代码规范要求与说明

注：本代码规范仅针对 Python 编码，若使用其他编程语言请适当参考本代码规范。本代码规范中的示例均来自 SNN_RL 项目，欢迎各位参考该项目。

遵循代码规范的重要性

遵循良好的代码规范能够：

1. 提高代码可读性，降低沟通成本。
2. 提高代码质量，降低代码维护成本。
3. 提高团队开发的合作效率，有助于 code review。
4. 培养良好的编码习惯，为未来参加工作打下良好的基础。
5. ……

代码规范细则

本代码规范以 Google 开源项目风格指南—— Python 风格指南为基础，请大家先自行查阅该风格指南，**仔细阅读。**

该风格指南中存在一些不适用的规范，应当以下面的规范为准：

注释

注释的 `#` 符号后面应当有一个空格。

注释中提到的任何数据类型或变量名应当用 `` ` `` 符号包起来。例如：

```
x = 1   # Define a variable called `x`.
```

行尾注释

单行代码如有必要也应当在行尾进行注释说明，行尾注释应当与单行代码的末尾相距至少一个空格。

块注释

一个具有一定功能性和目的性的代码块应当进行注释说明，具有块注释的代码块后面应当有一个空行。

行尾注释与块注释的示例如下：

```
# Create the Atari environments for learning and evaluation.
train_env = make_atari(args.env_id)
train_env = wrap_deepmind(train_env, frame_stack=True)
train_env.seed(args.seed)
eval_env = make_atari(args.env_id, args.max_episode_step)
eval_env = wrap_deepmind(eval_env, episode_life=False, clip_rewards=False, frame_stack=True)
eval_env.seed(args.seed)

replay_memory = ReplayMemory(args.replay_memory_size)  # Create the experience replay memory for learning.
action_num = train_env.action_space.n  # As the output layer size of the neural networks.
device = torch.device(args.device)
```

函数注释

一个函数除非非常简短易懂，否则应当注释说明其功能。

函数注释的示例如下：

```
def _frame_stack_to_state(self, frame_stack, is_batch=False):
    """ Convert `frame_stack` of `numpy.ndarray` type to `state` of `torch.tensor` type.（简介函数功能）

    （详细介绍函数的功能与逻辑）

    Args:
        frame_stack: `numpy.ndarray` （介绍传入参数类型与意义）
        is_batch: `bool` （介绍传入参数类型与意义）

    Returns:
        state: `torch.tensor` （介绍返回值类型与意义）

    """
    # To save memory, the frames stored in the experience replay memory are integer, therefore need to be normalized.
    state = numpy.array(frame_stack, dtype=numpy.float32) / 255.0

    # From numpy shape = (batch, height, width, channel) to pytorch shape (batch, channel, height, width).
    if is_batch:
        state = numpy.transpose(state, axes=(0, 3, 1, 2))
        state = torch.tensor(state, dtype=torch.float32, device=self._device)
    else:
        state = numpy.transpose(state, axes=(2, 0, 1))
        state = torch.tensor(state, dtype=torch.float32, device=self._device).unsqueeze(dim=0)

    return state
```

行长度

每行不超过 128 个字符。

如果单行代码加上行尾注释超过了行长度要求，则应当将行尾注释提前为块注释。例如：

```
# （行尾注释超过行长度）
eval_env = Monitor(eval_env, monitor_dir, video_callable=lambda episode_id: True, force=True)   # Set up the monitor for video recording.

# （应当将行尾注释提前为块注释）
# Set up the monitor for video recording.
eval_env = Monitor(eval_env, monitor_dir, video_callable=lambda episode_id: True, force=True)
```

类

如果一个类不继承自其他类，则不需要括号，如下：

```
class SampleClass:
    # TODO: ...
```

补充代码规范

本代码规范补充了一些 Google 开源项目风格指南**没有涵盖**的规范，如下：

命名

变量

变量的命名应当使用英文单词中的名词，使用单下划线作为单词之间的分隔，不能使用拼音。同时，除非是常用的缩写，否则需要尽量避免使用缩写。

变量名长不是问题，看不懂才是问题。

函数

函数的命名应当使用英文单词中的动词开头，使用单下划线作为单词之间的分隔。

应当尽可能将函数的功能通过函数名表达出来。

类的成员变量与方法

类的成员变量与方法的命名应当遵循“非必要不暴露”的原则，意思是除非有必要暴露给外界调用，否则都应该使用单下划线开头来命名。约定单下划线开头表示类的成员变量与方法是 protected 的，而非下划线开头命名的类的成员变量与方法可以被外界调用。

全局变量与类的成员变量

若非必要，应当避免使用全局变量与类的成员变量。

特别是对于类的成员变量，不要为了一时方便而把无关类本身的变量设置为类的成员变量。

函数

封装

一个具有明确功能或目标，且具有较高独立性的代码块应当封装为函数。例如：

```python
def set_logger():
    with open(LOG_CONFIG_FILE, mode="r") as file:
        log_config = yaml.load(file, yaml.SafeLoader)

    # Set the file name of the log file.
    log_name = f"{args.model}_{args.env_id}_{args.method}_{time_str}"
    log_config["handlers"]["file_handler"]["filename"] =
os.path.join(LOG_DIR, f"{log_name}.log")
    dictConfig(log_config)

    writer = SummaryWriter(log_dir=os.path.join(LOG_DIR, log_name))
    return writer


def set_random_seed(seed, is_using_cuda):
    random.seed(seed)
    numpy.random.seed(seed)
    torch.manual_seed(seed)

    if is_using_cuda:
        torch.cuda.manual_seed_all(seed)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False

def main():
    writer = set_logger()
    is_using_cuda = True if args.device == "cuda" else False
    set_random_seed(args.seed, is_using_cuda)

    if args.method == "learn":
        learn(writer)
    elif args.method == "play":
        play(writer)

    writer.close()
```

长度

一个函数的长度应当不超过 64 行。函数过长会导致难以阅读与理解，应当将长函数中的代码块函数化来缩减函数的长度。

README.md

一个完成的项目应当拥有一个 `README.md` 文档对该项目进行介绍，至少应当包含：

1. 项目名称。
2. 项目功能与目的。
3. 环境依赖。
4. 项目结构。
5. 运行方式。

推荐使用 Typora 编辑器编辑 Markdown (.md) 文档。

开源许可证

开源发布的项目应当具有开源许可证。

开源许可证即授权条款。开源项目并非完全没有限制。最基本的限制，就是开源项目强迫任何使用和修改该项目的人承认发起人的著作权和所有参与人的贡献。任何人拥有可以自由复制、修改、使用这些源代码的权利，不得设置针对任何人或团体领域的限制；不得限制开源项目的商业使用等。而开源许可证就是这样一个保证这些限制的法律文件。

常见的开源许可证有：

- `MIT License`
- `Apache License 2.0`
- `GNU General Public License v3.0`

`MIT License` 的内容示例如下：

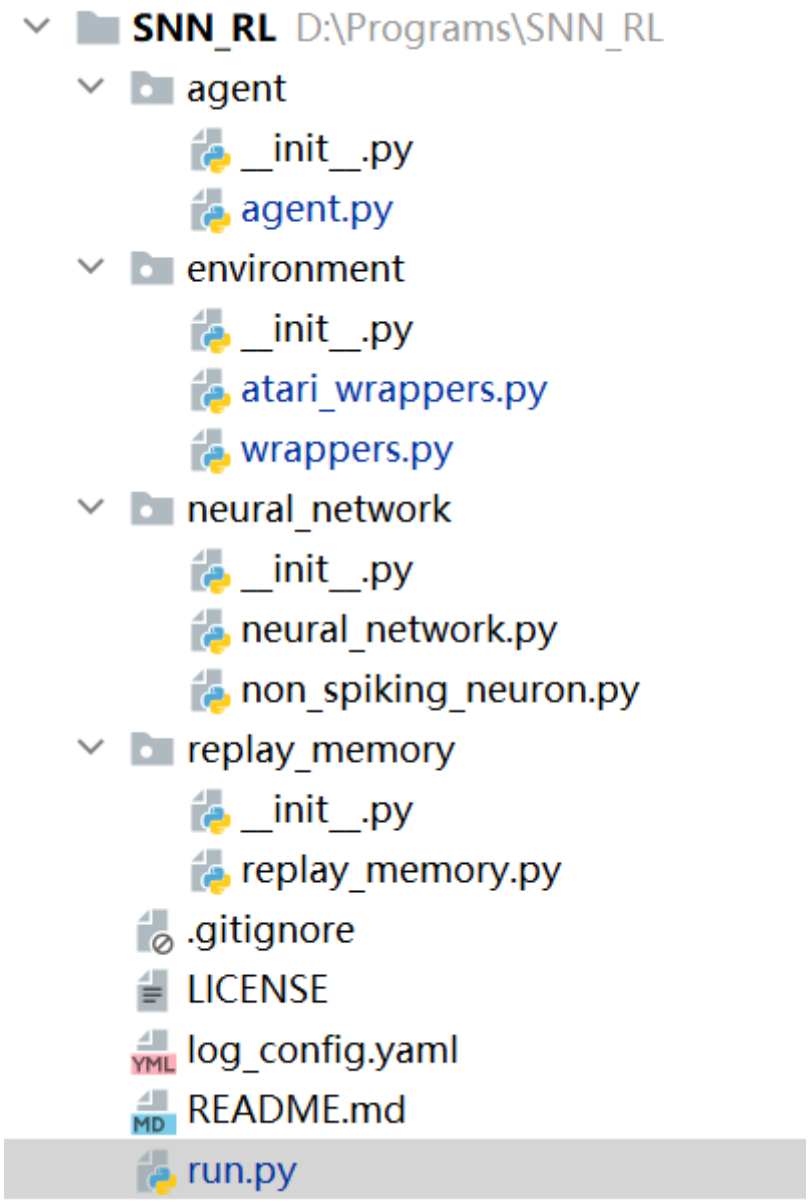
```
MIT License

Copyright (c) 2021 AptX395

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to
deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE
SOFTWARE.

```

项目结构

一个好的项目应当具有清晰明了的项目结构。项目结构需要满足以下要求：

1. 具有易辨识的程序入口文件，例如：`main.py`，`run.py`，`train.py` 等。
2. 程序入口文件应当唯一，避免同时存在多个程序入口文件。
3. 除程序入口文件以外的其他代码文件均使用包（Package）的形式封装，请注意模块（Module）与包的区别。

项目结构示例如下：



程序入口文件

main 函数

程序入口文件应当使用以下方式定义入口函数（main 函数）：

```python
def main():
    # TODO: ...

if __name__ == "__main__":
    main()
```

命令行参数

一个科研项目，常常需要进行很多不同条件的实验。为了便于实验，程序应当通过命令行参数的形式来运行。因此程序入口文件应当具有解析命令行参数的功能，实验所需的各种条件应当使用命令行参数进行指定。

程序入口文件中的命令行参数解析功能示例如下：

```
parser = argparse.ArgumentParser()

# Settings.
parser.add_argument("--model", type=str, default="Dqn", choices=["Dqn", "Dsqn", "DsqnRstdp"])
parser.add_argument("--env_id", type=str, default="BreakoutNoFrameskip-v4", choices=["BreakoutNoFrameskip-v4"])
parser.add_argument("--method", type=str, default="learn", choices=["learn", "play"])
parser.add_argument("--model_path", type=str)
parser.add_argument("--seed", type=int, default=0)
parser.add_argument("--device", type=str, default="cuda", choices=["cuda", "cpu"])
parser.add_argument("--gpu", type=str, default="0", choices=["0", "1", "2", "3"])
```

通过命令行参数运行程序的示例如下：

```
python run.py --model=Dqn --env_id=BreakoutNoFrameskip-v4 --method=learn --device=gpu --gpu=0
```

后台运行

一次实验往往需要运行较长时间，数小时、数天，甚至更长都有可能。使用前台方式运行程序可能因为意想不到的情况而中断，使用后台方式运行程序可以避免这个问题。

Linux 系统的后台程序的命令为在命令行运行程序的指令结尾加上 `&` 符号，为了避免退出登录而导致程序运行中断，还需要在命令行运行程序的指令开头加上 `nohup` 命令。`nohup` 命令会将程序的控制台输出重定向并存储到 `nohup.out` （默认）中，如果需要同时后台运行多个程序，应当指定程序的控制台输出重定向文件。

Linux 系统后台运行程序的示例如下：

```
nohup python run.py > nohup_output/experiment_name.out &
```

日志

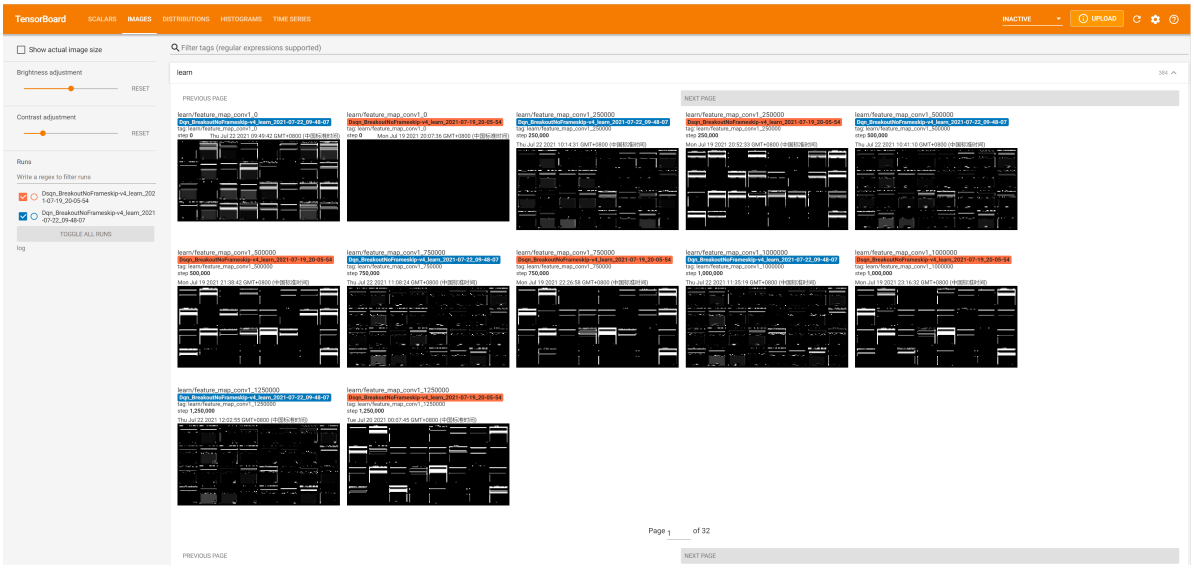
实验过程中往往需要监控准确率、训练进度、网络中间层输出等许多变量，简单的 `print` 语句并不能满足这些需求，应当使用 Python 的 `logging` 包将必要的信息记录在日志文件中，推荐使用 `tensorboard` 包记录进行实验分析所需的变量。

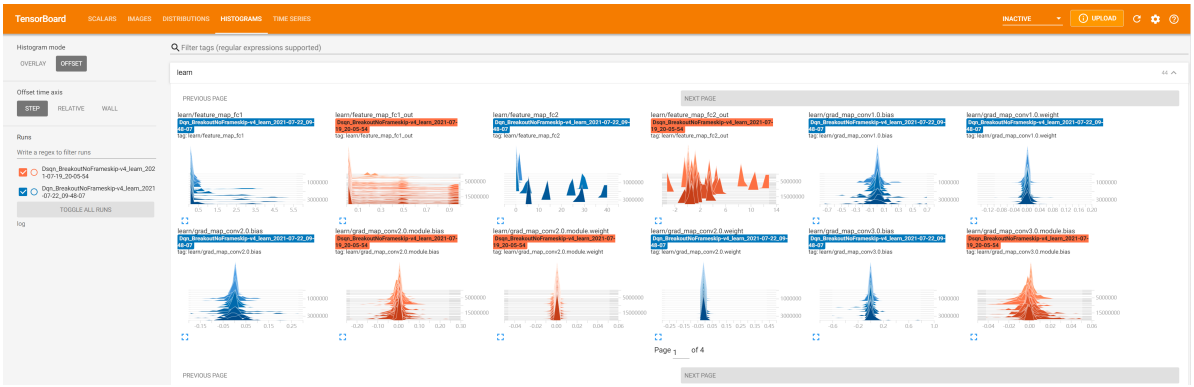
`tensorboard` 包提供了方便且丰富的功能，可以记录准确率、loss 等变量并以曲线的形式表示，网络中间层输出也可以可视化表示。

日志文件所记录的信息示例：

```
2021-07-19 00:01:43 | file_logger | agent | learn | INFO | Start learning
2021-07-19 00:01:43 | file_logger | agent | _explore | INFO | Start exploring
2021-07-19 00:02:54 | file_logger | agent | _explore | INFO | Exploring is done
2021-07-19 00:03:41 | file_logger | agent | _record_data | INFO | timestep: 0 average_score: 0.8333333333333334   score_std: 1.4161763857498668 average_max_q: 0.039163731038570404
2021-07-19 00:03:41 | file_logger | agent | _record_data | INFO | timestep: 0 max_average_score: 0.8333333333333334
2021-07-19 02:08:24 | file_logger | agent | _record_data | INFO | timestep: 250000  average_score: 2.1  score_std: 3.399509768579386    average_max_q: 0.6885067403316498
```

`tensorboard` 包所记录的信息示例：





代码管理

代码管理是开发项目的过程中不可或缺的重要一环，应当使用 Git 进行代码版本管理，使用 Github 进行代码托管，以便于代码回溯、合作开发等等。

项目内容

托管到 Github 的项目内容应当仅包含代码文件与必要的配置文件，而不应当出现日志、模型、数据集等文件。应当使用 `.gitignore` 文件指定需要忽略的文件与文件夹。

`.gitignore` 文件内容示例如下：

```
# PyCharm
.idea/

# VSCode
.vscode/

# Log
log/
*.log

# Model
model/

# Data
data/

# Nohup
nohup_output/
*.out

# Tensorboard
runs/
```

结语

本代码规范还不够完善，欢迎各位提出自己的建议与意见，对进行本代码规范进行补充。