist training settings
dist_params: # distributed training 的设置, 目前只在 Slurm 训练下才需要
  backend: nccl
  port: 29500

```

至此, 我们对于训练的配置文件有了一个初步的理解了。

4.3.2.2. 测试配置文件例子

我们以 test_MSRResNet_x4.yml 为例, 简单说明测试配置文件的每个部分。我们先把配置文件贴出来, 在后面附上解释。然后在说明框内会列举相关的要点。为方便说明, 整个配置文件会被分散成不同的板块来讲解。由于测试配置文件和训练配置文件很类似, 我们将简略地进行讲解。

```

# ----- Commands for running
# ----- Single GPU
# PYTHONPATH=".:$PYTHONPATH" CUDA_VISIBLE_DEVICES=0 python
↪ basicsr/test.py -opt options/test/SRResNet_SRGAN/test_MSResNet_x4.yml

```

```
# general settings
name: 001_MSResNet_x4_f64b16_DIV2K_1000k_B16G1_wandb # 实验名称
model_type: SRModel # 使用的 model 类型
scale: 4 # 输出比输入的倍数，在SR中是放大倍数；若有些任务没有这个配置，则写1
num_gpu: 1 # 测试卡数
manual_seed: 0 # 指定随机种子

# test dataset settings
datasets:
  test_1: # 测试数据集的设置，后缀1表示第一个测试集
    name: Set5 # 数据集的名称
    type: PairedImageDataset # 读取数据的 Dataset 类
    # GT 和输入 LQ 的根目录
    dataroot_gt: datasets/Set5/GTmod12
    dataroot_lq: datasets/Set5/LRbicx4
    io_backend: # IO 读取的 backend
      type: disk # disk 表示直接从硬盘读取
  test_2: # 测试数据集的设置，后缀2表示第二个测试集
    name: Set14
    type: PairedImageDataset
    dataroot_gt: datasets/Set14/GTmod12
    dataroot_lq: datasets/Set14/LRbicx4
    io_backend:
      type: disk

# network structures - 网络结构的设置
network_g: # 网络 g 的设置
  type: MSResNet # 网络结构 (Architecture) 的类型
  # 以下是 MSResNet 的参数设置
  num_in_ch: 3
  num_out_ch: 3
  num_feat: 64
  num_block: 16
  upscale: 4

# path
path:
  pretrain_network_g: experiments/001_..._wandb/models/net_g_1000000.pth # 
  ↳ 预训练模型的路径，需要以 pth 结尾的模型
  param_key_g: params # 读取的预训练的参数 key。若需要使用 EMA 模型，需要改成
  ↳ params_ema
  strict_load_g: true # 加载预训练模型时，是否需要网络参数的名称严格对应

# validation settings - 以下为 Validation (也是测试)的设置
val:
```

```
save_img: true # 是否需要在测试的时候保存图片
suffix: ~ # 对保存的图片添加后缀, 如果是 None, 则使用 exp name

metrics: # 测试时候使用的 metric
psnr: # metric 名字, 这个名字可以是任意的
  type: calculate_psnr # 选择指标类型
  # 以下属性是灵活的, 根据不同 metric 有不同的设置
  crop_border: 4 # 计算指标时 crop 图像边界像素范围 (不纳入计算范围)
  test_y_channel: false # 是否转成在 Y(CbCr) 空间上计算
  better: higher # the higher, the better. Default: higher
ssim: # 另外一个指标
  type: calculate_ssim
  crop_border: 4
  test_y_channel: false
  better: higher
```

注意:

- 如果模型训练的时候开启了 EMA, 则在测试的时候需要指定 param_key_g 为 params_ema。否则会出现测试和训练过程中 validation 不匹配的问题

至此, 我们对于测试的配置文件有了一个初步的理解了。

4.3.3. 命令行修改配置

BasicSR 使用 yml 文件进行配置。我们也推荐这样的方式, 因为这样可以记录并跟踪每一个实验的配置。但我们也希望在仅仅修改了一个小配置的情况下(比如修改 random seed), 不需要麻烦地新建并修改 yml 配置文件。它可以用原先的 yml 配置文件, 而在命令行中对配置做出修改。

BasicSR 提供了一个简便的命令行参数 ‘**--force_yml**’，在命令行中它的用法如下：

- ‘--force_yml’ 后面接如下的字符串, 每一个字符串修改一个配置, 如果有多个配置需要修改, 以空格隔开多个字符串
- 字符串采用格式: ‘train:ema_decay=0.999’。等号 (=) 前后分别表示 key 和 value。如果有层级结构, 使用冒号 (:) 来区分
- 修改之后记得检查打印的日志是否符合预期

示例:

有如下的配置 yml 文件, 我们希望修改: 1) random seed 为 1; 2) ema_decay 为 0.5; 3) 名字中体现配置。

```
1 # general settings
2 name: 001_MSRAutoEncoder_x4_f64b16_DIV2K_1000k_B16G1_wandb
3 model_type: SRModel
4 scale: 4
5 manual_seed: 0
6
7 ...
8
9 train:
10 ema_decay: 0.999
```

那么，我们的命令行就变成了：

```
1 python basicsr/train.py -opt
→ options/train/SRResNet_SRGAN/train_MSRAutoEncoder_x4.yml --force_yml
→ manual_seed=1 train:ema_decay=0.5
→ name=001_MSRAutoEncoder_x4_f64b16_DIV2K_1000k_B16G1_rand1_ema_decay0.5
```

§4.4. 数据 (Data Loader 和 Dataset)

这一小节我们主要介绍 `basicsr/data` 目录下相关的功能和函数。

在模型训练的过程中，我们需要不断地喂给网络模型数据。这个过程是通过 data loader 实现的。而每个 data loader 中又会把硬盘上的数据处理成训练所需的格式，这个过程是由不同的数据集决定的，PyTorch 里面叫做 `dataset`。我们一般说的 data loader，其实更多的是指 dataset 处理数据的细节。

1. dataloader: 开启多个线程读取并处理数据，定义在 `basicsr/data/_init_.py` 中
2. dataset: 这是 dataloader 调用的，它把数据（比如 PNG 图像）转成模型、网络能够接收的输入（往往是 PyTorch Tensor 类型），它涉及到的流程：
 - a) 读取数据
 - b) 对数据做变换 transforms。比如 crop，数据增强等
 - c) 转换成 PyTorch Tensor

4.4.1. basic/data 目录介绍

basicsr/data 下面主要的文件有:

```
basicsr
└── data
    ├── meta_info.py ..... 存放 meta txt 文件
    ├── __init__.py ..... build_dataset 和 build_dataloader
    ├── data_sampler.py ..... distributed training 时的采样策略
    ├── data_util.py ..... 提供了数据读取的工具函数, 比如读取文件路径等
    ├── transforms.py ..... 提供常用的数据变换、数据增强函数
    ├── degradations.py ..... 提供一些图像退化的合成函数
    ├── prefetch_dataloader.py ..... data prefecth文件, 详见第4.4.4小节
    ├── paired_image_dataset.py .. 下面是针对不同数据集的读取文件, 我们主要也是修改它们。它们都以 _dataset.py 结尾
    ├── reds_dataset.py
    ├── realesrgan_dataset.py
    ├── vimeo90k_dataset.py
    ├── ffhq_dataset.py
    └── ...
```

meta_info 的 txt 文件是可以根据自己的需要创建的。BasicSR 提供了一些约定的 meta_info 文件, 具体参见章节7.3: 数据准备。

💡 我们新建数据集的 dataset 的时候, 需要以 _dataset.py 结尾, 这样才能够被程序自动扫描 import。

BasicSR 提供的常用数据集的 Dataset 文件如表4.2。注意这里只列了一些基本的, 更多的 dataset 请参见代码或者在线 API 文档 <https://basicsr.readthedocs.io/en/latest/>。

类	任务	训练/测试	描述
PairedImageDataset	图像超分	训练	读取成对的训练数据
SingleImageDataset	图像超分	测试	只读取 low quality 的图像, 用于没有 GT 的测试中
REDS Dataset	视频超分	训练	读取 REDS 的训练数据集
Vimeo90K Dataset	视频超分	训练	读取 Vimeo90K 的训练数据集
VideoTestDataset	视频超分	测试	基础的视频超分测试集, 支持 Vid4, REDS 测试集
VideoTestVimeo90K Dataset	视频超分	测试	继承 VideoTestDataset; Vimeo90K 的测试数据集
VideoTestDUF Dataset	视频超分	测试	继承 VideoTestDataset; DUF 算法的测试数据集, 支持 Vid4
FFHQ Dataset	人脸生成	训练	读取 FFHQ 的训练数据集

表 4.2: BasicSR 提供的基本 dataset

4.4.2. Data loader 和 Dataset 的创建

dataloader 的创建是在 `basicr/data/__init__.py` 文件中的 `build_dataloader` 函数。这里不赘述, 请参见代码。

dataset 的创建是通过 Register 机制实现的。具体可以参考:

1. 章节[4.2: 代码主体结构](#) 说明了 Register 机制是如何根据配置文件中的类型, 来自动实例化类的
2. 章节[3.2.2: 入门](#) 通过一个完整的例子, 说明了其中 dataset 的创建过程

4.4.3. Dataset 示例讲解

下面, 我们以 `PairedImageDataset` 为例, 大致讲解 Dataset 文件的内容。`PairedImageDataset` 常被用在图像复原(超分辨率、去噪、去模糊等)任务中。它会从两个目录中读取成对的训练数据对。

```

1  @DATASET_REGISTRY.register()
2  class PairedImageDataset(data.Dataset):
3      def __init__(self, opt): # 这里是初始化函数
4          super(PairedImageDataset, self).__init__()
5          self.opt = opt
6          # file client (io backend) 这是初始化 file client 部分
7          self.file_client = None
8          self.io_backend_opt = opt['io_backend']
9          # 赋值常用的配置参数
10         self.mean = opt['mean'] if 'mean' in opt else None
11         self.std = opt['std'] if 'std' in opt else None

```

数据读取格式: 在 option 文件中可以指定数据读取方式 (`opt['io_backend']`)。

我们支持三种读取数据的模式:

1. 直接从 lmdb 格式的文件中读取
2. 若提供了 meta_info 文件, 则直接从该文件中列出的文件路径读取数据
3. 输入文件目录, 代码会自动扫描该目录中的文件, 然后读取

详见参见 File Client 说明章节[7.4: 数据准备](#)。

```

1  # 这块内容是根据 GT 和 LQ 的图像目录读取出相应的文件列表
2      self.gt_folder, self.lq_folder = opt['dataroot_gt'],
3          ↳ opt['dataroot_lq']
4      if 'filename_tmpl' in opt:
5          self.filename_tmpl = opt['filename_tmpl']

```

```

6         self.filename_tmpl = '{}'
7
8     # 如果输入是 lmdb, 则使用 paired_paths_from_lmdb 函数
9     if self.io_backend_opt['type'] == 'lmdb':
10         self.io_backend_opt['db_paths'] = [self.lq_folder,
11             ↪ self.gt_folder]
12         self.io_backend_opt['client_keys'] = ['lq', 'gt']
13         self.paths = paired_paths_from_lmdb([self.lq_folder,
14             ↪ self.gt_folder], ['lq', 'gt'])
15     # 如果输入是 meta_info_file 方式, 则使用
16     ↪ paired_paths_from_meta_info_file 函数
17     elif 'meta_info_file' in self.opt and self.opt['meta_info_file'] is
18         ↪ not None:
19         self.paths = paired_paths_from_meta_info_file([self.lq_folder,
20             ↪ self.gt_folder], ['lq', 'gt'], self.opt['meta_info_file'],
21             ↪ self.filename_tmpl)
22     # 如果是一般文件目录方式, 则使用 paired_paths_from_folder 函数
23     else:
24         self.paths = paired_paths_from_folder([self.lq_folder,
25             ↪ self.gt_folder], ['lq', 'gt'], self.filename_tmpl)
26     # 以上这些函数都已经实现在 basicsr/data/data_util.py 文件中

```

下面是一个 dataset 中重要的函数 `__getitem__`, 它定义了从输入图像, 经过变换、数据增强等变为 PyTorch Tensor 的过程。

```

1 def __getitem__(self, index):
2     # 初始化 file client
3     if self.file_client is None:
4         self.file_client = FileClient(self.io_backend_opt.pop('type'),
5             ↪ **self.io_backend_opt)
6
7     scale = self.opt['scale']
8
9     # 下面这个代码块是从存储介质中读取相应的数据到内存的过程
10    # Load gt and lq images. Dimension order: HWC; channel order: BGR;
11    # image range: [0, 1], float32.
12    gt_path = self.paths[index]['gt_path']
13    img_bytes = self.file_client.get(gt_path, 'gt')
14    img_gt = imfrombytes(img_bytes, float32=True)
15    lq_path = self.paths[index]['lq_path']
16    img_bytes = self.file_client.get(lq_path, 'lq')
17    img_lq = imfrombytes(img_bytes, float32=True)
18
19    # 下面这个代码块是做数据增强, 在这里主要是成对数据的随机裁剪和旋转、翻
20    ↪ 转等

```

```

19         # augmentation for training
20     if self.opt['phase'] == 'train':
21         gt_size = self.opt['gt_size']
22         # random crop
23         img_gt, img_lq = paired_random_crop(img_gt, img_lq, gt_size,
24                                             scale, gt_path)
25         # flip, rotation
26         img_gt, img_lq = augment([img_gt, img_lq], self.opt['use_hflip'],
27                                   self.opt['use_rot'])
28
29         # 如果有需要的花，会做色彩空间转换
30         # color space transform
31         if 'color' in self.opt and self.opt['color'] == 'y':
32             img_gt = rgb2ycbcbgr2ycbrr(img_gt, y_only=True)[..., None]
33             img_lq = bgr2ycbcr(img_lq, y_only=True)[..., None]
34
35         # 以下代码块将 numpy 数据格式转换成 PyTorch 所需的 Tensor 格式，并根据
36         # 需要作归一化
37         # BGR to RGB, HWC to CHW, numpy to tensor
38         img_gt, img_lq = img2tensor([img_gt, img_lq], bgr2rgb=True,
39                                     float32=True)
40
41         # normalize
42         if self.mean is not None or self.std is not None:
43             normalize(img_lq, self.mean, self.std, inplace=True)
44             normalize(img_gt, self.mean, self.std, inplace=True)
45
46         # 最后，我们返回一个字典，包括输入的 LQ 图像，作为标签的 GT 图像，以及他
47         # 们的路径
48         return {'lq': img_lq, 'gt': img_gt, 'lq_path': lq_path, 'gt_path':
49                 gt_path}

```

4.4.4. Dataset prefetch 说明

下面我们介绍 **数据预读取 prefetch** 机制。为了加速数据读取的进程，我们提供了数据预读取机制，具体的实现代码详见 `data/prefetch_dataloader.py`。

目前我们提供了三种数据预读取方式，可以在训练配置文件中进行设置。

1. `None` 模式。默认不使用。如果使用了 LMDB 或者 IO 开销不大，可以不使用 `prefetch`。

```

prefetch_mode: ~

```

2. `cuda` 模式。使用 CUDA prefetcher，具体介绍可以参考 [NVIDIA/apex](#)。这个模式会多占用一些 GPU 显存。注意，如果使用这个模型，一定要设置 `pin_memory=True`。

```

prefetch_mode: cuda
pin_memory: true

```

3. cpu 模式。使用 CPU prefetcher，具体介绍可以参考 [IgorSusmelj/pytorch-styleguide](#)。这个加速效果可能不明显。

```

prefetch_mode: cpu
num_prefetch_queue: 1  # 1 by default

```

 如果训练过程中 IO 读取慢，推荐使用 LMDB 数据格式加速，详见章节[7.2.1: 数据准备](#)。

§4.5. 模型 (Model)

4.5.1. basic/models 目录介绍

`basicsr/models` 下面主要的文件有：

```

basicsr
└ models
    ├── __init__.py ..... 定义 build_model, 扫描所有 models 并注册。
    ├── base_model.py ..... 模型的基类, 定义模型的共用操作。
    ├── lr_scheduler.py ..... 自定义的学习率策略
    ├── sr_model.py ..... 下面是不同的模型, 我们主要也是修改它们, 它们都以 _model.py 结尾。
        └ sr_model 是用于超分的模型。
    ├── srgan_model.py
    ├── realesrgan_model.py
    ├── edvr_model.py
    ├── video_base_model.py ..... 视频超分基础模型
    ├── video_gan_model.py
    └ ...

```

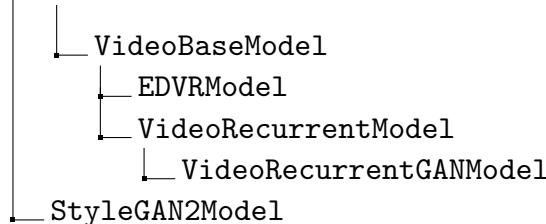
 我们新建 model 的时候，需要以 `_model.py` 结尾，这样才能够被程序自动扫描 import。

为增加模型间的复用，很多模型都是继承的，以下为主要模型的继承关系。通过继承，可以精简代码开发，复用功能函数。注意这里只列了一些基本的，更多的 model 请参见代码或者在线 API 文档 <https://basicsr.readthedocs.io/en/latest/>。

```

BaseModel ..... 模型基类
└ SRModel
    └ SRGANModel
        └ ESRGANModel
            └ RealESRGANModel
    └ RealESRNetModel
    └ SwinIRModel

```



4.5.2. Model 的创建

model 的创建是通过 Register 机制实现的。具体可以参考：

1. 章节[4.2: 代码主体结构](#) 说明了 Register 机制是如何根据配置文件中的类型，来自动实例化类的
2. 章节[3.2.2: 入门](#) 通过一个完整的例子，说明了其中 model 的创建过程

model 创建完后，网络结构、损失函数等都是在 model 的初始化过程中创建的。章节[3.2.2: 入门](#) 也提供了一个概览。

4.5.3. Base Model 和 Model 示例讲解

4.5.3.1. Base Model

Base Model 是所有模型的基类，定义一些共同操作。这里做一个简要的介绍。

```
1  class BaseModel():
2      def __init__(self, opt):
3          # 初始化
4          ...
5      def feed_data(self, data):
6          # 喂数据，需要在继承的类里面具体实现
7          pass
8      def optimize_parameters(self):
9          # 优化参数，这里特指一次完整的训练过程，即 train_step
10         pass
11     def save(self, epoch, current_iter):
12         # 保存训练模型和训练状态
13         pass
14     def validation(self, dataloader, current_iter, tb_logger,
15                   save_img=False):
16         # validation 函数
17         ...
18     def model_ema(self, decay=0.999):
19         # 进行模型 EMA
20         ...
21     def model_to_device(self, net):
22         # 将模型放到 GPU 上
```

```
22     ...
23     def get_optimizer(self, optim_type, params, lr, **kwargs):
24         # 根据 yml 配置文件获取优化器
25
26     def setup_schedulers(self):
27         # 根据 yml 配置文件获取学习率的策略方式
28
29     ...
30     @master_only
31     def print_network(self, net):
32         # 打印网络，包括总参数
33
34     ...
35     def update_learning_rate(self, current_iter, warmup_iter=-1):
36         # 更新学习率
37
38     ...
39     @master_only
40     def save_network(self, net, net_label, current_iter, param_key='params'):
41         # 保存网络参数
42
43     ...
44     def load_network(self, net, load_path, strict=True, param_key='params'):
45         # 加载网络参数
46
47     ...
48     @master_only
49     def save_training_state(self, epoch, current_iter):
50         # 保存网络训练状态
51
52     ...
53     def resume_training(self, resume_state):
54         # 断点恢复训练
55
56     ...
57     def reduce_loss_dict(self, loss_dict):
58         # 在多卡训练时，平均多个 GPU 的损失函数
59
60     ...
```

说明:

- 对于没有实现的函数 (内容是 `pass`)，需要在继承的类中实现函数的功能，即它们是必须要重新实现的
- `@master_only` 表示在多卡下，只在主卡上进行调用

4.5.3.2. SRModel 简单说明

我们在这里简单说明 `SRModel` 类，它是图像超分辨率模型的基础类，定义了基础的单张图像超分辨率模型。下面的代码主要为了说明核心流程，代码不一定完整。

```
1  class SRModel(BaseModel):
2      """Base SR model for single image super-resolution."""
3
4      def __init__(self, opt):
5          # 初始化，主要包括以下几块内容：
6
7          # 定义网络结构，根据配置文件，自动实例化相应的网络结构类
8          self.net_g = build_network(opt['network_g'])
9          self.net_g = self.model_to_device(self.net_g)    # 将网络放到 GPU 上
10         self.print_network(self.net_g)    # 打印网络
11
12         # 加载预训练网络
13         load_path = self.opt['path'].get('pretrain_network_g', None)
14         if load_path is not None:
15             param_key = self.opt['path'].get('param_key_g', 'params')
16             self.load_network(self.net_g, load_path,
17                               self.opt['path'].get('strict_load_g', True), param_key)
18
19         # 初始化训练的设置
20         if self.is_train:
21             self.init_training_settings()
22
23     def init_training_settings(self):
24         # 初始化训练设置。包括优化器、损失函数的定义，学习率的初始化等
25         self.net_g.train()
26         ...
27         if self.ema_decay > 0:
28             # 如果设置了模型 EMA，则设置 EMA 模型和参数
29             ...
30
31             # 定以损失函数，通过 build_loss，根据配置文件，实例化相应的损失函数
32             self.cri_pix = build_loss(train_opt['pixel_opt']).to(self.device)
33             self.cri_perceptual =
34                 build_loss(train_opt['perceptual_opt']).to(self.device)
35
36             # 设置优化器、学习率策略
37             self.setup_optimizers()
38             self.setup.schedulers()
39
40     def setup_optimizers(self):
41         # 设置优化器，可以控制哪些参数会被更新
```

```
40         for k, v in self.net_g.named_parameters():
41             optim_params.append(v)
42
43         optim_type = train_opt['optim_g'].pop('type')
44         self.optimizer_g = self.get_optimizer(optim_type, optim_params,
45             **train_opt['optim_g'])
46         self.optimizers.append(self.optimizer_g)
47
48     def feed_data(self, data):
49         # 把训练数据送入模型。这里是从 dataloader 中取出数据，用于训练或测试。在
50         # SRModel 中，每次取用一个 batch 的 LR 和 GT 图像
51         # 其他模型对 batch 做不同操作时，经常会改写这个函数。比如只读取 GT、读取額
52         # 外 label、对读取的数据添加 degradation 等操作，都通过修改 feed_data()
53         # 来实现
54         self.lq = data['lq'].to(self.device)
55         self.gt = data['gt'].to(self.device)
56
57     def optimize_parameters(self, current_iter):
58         # 优化参数，这里会完成一次完整的训练过程，即 train_step
59         # 将优化器梯度归零
60         self.optimizer_g.zero_grad()
61         # forward 网络
62         self.output = self.net_g(self.lq)
63
64         # 计算 loss
65         l_pix = self.cri_pix(self.output, self.gt)
66         l_total += l_pix
67         loss_dict['l_pix'] = l_pix
68         ...
69
70         # 梯度反传
71         l_total.backward()
72         # 优化器优化
73         self.optimizer_g.step()
74
75         # 同步多卡上的损失函数
76         self.log_dict = self.reduce_loss_dict(loss_dict)
77
78     def test(self):
79         # 测试函数
80         self.net_g.eval()
81         with torch.no_grad():
82             self.output = self.net_g(self.lq)
83         self.net_g.train()
```

```

81     def dist_validation(self, dataloader, current_iter, ...):
82     def nondist_validation(self, dataloader, current_iter, ...):
83         # validation (dist 表示 distributed training, 多卡训练)。这里分为了多卡和
84         ↪ 单卡的 validation
85         # 这个过程包括读取验证数据集、测试、计算指标、保存结果图等
86
87     def save(self, epoch, current_iter):
88         # 保存训练模型和训练状态

```

4.5.4. 保存模型、训练状态 和 Resume

训练的时候， checkpoints 会保存两个文件：

1. 网络参数 .pth 文件。在每个实验的 models 文件夹中，文件名诸如: net_g_5000.pth、net_g_10000.pth
2. 包含 optimizer 和 scheduler 信息的 .state 文件。在每个实验的 training_states 文件夹中，文件名诸如: 5000.state、10000.state

根据这两个文件，就可以 resume 了。

4.5.4.1. 如何 Resume?

Resume 指程序中断后，我们希望能够接着中断的地方，继续训练。只要有保存的网络参数和训练状态，我们就可以断点重训啦。在 Resume 下，实验文件夹不会被覆盖，而是会继续上次的实验文件夹继续保存文件。但是 log 文件会重新产生。

有两种方式进行 resume：

1. 手动 resume。在 yml 配置文件内中，设置 resume_state 为待 resume 的 .state 文件路径。然后重新运行训练命令。此时程序会自动查找相对应的网络参数 .pth 文件 (即不需要设置类似 pretrain_network_g 的路径)；然后进行 resume。注意：resume_state 设置后，会覆盖 pretrain_network_g 的设定
2. 自动 resume。只要在命令行中加入 '--auto_resume'，程序就会找到保存的最近的模型参数和状态，并加载进来，接着训练啦

4.5.5. 模型 validation

在框架设计的时候，我们把 validation 放到每个 model 中，作为它的成员函数。

根据单卡和多卡训练的不同，我们分别定义了 nondist_validation 和 dist_validation，对应单卡和多卡 validation。

4.5.6. EMA 介绍

EMA (Exponential Moving Average)，指数移动平均。它是用来“平均”一个变量在历史上的值。使用怎样的权重平均呢？如名字所示，随着时间，越是过往的时间，以一个指数衰减的权重来平均。

在 BasicSR 里面，EMA 一般作用在模型的参数上。它的效果一般是：

- 稳定训练效果。GAN 训练的结果一般瑕疪更少，视觉效果更好
- 对于以 PSNR 为目的的模型，其 PSNR 一般会更高一些

由于开启 EMA 的代价几乎可以不计，所以我们推荐开启 EMA。

🔔 如何开启 EMA？

在 yml 的配置文件中，我们只要指定 `ema_decay` 大于 0，就会开始 EMA。

```
1 # training settings
2 train:
3     ema_decay: 0.9 # 开启 EMA，滑动系数为 0.9
```

开启了 EMA 后，保存的模型会有两个字段：`params` 和 `params_ema`，其中 `params_ema` 就是 EMA 保存的模型。在测试或者推理的时候，我们要留意加载的到底是 `params` 还是 `params_ema`，这个在 yml 文件中，一般通过 `param_key_g: params_ema` 来指定。

§4.6. 网络结构 (Architecture)

4.6.1. basic/arch 目录介绍

在 `basicsr/archs/` 目录下，我们提供了若干经典的网络结构：

```
basicsr
└── archs
    ├── __init__.py ..... 定义 build_network，扫描所有 networks 并注册。
    ├── arch_util.py ..... 定义一些在网络结构中常用的工具函数。
    ├── edsr_arch.py ..... 下面是不同的网络结构，我们主要也是修改它们。它们都以 _arch.py 结尾。具体描述参见下面的表格。
    ├── srresnet_arch.py
    ├── rrdnet_arch.py
    └── ...
```

🔔 我们新建 arch 的时候，需要以 `_arch.py` 结尾，这样才能够被程序自动扫描 import。

网络结构	任务	文件	描述
EDSR	图像超分	edsr_arch.py	论文 EDSR 结构
SRResNet	图像超分	srresnet_arch.py	论文 SRGAN 的 Generator
RRDB	图像超分	rrdbnet_arch.py	论文 ESRGAN 的 Generator
RCAN	图像超分	rcan_arch.py	论文 RCAN 的结构
SwinIR	图像超分	swinir_arch.py	论文 SwinIR 的结构
ECBSR	图像超分	ecbsr_arch.py	论文 ECBSR 的结构
SRVGG	图像超分	srvgg_arch.py	VGG 结构改造的 SR 网络
EDVR	视频超分	edvr_arch.py	论文 EDVR 的结构
BasicVSR	视频超分	basicvsr_arch.py	论文 BasicVSR 的结构
BasicVSR++	视频超分	basicvsrpp_arch.py	论文 BasicVSR++ 的结构
DUF	视频超分	duf_arch.py	论文 DUF 的结构
TOF	视频超分	tof_arch.py	论文 TOF 的结构
DFDNet	人脸复原	dfdnet_arch.py	论文 Deep Face Dictionary Network 的结构
StyleGAN2	人脸生成	stylegan2_arch.py	论文 StyleGAN2 的结构
RIDNet	图像去噪	ridnet_arch_arch.py	论文 RIDNet 的结构
VGG	工具结构	vgg_arch.py	经典 VGG 结构
InceptionV3	工具结构	inception.py	经典 InceptionV3 的结构，用于计算 FID 指标
VGG & UNet	工具结构	discriminator_arch.py	在 GAN 训练中，常用的 Discriminator 结构
SpyNet	工具结构	spynet_arch.py	论文 SpyNet 的结构

表 4.3: BasicSR 提供的经典网络结构

4.6.2. 网络结构 arch 的创建

arch 的创建是通过 Register 机制实现的。具体可以参考：

1. 章节4.2: 代码主体结构 说明了 Register 机制是如何根据配置文件中的类型，来自动实例化类的
2. 章节3.2.2: 入门 通过一个完整的例子，说明了其中 model 的创建过程，在 model 初始化时，进行了 arch 的创建

§4.7. 损失函数 (Loss)

4.7.1. basic/losses 目录介绍

在 `basicr/losses/` 目录下，我们提供了若干常用的损失函数。

```
basicr
└── losses
    ├── __init__.py ..... 定义 build_loss, 扫描所有 loss 并注册。
    ├── loss_util.py ..... 提供损失函数中的常用工具函数。
    ├── basic_loss.py .. 下面是不同的损失函数，我们主要也是修改它们。它们都以 _loss.py
        结尾。这里主要定义 pixel loss (L1/L2), perceptual loss 等。
    ├── gan_loss.py ..... 主要定义和 GAN 训练相关的函数。
    └── ...
```

 我们新建 loss 的时候，需要以 `_loss.py` 结尾，这样才能够被程序自动扫描 import。

4.7.2. loss 的创建

loss 的创建是通过 Register 机制实现的。具体可以参考：

1. 章节4.2: 代码主体结构 说明了 Register 机制是如何根据配置文件中的类型，来自动实例化类的
2. 章节3.2.2: 入门 通过一个完整的例子，说明了其中 model 的创建过程，在 model 初始化时，进行了 loss 的创建

4.7.3. loss 在日志中的添加

loss 添加的大部分工作都是代码自动化的。在实际使用中，只需要在 `xx_model.py` 计算 loss 之后，执行 `loss_dict['新的loss'] = 新的loss`，即可同时记录在 `log` 和 `tensorboard (tb_logger)` 文件中。

以 `basicr/models/sr_models.py` 为例：

```
1  class SRModel(BaseModel):
2      ...
3      def optimize_parameters(self, current_iter):
4          ...
5          loss_dict = OrderedDict() # 使用有序字典，可以在 log 显示的时候，保持
6          # 我们添加先后的顺序
7          ...
8          loss_dict['l_pix'] = l_pix # 添加 pixel loss, 字典的 key 以 l_ 打头
9          loss_dict['l_percep'] = l_percep # 添加 perceptual loss, 字典的 key
10         # 以 l_ 打头
11         ...
```

```
10     self.log_dict = self.reduce_loss_dict(loss_dict) # 通过调用  
→    reduce_loss_dict 来同步多卡的 loss  
11     # 只要赋值到 self.log_dict, 程序即可自动进行后续的 log
```

添加非 loss 的值

有时候需要记录每个 iteration 的其他一些值 (比如 gradient norm)，我们希望可以在 log 文件的每一行记录和 tb_logger 中都有体现。如果为了简便，也可以使用以上这种方法记录不是 loss 的其他值。即把其他值也添加到 loss_dict 中。

注：其实 loss_dict 正确的含义应该是 log_dict。但由于历史原因，被叫做 loss_dict，然后为了后面的兼容，我们就不改名字了。

log 命名的约定：在 log 的时候，loss 项使用 l_ 开头，这样在 Tensorboard 显示的时候，所有 loss 会被组织到一起。比如在 basicsr/models/srgan_model.py 中，使用了 l_g_pix, l_g_percep, l_g_gan 等。在 basicsr/utils/logger.py 中，他们会被组织到一起：

```
if k.startswith('l_'):  
    self.tb_logger.add_scalar(f'losses/{k}', v, current_iter)  
else:  
    self.tb_logger.add_scalar(k, v, current_iter)
```

§4.8. 算子 (Ops)

4.8.1. 什么是算子？

当使用 PyTorch 时，绝大多数操作为张量 (Tensor) 的运算。张量计算的种类有很多，比如加法、乘法、矩阵相乘、矩阵转置等，这些计算被称为算子 (Operator)，它们是PyTorch的核心组件。

有时候出于一些其他方面的考虑，会需要增加底层算子。例如有时候对性能要求很高，Python 不满足需求，因此 PyTorch 也提供了直接扩展底层C++算子的能力。

4.8.2. BasicSR 中的自定义算子

BasicSR 中所用的自定义算子代码在 `BasicSR/basicsr/ops` 中。采用 C++ extension 的方式添加。它与 PyTorch 相互解耦，分开编译。它的原理其实就是在通过 pybind11，将 C++ 编译为 PyTorch 的一个模块，这样就可以在 PyTorch 中通过这个新的模块来执行新的操作了。添加方法详情参阅 [pytorch官方文档](#)。

BasicSR 中的自定义算子主要有：

- 可变形卷积 DCN: `deform_conv`。它在 BasicSR 里面主要用来做隐式对齐，比如用在 EDVR 中。注意：如果安装的 Torchvision 版本 $\geq 0.9.0$ ，会自动使用 TorchVision 中提供的 DCN，故不需要安装此编译算子
- StyleGAN2 中的特定的算子，比如：`upfirdn2d`, `fused_act`

4.8.3. BasicSR 中算子的编译、安装、使用

编译和安装的过程参见章节[4.8.3: 代码主体结构](#)。

当算子成功编译之后，调用自定义算子就像调用 PyTorch 原生算子一样。

```
1 # 调用 PyTorch 原生算子 nn
2 from torch import nn
3
4 Class xxx():
5     def __init__():
6         conv1 = nn.Conv2d(...)
7
8     def forward():
9         out = conv1(...)
10
11 # 调用自定义算子，可以直接在 forward 中使用
12 from basicsr.ops.upfirdn2d import upfirdn2d
13
14 Class xxx():
15     def forward():
16         out = upfirdn2d(...)
```

§4.9. 日志系统 (Logger)

BasicSR 的日志系统主要包括：

1. 记录的 log 文件
2. 通过 tensorboard 可视化的 tb_logger
3. wandb

日志系统的实现不过多赘述，如果有兴趣可以参阅 [BasicSR/basicsr/utils/logger.py](#)。本章节主要讲述日志系统各个项目代表什么，以及如何使用。

4.9.1. log 文件记录与解读

当进行实验的时候，代码会在 BasicSR/experiments 中创建一个属于当前实验的文件夹，文件夹名字为实验名，文件夹中会存在一个按照 `train_[exp_name]_[timestamp].log` 方式命名的 log 文件。下面我们来说明如何按照我们自己的要求记录 log 文件，以及现有 log 文件每个条目的意义。

🔔 添加一条 log 信息非常简单：

```
1 logger = get_root_logger() # 获得 logger
2 logger.info(要添加的内容) # 添加正常信息
3 logger.warning(要添加的警告) # 添加警告信息
```

下面是一个 log 文件的例子：

```
1 # log 文件所在位置: experiments/实验名字/train_[exp_name]_[timestamp].log
2 2022-06-17 02:27:36,068 INFO:
3 # 在代码中调用 logger.info()后，会自动记录时间并增加条目 INFO:
4 # 为节省篇幅，下面 log 中删掉了时间信息
5
6 Version Information:
7 # 软件的版本
8     BasicSR: 1.3.3.10
9     PyTorch: 1.9.1+cu111
10    TorchVision: 0.10.1+cu111
11 INFO:
12     name: 000_SRResNet_DIV2K
13 # 实验名，整个 配置option 的内容都会记录在这里，这里省略掉了
14
15
16 INFO: Dataset [XXXDataset] - XXXdata is built.
17 INFO: Training statistics:
```

2021-09-21 17:05:13,882 INFO:
 Version Information:
 BasicSR: 1.3.4.2
 PyTorch: 1.7.1
 TorchVision: 0.8.2
 2021-09-21 17:05:13,883 INFO:
 name: 001_MSRResNet_x4_f64b16_DIV2K_1000k_B16G1_wandb
 model_type: SRModel
 scale: 4
 num_gpu: 1
 manual_seed: 0
 datasets:[
 train:[
 name: DIV2K
 type: PairedImageDataset
 dataroot_gt: datasets/DF2K/DIV2K_train_HR_sub
 dataroot_lq: datasets/DF2K/DIV2K_train_LR_bicubic_X4_sub
 meta_info_file: basicsr/data/meta_info/meta_info_DIV2K800sub_GT.txt
 filename_tmpl: {}
 2021-09-21 17:05:19,061 INFO: Use Exponential Moving Average with decay: 0.999
 2021-09-21 17:05:19,118 INFO: Network [MSRResNet] is created.
 2021-09-21 17:05:19,130 INFO: Loss [L1Loss] is created.
 2021-09-21 17:05:19,131 INFO: Model [SRModel] is created.
 2021-09-21 17:05:20,019 INFO: Start training from epoch: 0, iter: 0 [epoch, iter, learning_rate] [estimated time, loss]
 2021-09-21 17:05:29,832 INFO: [001_M..] [epoch: 0, iter: 0, lr:(2.000e-04,)] [eta: 22:24:21, time (data): 0.098 (0.065)] l_pix: 2.8622e-02
 2021-09-21 17:05:39,876 INFO: [001_M..] [epoch: 0, iter: 200, lr:(2.000e-04,)] [eta: 1 day, 1:08:16, time (data): 0.099 (0.068)] l_pix: 3.2772e-02
 2021-09-21 17:05:47,419 INFO: [001_M..] [epoch: 0, iter: 300, lr:(2.000e-04,)] [eta: 23:44:32, time (data): 0.076 (0.046)] l_pix: 2.6004e-02
 2021-09-21 17:05:55,528 INFO: [001_M..] [epoch: 0, iter: 400, lr:(2.000e-04,)] [eta: 23:26:05, time (data): 0.078 (0.048)] l_pix: 1.9229e-02
 2021-09-21 17:06:03,058 INFO: [001_M..] [epoch: 0, iter: 500, lr:(2.000e-04,)] [eta: 22:55:41, time (data): 0.076 (0.046)] l_pix: 2.1077e-02
 2021-09-21 17:06:09,876 INFO: [001_M..] [epoch: 0, iter: 600, lr:(2.000e-04,)] [eta: 22:15:38, time (data): 0.072 (0.042)] l_pix: 2.4335e-02
 2021-09-21 17:06:17,291 INFO: [001_M..] [epoch: 0, iter: 700, lr:(2.000e-04,)] [eta: 22:01:09, time (data): 0.072 (0.042)] l_pix: 2.8497e-02
 2021-09-21 17:06:27,287 INFO: [001_M..] [epoch: 0, iter: 800, lr:(2.000e-04,)] [eta: 22:43:55, time (data): 0.086 (0.056)] l_pix: 4.2771e-02

图 4.2: 实验过程中产生的 log 信息

```

18 # 训练数据的信息, 数量、batchsize 等
19     Number of train images: 38684
20     Dataset enlarge ratio: 1
21     Batch size per gpu: 16
22     World size (gpu number): 1
23     Require iter number per epoch: 2418
24     Total epochs: 207; iters: 500000.
25     INFO: Dataset [PairedImageDataset] - validation is built.
26     # dataset 类型
27     INFO: Number of val images/folders in validation: 14
28     # 验证集信息
29     INFO: Network [MSRResNet] is created.
30     INFO: Network: MSRResNet, with parameters: 1,222,147
31     # 网络参数量
32     INFO: MSRResNet(
33     # 网络结构, 此处省略
34
35     INFO: Use Exponential Moving Average with decay: 0.999
36     INFO: Loss [L1Loss] is created.
37     INFO: Model [RealesRNetModel_XXX] is created.
38     INFO: Start training from epoch: 0, iter: 0
39     # 开始训练

```

```

40 INFO: [001_M..] [epoch: 0, iter: 100, lr:(2.000e-04,)] [eta: 2 days,
→ 21:55:41, time (data): 0.040 (0.004)] l_pix: 5.0581e-02
41 # [001_M..]: 实验名字，推荐使用数字打头的实验名，这样可以很方便区分实验
42 # [epoch: 第几轮, iter: 第几次迭代 (一次迭代是一个batch), lr: 学习率 (如果分组,
→ 则会显示多组)]
43 # [eta: 预估剩余时间, time (data): 一次迭代所需时间 (读取数据的时间)]
44 # l_pix: 当前的各项 loss
45
46 INFO: Validation validation
47 # 验证集验证结果
48     # psnr: 18.0289
49
50 INFO: End of training. Time consumed: 10:22:17
51 INFO: Save the latest model.
52 INFO: Validation validation
53 # 训练结束，最终验证集结果
54     # psnr: 20.7466

```

4.9.2. tensorboard logger 记录及解读

除了上述的 log 文件外，BasicSR 还会生成可以用 tensorboard 打开的 tb_logger 文件。一般保存在 BasicSR/experiments/tb_logger/实验名。

如何开启

在 yml 配置文件中设置 ‘use_tb_logger: true’:

```

1 # yml 配置文件
2     logger:
3         use_tb_logger: true

```

如何查看

在命令行输入以下命令，就可以在浏览器中查看：

```
1 tensorboard --logdir tb_logger --port 5500 --bind_all
```

下图4.3是一个示例：

如何添加

1. 被添加到 log 文件 (第4.9.1小节：[代码主体结构](#)) 中的值，会被自动添加到 tb_logger 中，比如损失函数 (第4.7.3小节：[代码主体结构](#))
2. 其他需要的 (比如 validation 时，metric的值)，可仿照下面的方式进行添加

下面是 basicr/models/sr_models.py 中 validation 添加 metrics 值到 tb_logger 的例子：

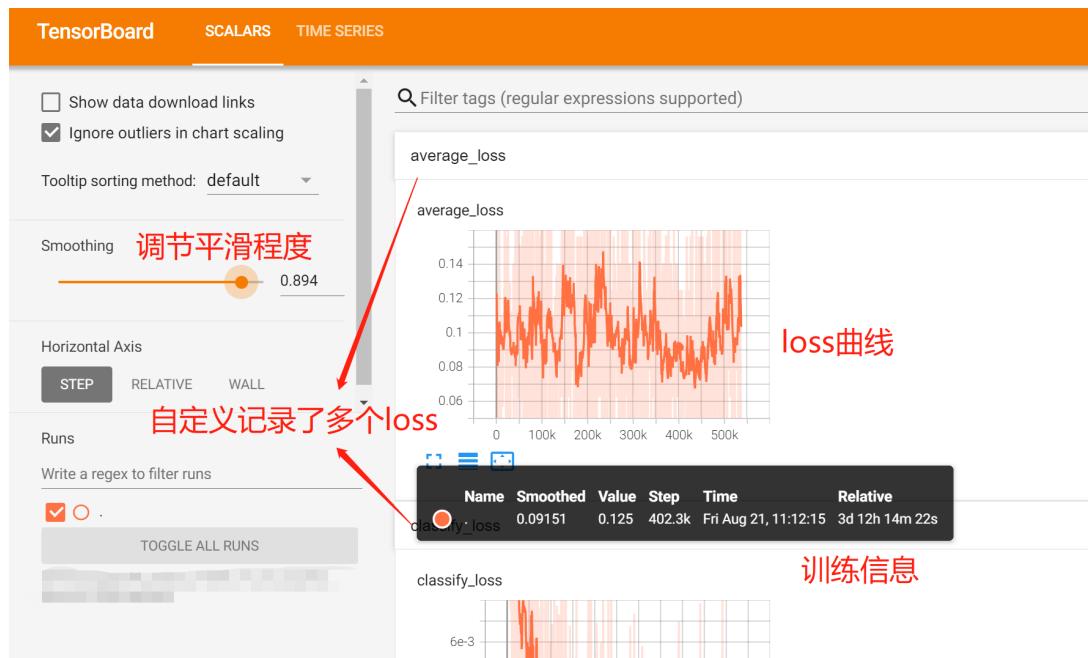


图 4.3: tensorboard示例图

```

1 def _log_validation_metric_values(self, current_iter, dataset_name,
2                                 tb_logger):
3     ...
4     for metric, value in self.metric_results.items():
5         tb_logger.add_scalar(f'metrics/{dataset_name}/{metric}', value,
6                             current_iter)

```

如果有需要添加其他图片内容，也可以根据需要添加，比如在 `basicsr/models/stylegan2_model.py` 中 validation 添加 samples 到 `tb_logger` 的例子：

```

1 def nondist_validation(self, dataloader, current_iter, tb_logger, save_img):
2     # add sample images to tb_logger
3     result = (result / 255.).astype(np.float32)
4     result = cv2.cvtColor(result, cv2.COLOR_BGR2RGB)
5     if tb_logger is not None:
6         tb_logger.add_image('samples', result, global_step=current_iter,
7                             dataformats='HWC')

```

4.9.3. Wandb 记录及解读

wandb 类似 tensorboard 的云端版本，可以在浏览器方便地查看模型训练的过程和曲线。

BasicSR 提供了部分模型的 Wandb 训练曲线

<https://app.wandb.ai/xintao/basicsr>

我们目前只是把 tensorboard 的内容同步到 wandb 上，因此要使用 wandb，必须打开 tensorboard logger。配置文件如下：

```
1  logger:  
2      # 是否使用 tensorboard logger  
3  use_tb_logger: true  
4      # 是否使用 wandb logger, 目前 wandb 只是同步 tensorboard 的内容, 因此要使用  
5          ↳ wandb, 必须也同时使用 tensorboard  
6  wandb:  
7      # wandb 的 project. 默认是 None, 即不使用 wandb  
8      project: ~  
9      # 如果是 resume, 可以输入上次的 wandb id, 则 log 可以接起来  
10     resume_id: ~
```

第 5 章

指标

本章节介绍在图像超分辨研究中经常使用的评价指标的相关知识，以及如何在 BasicSR 框架中使用这些指标进行测试。

§5.1. 概述

深度学习发展的旋风产生了源源不断的图像处理算法，这些算法可以生成失真较小、或对人感知上友好的复原图像。然而，限制图像处理方法未来发展的关键瓶颈之一就是“评估机制”。尽管人眼几乎可以毫不费力地区分感知上更好的图像，但算法要公平地衡量视觉质量是一项挑战。我们通常通过一些图像质量评估方法 (Image quality assessment, IQA) 测量复原图像和真实图像(Ground Truth, GT) 之间的相似性来评估。最近，一些不需要参考图像的 IQA 方法也被用于评估各种算法，例如 Ma 和 Perceptual Index (PI)。在某种程度上，这些 IQA 方法是图像处理领域取得长足进步的主要原因，因为他们提供了一个量化的基准，以促进指标上更优秀的算法的诞生。

接下来，我们先介绍图像质量评估(IQA)方面的知识。IQA 方法用于测量在采集、压缩、复制和后处理操作过程中可能会降低的图像质量。根据不同的使用场景，IQA 方法可以分为全参考法 (Full-reference IQA, FR-IQA) 和无参考法 (No-reference IQA, NR-IQA)。FR-IQA 方法通常从信息或感知特征相似度的角度衡量两幅图像之间的相似度，已广泛应用于图像/视频编码、恢复和通信质量的评估。除了最广泛使用的 PSNR，FR-IQA 已经经过广泛的研究，并至少可以追溯到 2004 年提出的 SSIM，它首先在测量图像相似性时引入了结构信息。PSNR 和 SSIM 同时也是在各种图像复原研究中使用最广泛的评价指标。除此之外，很多 FR-IQA 方法也被提出来弥补 IQA 方法的结果与人类判断之间的差距，例如 IFC, VSI, FSIM 等。然而，不断出现的新算法一直在不断提高图像恢复的效果，PSNR 和 SSIM 的定量结果和感知质量之间越来越不一致。有研究指出，面向感知效果的图像处理中，PSNR 和 SSIM 等指标衡量的失真程度和图像所展示的感知质量是彼此冲突的。此时，一些更符合人感知判断的评价指标也被用于评价图像复原算法，如 LPIPS。

除了上述 FR-IQA 方法外，一些 NR-IQA 方法也经常被用来在没有参考图像时评价图像的质量。一个比较典型的场景就是对真实世界中图像复原效果的评价。一些流行的 NR-IQA 方法包括 NIQE、BRISQUE 和 PI。在最近的一些工作中，结合 NR-IQA 和 FR-IQA 方法来测量 IR 算法。

§5.2. PSNR

PSNR (Peak signal-to-noise ratio, 峰值信噪比) 是图像处理研究中应用最广泛的评价指标之一。PSNR 是一个表示信号的最大可能功率和影响它的精度的破坏性噪声功率的比值的工程术语。PSNR 常用对数分贝单位来表示，简写为 dB (decibel)。PSNR 基于逐像素的均方误差 (Mean square error, MSE) 来定义。两个尺寸为 $m \times n$ 的单通道图像 I 和 I' ，其中 I 是高质量的参考图像， I' 为经过退化的低质量图片或者复原后的图像，那么它们的均方误差定义为：

$$\text{MSE} = \frac{1}{mn} \sum_{i=1}^m \sum_{j=1}^n (I[i, j] - I'[i, j])^2.$$

而 PSNR 被定义为：

$$\text{PSNR} = 10 \times \log_{10} \left(\frac{\text{Peak}^2}{\text{MSE}} \right) = 20 \times \log_{10} \left(\frac{\text{Peak}}{\sqrt{\text{MSE}}} \right).$$

其中，Peak 是表示图像像素强度的最大取值，如果每个采样点用 8 位表示，那么 Peak = 255。

在 BasicSR 框架中，与 PSNR 计算相关的代码存放在 `basicsr/metrics/psnr_ssimm.py` 文件中。对于 `numpy.ndarray` 类型的变量，我们约定输入图像的数据格式为 `Unit8`，尺寸为 `[h, w, c]` (高，宽，通道数)。对于彩色图片通道顺序为 `BGR`。此时输入图像像素的取值范围为 `[0, 255]` 整数取值。此时，我们使用如下函数计算 PSNR：

```

1  @METRIC_REGISTRY.register()
2  def calculate_psnr(img, img2, crop_border, input_order='HWC',
3      test_y_channel=False, **kwargs):
4      # img, img2: 输入图像变量
5      # crop_border: 是否在计算PSNR时切除边缘的像素。使用神经网络处理图像时，边缘
6      # → 的几个像素通常会有较大误差。
7      # input_order: 输入的尺寸顺序，默认为 'HWC'
8      # test_y_channel: 是否转换到 Y 空间计算 PSNR。Y 指代 YCbCr 格式图像中的灰度
9      # → 通道。
10     ...

```

当输入变量为 `torch.Tensor` 类型时，我们约定输入图像的数据格式为 `Float32`，尺寸为 `[n, c, h, w]` (批次大小，通道数 (3 或者 1)，高，宽)。对于彩色图片通道顺序为 `RGB`。此时输入图像像素的取值范围为 `[0, 1]` 浮点数取值。此时，我们使用如下以 `_pt` 结尾的函数计算 PSNR：

```

1
2  @METRIC_REGISTRY.register()
3  def calculate_psnr_pt(img, img2, crop_border,
4      test_y_channel=False, **kwargs):
5      ...

```

需要注意的是，此函数支持对于一整个批次 (batch) 的数据计算 PSNR。

Image	色彩空间	Matlab	Numpy	Pytorch CPU	Pytorch GPU
Set14/baboon	RGB	20.419710	20.419710	20.419710	20.419710
Set14/baboon	Y	—	22.441898	22.441899	22.444916
Set14/comic	RGB	20.239912	20.239912	20.239912	20.239912
Set14/comic	Y	—	21.720398	21.720398	21.721663

表 5.1: 各个 PSNR 实现结果之间的比较

在实现上, PSNR 的计算在不同人、不同版本的实现之间有微小的差异。我们对比了我们的实现和其他实现之间的差异, 结果如表 5.2 所示:

§5.3. SSIM

SSIM (structural similarity index, 图像结构相似性指标) 是另一个被广泛使用的图像相似度评价指标。与 PSNR 评价逐像素的图像之间差异不同，SSIM 在图像质量上的衡量更侧重于图像的结构信息，这与人类对于视觉信息的感知是相似的。因此普遍认为 SSIM 更贴近人类对于图像质量的判断。

此处的结构相似性的基本思想是自然图像是高度结构化的，即自然图像中相邻像素之间存在很强的相关性，这种相关性承载着场景中物体的结构信息。人类视觉系统习惯于在查看图像时提取这样的结构信息。因此，在设计衡量图像畸变程度的图像质量测量指标时，结构畸变的测量是重要的一环。

给定两个图像信号 \mathbf{x} 和 \mathbf{y} ，SSIM 被定义为：

$$\text{SSIM}(\mathbf{x}, \mathbf{y}) = [l(\mathbf{x}, \mathbf{y})]^\alpha [c(\mathbf{x}, \mathbf{y})]^\beta [s(\mathbf{x}, \mathbf{y})]^\gamma,$$

SSIM 由亮度对比 $l(\mathbf{x}, \mathbf{y})$ 、对比度对比 $c(\mathbf{x}, \mathbf{y})$ 、结构对比 $s(\mathbf{x}, \mathbf{y})$ 三部分组成。这些评价指标由以下方式定义：

$$l(\mathbf{x}, \mathbf{y}) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, c(\mathbf{x}, \mathbf{y}) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, s(\mathbf{x}, \mathbf{y}) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}.$$

其中 $\alpha > 0$, $\beta > 0$, $\gamma > 0$ 用于调整亮度、对比度和结构之间的相对重要性。 μ_x 及 μ_y 、 σ_x 及 σ_y 分别表示 \mathbf{x} 和 \mathbf{y} 的平均值和标准差， σ_{xy} 是 \mathbf{x} 和 \mathbf{y} 的协方差， C_1 、 C_2 、 C_3 是常数，用于维持结果的稳定。实际使用时，为简化起见，我们定义参数为 $\alpha = \beta = \gamma = 1$ 以及 $C_3 = C_2/2$ ，得到：

$$\text{SSIM}(\mathbf{x}, \mathbf{y}) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}.$$

在实际计算两幅图像的结构相似度指数时，我们会指定一些局部化的窗口，一般为 $N \times N$ 的小块，计算窗口内信号的结构相似度指数。然后每次以像素为单位移动窗口，直到计算出整幅图像每个位置的局部结构相似度指数。所有局部结构相似度指标的平均值为两幅图像的结构相似度指标。结构相似度指数的值越大，表明两个信号之间的相似度越高。一般来讲，PSNR 和 SSIM 的结果趋势是一致的，即一般 PSNR 高，则 SSIM 也高。

在 BasicSR 框架中，与 PSNR 计算相关的代码存放在 `basicsr/metrics/psnr_ssim.py` 文件中。与计算 PSNR 的接口类似，其函数包含对 `numpy.ndarray` 类型的输入进行处理的 `calculate_ssim` 函数以及对 `torch.Tensor` 类型的输入进行处理的 `calculate_ssim_pt` 函数。其对于输入的类型、格式、尺寸以及各参数的约定是与 PSNR 的计算中描述的一致的。

与 PSNR 不同，SSIM 的计算在不同实验版本之间的差异较大。在 BasicSR 中，我们以 Matlab 最原始的版本保持一致。我们对比了我们的实现和其他实现之间的差异，结果如表 5.2 所示

Image	色彩空间	Matlab	Numpy	Pytorch CPU	Pytorch GPU
Set14/baboon	RGB	0.391853	0.391853	0.391853	0.391853
Set14/baboon	Y	—	0.453097	0.453097	0.453171
Set14/comic	RGB	.567738	.567738	.567738	.567738
Set14/comic	Y	—	0.585511	0.585511	0.585522

表 5.2: 各个 SSIM 实现结果之间的比较

§5.4. NIQE

将在第二期加入

§5.5. 如何使用指标

5.5.1. 通过配置文件指定

在训练的 validation 阶段或者使用 `test.py` 测试时，我们可以在配饰文件中指定所需要使用的指标。这样程序就会自动计算相应指标了。比如下面的配置就会计算 PSNR 和 NIQE 的指标，分别调用了 `calculate_psnr` 和 `calculate_niqe`。

```
# validation settings
val:
  ...
metrics: # 这块是 validation 中使用的指标的配置
  psnr: # metric 名字，这个名字可以是任意的
    type: calculate_psnr # 选择指标类型
    # 以下属性是灵活的，根据不同 metric 有不同的设置
    crop_border: 4 # 计算指标时 crop 图像边界像素范围（不纳入计算范围）
    test_y_channel: false # 是否转成在 Y(CbCr) 空间上计算
    better: higher # 该指标是越高越好，还是越低越好。选择 higher 或者
      ↳ lower, 默认为 higher
  niqe: # 这是在 validation 中使用的另外一个指标
    type: calculate_niqe
    crop_border: 4
    better: lower # the lower, the better
```

5.5.2. 使用脚本计算

我们在 `scripts/metrics` 文件中也提供了调用指标的脚本。读者可以根据相关说明计算指标。

第 6 章

如何添加与修改

本章主要介绍如何在 BasicSR 框架中添加自定义的 Dataset，网络结构 (Architecture)，模型 (Model)，损失函数 (Loss) 以及指标 (Metric)。使用者需要关注四个方面，即：

1. 相关文件的存放和命名
2. 编写自定义文件
3. 注册新添加类
4. 以及在配置文件中进行设置

这一部分的内容大体上十分相似，使用者只要对某一个模块比较熟悉（如添加修改网络结构），即可快速类比至其他各个部分。在添加新的自定义模块时，理解并参考已有文件可以帮助使用者快速上手。

值得提及的是，当用户使用 **BasicSR-template** 进行开发，尤其是针对指标模块时，以下操作可能并不完全适用，具体详见第10章：**BasicSR-examples 模板** 相关部分。

§6.1. 添加修改 Dataset

第 1 步 Dataset 文件的存放与命名：Dataset 文件存放在 `basicsr/data/` 文件夹下。例如，`basicsr/data/paired_image_dataset.py`。用户可根据需求对已有的 Dataset 进行修改，或是添加自定义 Dataset 文件。在创建新的自定义 Dataset 文件时，注意文件名需以 `_dataset.py` 作为结尾

第 2 步 编写自定义 Dataset：在 Dataset 文件中对自定义 Dataset 类进行命名，**需要注意新建类名不能与已有类名重复，否则会导致后续注册机制报错**。关于 Dataset 文件中函数功能详解见章节4.4：**代码主体结构**，此处不再赘述。对于需要添加新的设置参数，用户可以灵活利用 `opt` 参数从配置文件中读取

第 3 步 注册 Dataset：用户需要对新建的 Dataset 类进行注册。注册机制的原理详见章节4.2：**代码主体结构**。此处具体操作为，首先对 `DATASET_REGISTRY` 函数进行导入，然后在新建类上方添加修饰器来注册新建函数。以 `paired_image_dataset.py` 中的 `PairedImageDataset` 为例：

```

1  from basicsr.utils.registry import DATASET_REGISTRY
2
3  @DATASET_REGISTRY.register()
4  class PairedImageDataset(data.Dataset):
5      ...

```

第 4 步 在配置文件中设置自定义 Dataset：将配置文件（即 YAML 文件）中 **datasets** 部分中 **type** 参数设置为新建的 Dataset 类名即可。该部分其余参数的功能与 Dataset 中用户自定义的功能对应。以使用 `paired_image_dataset.py` 中的 **PairedImageDataset** 为例：

```

1  # dataset and data loader settings
2  dataset:
3      ...
4      type: PairedImageDataset    # 设置为需要使用的 Dataset 类名
5      ...

```

§6.2. 添加修改模型

第 1 步 模型文件的存放与命名：模型文件存放在 `basicsr/models/` 文件夹下。例如，`basicsr/archs/sr_model.py`。用户可根据需求对已有的模型进行修改，或是添加自定义模型文件。在创建新的自定义模型文件时，注意文件名需以 `_model.py` 作为结尾。

第 2 步 编写自定义模型：在模型文件中对自定义模型类进行命名，需要注意新建类名不能与已有类名重复，否则会导致后续注册机制报错。关于模型文件中的函数功能详解见章节4.5：代码主体结构。模型部分涉及的函数较多，但一般情况下需要改写的部分非常有限。用户往往只需要继承已有模型，并对需要更改的函数进行重构即可。以 `basicsr/archs/swinir_model.py` 中的 **SwinIRModel** 为例，该模型相较于图像超分通用的 `basicsr/archs/sr_model.py` 中的 **SRModel** 仅需更改 `test` 函数，因此 **SwinIRModel** 类在继承了 **SRModel** 的基础上只对 `test` 函数进行了重构：

```

1  class SwinIRModel(SRModel):    # SwinIRModel 继承自 SRModel
2      def test(self):    # 重构 test 函数
3          ...

```

第 3 步 注册模型：用户需要对新建的模型类进行注册。注册机制的原理详见章节4.2：代码主体结构。此处具体操作为，首先对 **MODEL_REGISTRY** 函数进行导入，然后在新建类上方添加修饰器来注册新建类。以 `sr_model.py` 中的 **SRModel** 为例：

```

1  from basicsr.utils.registry import MODEL_REGISTRY
2
3  @MODEL_REGISTRY.register()
4  class SRModel(nn.Module):
5      ...

```

第 4 步 在配置文件中设置自定义模型：将配置文件（即 YAML 文件）中 **general settings** 部分中的 **model_type** 参数设置为新建的模型类名即可。以使用 [sr_model.py](#) 中的 **SRModel** 为例：

```
1 # general settings
2 ...
3 type: SRModel # 设置为需要使用的模型类名
4 ...
```

除此之外，模型与整个配置文件的内容都是息息相关的，涉及到数据的读取与处理、模型网络结构、训练优化和测试评估等几乎所有内容的设置组成，而非一个独立的部分。用户在修改配置文件的结构时，建议参考已有文件作为模板，重点对模型进行修改的部分在配置文件中做对应处理。

§6.3. 添加修改网络结构

第 1 步 网络结构文件的存放与命名：网络结构文件存放在 [basicsr/archs/](#) 文件夹下。例如，[basicsr/archs/srresnet_arch.py](#)。用户可根据需求对已有的网络结构进行修改，或是添加自定义网络结构文件。在创建新的自定义网络结构文件时，注意文件名需以 **_arch.py** 作为结尾。

第 2 步 编写自定义网络结构：在网络结构文件中对自定义网络结构类进行命名，需要注意新建类名不能与已有类名重复，否则会导致后续注册机制报错。关于网络结构文件中的函数功能详解见章节[4.6: 代码主体结构](#)。对于需要手工设置的参数，用户可以灵活利用 **opt** 参数从配置文件中读取。

第 3 步 注册网络结构：用户需要对新建的网络结构类进行注册。注册机制的原理详见章节[4.2: 代码主体结构](#)。此处具体操作为，首先对 **ARCH_REGISTRY** 函数进行导入，然后在新建类上方添加修饰器来注册新建类。以 [srresnet_arch.py](#) 中的 **MSRResNet** 类为例：

```
1 from basicsr.utils.registry import ARCH_REGISTRY
2
3 @ARCH_REGISTRY.register()
4 class MSRResNet(nn.Module):
5     ...
```

第 4 步 在配置文件中设置自定义网络结构：将配置文件（即 YAML 文件）中 **network structures** 部分中的 **type** 参数设置为新建的网络结构类名即可。该部分其余参数的功能与模型和网络结构中用户自定义的功能对应。以使用 [srresnet_arch.py](#) 中的 **MSRResNet** 为例：

```
1 # network structures
2 network_g: # g网络设置
3 ...
4 type: MSRResNet # 设置为需要使用的网络结构类名
5 ...
```

§6.4. 添加修改损失函数

第 1 步 损失函数的存放与命名：损失函数文件存放在 `basicsr/losses/` 文件夹下。例如，`basicsr/losses/gan_loss.py`。用户可根据需求对已有的损失函数进行修改，或是添加自定义损失函数文件。在创建新的损失函数文件时，注意文件名需以 `_loss.py` 作为结尾。

第 2 步 编写自定义损失函数：在损失函数文件中对自定义损失函数类进行命名，**需要注意新建类名不能与已有类名重复，否则会导致后续注册机制报错**。关于损失函数的功能详解见章节4.7：代码主体结构。对于需要手工设置的参数，用户可以灵活利用 `opt` 参数从配置文件中读取。

第 3 步 注册损失函数：用户需要对新建的损失函数类进行注册。注册机制的原理详见章节4.2：代码主体结构。此处具体操作为，首先对 `LOSS_REGISTRY` 函数进行导入，然后在新建类上方添加修饰器来注册新建类。以 `basicsr/losses/basic_loss.py` 中的 `L1Loss` 为例：

```
1  from basicsr.utils.registry import LOSS_REGISTRY
2
3  @LOSS_REGISTRY.register()
4  class L1Loss(nn.Module):
5      ...
```

第 4 步 在配置文件中设置自定义损失函数：将配置文件（即 YAML 文件）中 `losses` 部分中相应损失函数项的 `type` 参数设置为新建的损失类名即可。需要注意损失函数项的存在与模型有关。以使用 `basicsr/losses/basic_loss.py` 中的 `L1Loss` 为例：

```
1  # losses
2  pixel_opt: # pixel-wise 损失函数项, 与模型有关
3      type: L1Loss # 设置为需要使用的损失函数类名
4      ...
```

🔔 添加非 Class 的损失函数

在实际使用情况中，我们会遇到一些损失函数，他们不是以类 (Class) 的形式出现，而是普通的函数。比如 StyleGAN2 中使用的 `r1_penalty` 和 `gradient_penalty_loss`。

此时，我们不再以注册机制的方式使用，而是直接在模型中调用相关函数。

```
1  from basicsr.losses.gan_loss import r1_penalty
2
3  class StyleGAN2Model(BaseModel):
4      ...
5      def optimize_parameters(self, current_iter):
6          ...
7          real_pred = self.net_d(self.real_img)
8          l_d_r1 = r1_penalty(real_pred, self.real_img) # 直接调用损失函数
9          ↪ r1_penalty
10         ...
```

§6.5. 添加修改指标

第1步 指标的存放与命名：指标文件存放在 `basicsr/metrics/` 文件夹下。对于命名规则无要求，一般直接以功能命名即可，如 `basicsr/metrics/psnr_ssim.py`。

第2步 编写自定义指标：在指标文件中对自定义指标函数进行命名，需要注意新建函数名不能与已有函数名重复，否则会导致后续注册机制报错。关于指标的功能详解见第5章：[指标](#)。在编写完自定义指标后，注意在 `basicsr/metrics/__init__.py` 文件中对添加的自定义指标进行导入。以 `calculate_psnr` 为例：

```
1  from .psnr_ssim import calculate_psnr
```

第3步 注册指标：用户需要对新建的指标函数进行注册。注册机制的原理详见[章节4.2：代码主体结构](#)。此处具体操作为，首先对 `METRIC_REGISTRY` 函数进行导入，然后在新建函数上方添加修饰器来注册新建函数。以 `psnr_ssim.py` 中的 `calculate_psnr` 为例：

```
1  from basicsr.utils.registry import METRIC_REGISTRY
2
3  @METRIC_REGISTRY.register()
4  def calculate_psnr(img, img2, ...):
5      ...
```

第4步 在配置文件中设置自定义指标：将配置文件（即 YAML文件）中 `validation settings` 部分中 `metric` 部分中的 `type` 参数设置为新建的指标函数名即可。指标的其他参数设置对应其功能部分代码。以使用 `psnr_ssim.py` 中的 `calculate_psnr` 为例：

```
1  # validation settings
2  val:
3      ...
4      metrics:
5          psnr: # 指标名称，可以是任意的，用于标记
6              type: calculate_psnr # 设置为需要使用的指标函数名
7              ...
```

■ 注意

- 指标的新建注册机制和其他几类 (dataset, model, arch, loss) 不同: 1) 需要显式地在 `basicsr/metrics/__init__.py` import 函数; 2) 新建的是函数而不是类
- 目前指标主要是在 Numpy 上计算的, 比如 `calculate_psnr`、`calculate_ssim`, 有些也提供了基于 PyTorch 计算的版本, 比如 `calculate_psnr_pt`、`calculate_psnr_pt`

第 7 章

数据准备

这部分主要讲述数据存储形式，FileClient 类，以及一些常见数据集的获取和描述。

§7.1. 常见用法

目前支持的数据存储形式有以下三种：

1. 直接以图像/视频帧的格式存放在硬盘
2. 制作成 LMDB。训练数据使用这种形式，一般会加快读取速度
3. 若是支持 Memcached，则可以使用。它们一般应用在集群上

目前，我们可以通过 yaml 配置文件方便地修改。以支持 DIV2K 的 PairedImageDataset 为例，根据不同的要求修改 yaml 文件。

1. 直接读取硬盘数据

```
1 type: PairedImageDataset
2 dataroot_gt: datasets/DIV2K/DIV2K_train_HR_sub
3 dataroot_lq: datasets/DIV2K/DIV2K_train_LR_bicubic/X4_sub
4 io_backend:
5   type: disk
```

2. 使用 LMDB。在使用前需要先制作 LMDB，参见章节 7.2.1：数据准备，注意我们在原有的 LMDB 上，新增加了 meta 信息，而且具体保存二进制内容也不同，因此其他来源的 LMDB 并不能直接拿过来使用

```
1 type: PairedImageDataset
2 dataroot_gt: datasets/DIV2K/DIV2K_train_HR_sub.lmdb
3 dataroot_lq: datasets/DIV2K/DIV2K_train_LR_bicubic_X4_sub.lmdb
4 io_backend:
5   type: lmdb
```

3. 使用 Memcached。机器/集群需要支持 Memcached。具体的配置文件根据实际的 Memcached 需要进行修改:

```

1 type: PairedImageDataset
2 dataroot_gt: datasets/DIV2K_train_HR_sub
3 dataroot_lq: datasets/DIV2K_train_LR_bicubicX4_sub
4 io_backend:
5   type: memcached
6   server_list_cfg: /mnt/lustre/share/memcached_client/server_list.conf
7   client_cfg: /mnt/lustre/share/memcached_client/client.conf
8   sys_path: /mnt/lustre/share/pymc/py3

```

§7.2. 数据存储格式

7.2.1. LMDB 具体说明

我们在训练的时候使用 LMDB 存储形式可以加快 IO 和 CPU 解压缩的速度 (测试的时候数据较少, 一般就没有太必要使用 LMDB)。其具体的加速要根据机器的配置来, 以下几个因素会影响:

- 有的机器设置了定时清理缓存, 而 LMDB 依赖于缓存。因此若一直缓存不进去, 则需要检查一下。一般 `free -h` 命令下, LMDB 占用的缓存会记录在 `buff/cache` 条目下面
- 机器的内存是否足够大, 能够把整个 LMDB 数据都放进去。如果不是, 则它由于需要不断更换缓存, 会影响速度
- 若是第一次缓存 LMDB 数据集, 可能会影响训练速度。可以在训练前, 进入 LMDB 数据集目录, 把数据先缓存进去: `cat data.mdb > /dev/null`

7.2.1.1. 文件结构

除了标准的 LMDB 文件 (`data.mdb` 和 `lock.mdb`) 外, 我们还增加了 `meta_info.txt` 来记录额外的信息。下面用一个例子来说明:

```

1 DIV2K_train_HR_sub.lmdb
2   ├── data.mdb
3   ├── lock.mdb
4   └── meta_info.txt

```

7.2.1.2. meta信息

`meta_info.txt`。我们采用 `txt` 来记录, 是为了可读性。其里面的内容为:

```
1 0001_s001.png (480,480,3) 1
2 0001_s002.png (480,480,3) 1
3 0001_s003.png (480,480,3) 1
4 0001_s004.png (480,480,3) 1
5 ...
```

每一行记录了一张图片，有三个字段，分别表示：

1. 图像名称 (带后缀): 0001_s001.png
2. 图像大小: (480,480,3) 表示是 $480 \times 480 \times 3$ 的图像
3. 其他参数 (BasicSR 里面使用了 cv2 压缩 png 程度): 因为在复原任务中，我们通常使用 png 来存储，所以这个 1 表示 png 的压缩程度，也就是 CV_IMWRITE_PNG_COMPRESSION 为 1。CV_IMWRITE_PNG_COMPRESSION 可以取值为 [0, 9] 的整数，更大的值表示更强的压缩，即更小的储存空间和更长的压缩时间

7.2.1.3. 二进制内容

为了方便，我们在 LMDB 数据集中存储的二进制内容是 cv2 encode 过的 image: cv2.imencode('.png', img, [cv2.IMWRITE_PNG_COMPRESSION, compress_level])。可以通过 compress_level 控制压缩程度，平衡储存空间和读取(包括解压缩)的速度。

7.2.1.4. 如何制作

我们提供了脚本 scripts/data_preparation/create_lmdb.py 来制作。在运行脚本前，需要根据需求修改相应的参数。目前支持 DIV2K, REDS 和 Vimeo90K 数据集，其他数据集可仿照进行制作。

```
python scripts/data_preparation/create_lmdb.py --dataset div2k
python scripts/data_preparation/create_lmdb.py --dataset reds
python scripts/data_preparation/create_lmdb.py --dataset vimeo90k
```

■ 加速 IO 方法

除了使用 LMDB 加速 IO 外，还可以使用 prefetch 方式，具体参见章节[4.4.4: 代码主体结构](#)。

§7.3. meta 文件介绍

meta 文件是记录数据集信息的。一般我们使用 txt 格式, 这样我们打开就能够知道它里面记录的内容。

有时候我们从一个目录里面扫描全部的文件会比较慢、耗时, 此时如果提供了 meta 文件, 就可以比较快速地得到所有文件(比如图片)的路径列表了。

同时我们也会使用 meta 文件来划分数据集, 比如训练、测试集等。

它一般在以下几个场景中使用:

1. 制作 LMDB 后会同步产生一个 meta 文件, 这个 meta 有着自己固定的格式, 不能修改, 否则可能会影响 LMDB 数据的读取。详细参见章节 [7.2.1: 数据准备](#)
2. PairedImageDataset 支持 `meta_info_file` 参数, 会使用这个 meta 文件生成待读取的文件路径。这个可以根据用户自己的需要进行自定义

待完善

7.3.1. 现有 meta 文件说明

在 `basicsr/data/meta_info` 目录下有提供了一些常用的 meta 文件, 说明如下:

待完善

§7.4. File Client 介绍

我们参考了 MMCV 的 FileClient 设计。为了使其兼容 BasicSR, 我们对接口做了一些改动(主要是为了适应 LMDB)。具体可以参见代码 `file_client.py`。

待完善

§7.5. 常见数据集介绍与准备

推荐把数据通过 `ln -s src dst` 软链接到 datasets 目录下。

7.5.1. 图像数据集 DIV2K 与 DF2K

DIV2K 与 DF2K 数据集被广泛使用在图像复原的任务中。其中 DF2K 是 DIV2K 和 Flickr2K 的融合。

数据准备步骤

1. 从 [DIV2K 官网](#) 下载数据。Flickr 2K 可从 <https://cv.snu.ac.kr/research/EDSR/Flickr2K.tar> 下载

2. Crop to sub-images: 因为 DIV2K 数据集是 2K 分辨率的 (比如: 2048×1080), 而我们在训练的时候往往并不那么大 (常见的是 128×128 或者 192×192 的训练 patch). 因此我们可以先把 2K 的图片裁剪成有 overlap 的 480×480 的子图像块. 然后再由 dataloader 从这个 480×480 的子图像块中随机 crop 出 128×128 或者 192×192 的训练 patch。运行脚本 `extract_subimages.py`:

```
python scripts/data_preparation/extract_subimages.py
```

使用之前可能需要修改文件里面的路径和配置参数。注意: sub-image 的尺寸和训练 patch 的尺寸 (gt_size) 是不同的。我们先把2K分辨率的图像 crop 成 sub-images (往往是 480×480), 然后存储起来。在训练的时候, dataloader 会读取这些 sub-images, 然后进一步随机裁剪成 gt_size \times gt_size 的大小

3. [可选] 若需要使用 LMDB, 则需要制作 LMDB, 参考章节7.2.1。**数据准备** 运行脚本:

```
python scripts/data_preparation/create_lmdb.py --dataset div2k
```

注意选择 `create_lmdb_for_div2k` 函数, 并需要修改函数相应的配置和路径

4. 单元测试: 我们可以单独测试 dataset 是否正常。注意修改函数相应的配置和路径: `test_scripts/test_paired_image_dataset.py`。
5. [可选] 若需要生成 meta_info_file 文件, 请运行

```
python scripts/data_preparation/generate_meta_info.py
```

7.5.2. 视频帧数据集 REDS

REDS 是常用的视频帧数据集。数据集官方网站: <https://seungjunnah.github.io/Datasets/red.html>。我们重新整合了 training 和 validation 数据到一个文件夹中: 训练集合原来有240个clip (序号从000到239), 我们把 validation clips 重命名, 从240到269。

Validation 的划分

官方的 validation 划分和 EDVR、BasicVSR 论文中的划分不同 (当时为了比赛的设置):

name	clips	total number
REDSOfficial	[240, 269]	30 clips
REDS4	000, 011, 015, 020 clips from the original training set	4 clips

表 7.1: REDS 数据集中 Validation 的划分

余下的 clips 拿来做训练集合。注意: 我们不需要显式地分开训练和验证集合, dataloader 会做这件事。

数据准备步骤

1. 从[官网](#)下载数据

2. 整合 training 和 validation 数据。运行

```
python scripts/data_preparation/regroup_reds_dataset.py
```

3. [可选] 若需要使用 LMDB, 则需要制作 LMDB, 参考章节 7.2.1: **数据准备**。运行

```
python scripts/data_preparation/create_lmdb.py --dataset reds
```

注意选择 `create_lmdb_for_reds` 函数, 并需要修改函数相应的配置和路径

4. 单元测试: 我们可以单独测试 `dataset` 是否正常。注意修改函数相应的配置和路径: `test_scripts/test_reds_dataset.py`。

7.5.3. 视频帧数据集 Vimeo90K

Vimeo90K 是常用的视频帧数据集。官网地址: <http://toflow.csail.mit.edu>。

数据准备步骤

1. 下载数据: [Septuplets dataset -> The original training + test set \(82GB\)](#). 这些是 Ground-Truth。里面有 `sep_trainlist.txt` 文件来区分训练数据
2. 生成低分辨率图片。Vimeo90K 测试集中的低分辨率图片是由 MATLAB bicubic 降采样函数而来。运行脚本 `data_scripts/generate_LR_Vimeo90K.m` (run in MATLAB) 来生成低清图片
3. [可选] 若需要使用 LMDB, 则需要制作 LMDB, 参考章节 7.2.1: **数据准备**。运行

```
python scripts/data_preparation/create_lmdb.py --dataset vimeo90k
```

注意选择 `create_lmdb_for_vimeo90k` 函数, 并需要修改函数相应的配置和路径

4. 单元测试: 我们可以单独测试 `dataset` 是否正常。注意修改函数相应的配置和路径: `test_scripts/test_vimeo90k_dataset.py`。

第 8 章

部署

将在第二期加入

第 9 章

脚本介绍

将在第二期加入

第 10 章

BasicSR-examples 模板

将在第二期加入

可暂时先参考: <https://github.com/xinntao/BasicSR-examples>

第 11 章

经验

将在第二期加入



BasicSR

SUPER RESTORATION