设计文档

1. envs

强化学习用到的环境, 目前只包含围棋环境go_env. 之后可以将空战环境加入这里.

go_env

围棋环境, 由c++实现(源码见 `go_env/cpp_src`), 并封装成python类GoEnv(在 `go_env/environment.py` 中定义).

简单使用示例:

```
import envs
import configs  # 所有设置
env = envs.GoEnv(configs.Config())

state, _, _ = env.reset() # 获取初始状态
state, _, _, done = env.step(state, 3)  # 落子
state, _, _, done = env.step(state, 28)
state, _, _, done = env.step(state, 72)

env.render(state)  # 显示棋盘, "X"表示黑子, "O"表示白子
print("done:", done)


# 运行结果
. . . X . O . . .
. . . . . . . . .
. . . . . . . . .
. X . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
done: False
```

获取合法动作和编码状态

```
for action in [2, 31, 4, 9, 44, 56, 77, 8]:  # 随便选一些动作
    state, legal_actions, observation, done = env.step(state, action)
env.render(state)
print("done:", done)
print(legal_actions)  # 合法动作
print(observation[2]) # 用于输入神经网络的编码平面，平面2是己方3气及以上的连通块
```

```
. . O X O O . . X
X . . . . . . . .
. . . . . . . . .
. X . . X . . . .
. . . . . . . . O
. . . . . . . . .
. . X . . . . . .
. . . . . . . . .
. . . . . O . . .
done: False
[ 0  1  6  7 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 29 30
 32 33 34 35 36 37 38 39 40 41 42 43 45 46 47 48 49 50 51 52 53 54 55 57
 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 78 79 80]
[[0. 0. 0. 0. 1. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0.]]
```

2. models

包含网络模型和使用到的网络层, 由pytorch实现.

`.layers`：模型中用到的网络层和模块, 包括残差块, transformer层, 块嵌入等

`.loss`：损失函数, 目前包含**软标签交叉熵损失**(pytorch没有实现, 需自己定义)

`.utils`：其他工具函数

`alphazero_network.py`：主要模型, 有"Conv", "Transformer", "Hybrid"三种模型可以选择

```
import models
import configs

# 使用cpu
model = models.AlphaZeroNetwork(config.Config())
observation = torch.from_numpy(observation).unsqueeze(0) # 转成pytorch张量
policy, value = model(observation) # 网络推理

# 使用gpu
model = models.AlphaZeroNetwork(config.Config()).to("cuda")
observation = torch.from_numpy(observation).unsqueeze(0).to(
    next(model.parameters()).device
)
policy, value = model(observation) # 网络推理
```

3. search

搜索算法. 包含蒙特卡罗树搜索、节点及动作选择算法等, 使用python实现.

`mcts.py`：蒙特卡罗树搜索模块

`nodes.py`：树搜索中用到的节点

`select_action.py`：动作选择算法, 包括UCB以及**截断概率**等

简单使用示例:

```
import search
import models, envs  # 要用到模型和环境
import configs

config = config.Config()
mcts = search.MCTS(config)
model = models.AlphaZeroNetwork(config).to("cuda")
env = envs.GoEnv(config)

# 首先获取新的根节点
root, _ = mcts.get_new_root(
    old_root=None,
    fall_action=None, # 前两个都先设为None就行了
    model=model,      # 模型
    env=env           # 环境
)
# 在根节点上搜索指定次数
root, _ = mcts.run(root, model, env,
    add_exploration_noise=True,  # 可以选择添加探索噪声
    num_rollouts=210  # 设定搜索次数
)
```

这样就完成了对该根节点的树搜索, 假设实际落子动作为action, 则继承action对应的子树继续搜索

```
# 还是用get_new_root来获得新根节点
root, done = mcts.get_new_root(
    old_root=root,      # 旧树的根节点
    fall_action=action, # 实际落子动作
    model=model,        # 模型
    env=env             # 环境
)
```

此时的root就是action对应子树的根节点, 重复上面的步骤继续搜索即可

```
root, _ = mcts.run(root, model, env,
    add_exploration_noise=True
    # 也可以不指定搜索次数, 自动使用config里设置的值
)
```

直到 `done==True` 时, 一局自对弈完成.

4. self_play.py

自对弈模块. 利用 `search.MCTS` 进行持续自对弈, `@ray.remote` 的修饰使之成为ray的一个actor, 多个actor可以并行地进行自对弈.(ray是一个分布式执行引擎)

`class SelfPlay` 的主要函数有

`SelfPlay.continuous_self_play()`: 用于持续自对弈

`SelfPlay.policy_evaluate()`: 让两个模型对弈n局, 评估模型的好坏

`class GameHistory` 包含 `observation_history`, `policy_history`, `value_history` 等多个**列表**, 用于保存游戏历史到 `replay_buffer` 中

5. replay_buffer.py

经验回放池. 自对弈产生的历史保存于此, 再采样batch用来训练神经网络. buffer的容量有限, 如果满了优先删除最早存入的数据(先入先出). 是一个单独的ray的actor.

`GameHistory` 以**字典**的形式保存在 `ReplayBuffer.buffer` 中, 保证了采样时能进行快速的随机访问(字典的底层由哈希表实现, 随机访问的速度远快于基于链表的列表).

`class ReplayBuffer` 的主要函数有

`ReplayBuffer.get_buffer()` 用于获取buffer(保存到硬盘)

`ReplayBuffer.get_batch()` 随机采样batch用于训练神经网络, 抽样时还会对样本进行数据增强, 包括旋转和翻转的8种组合(见 `ReplayBuffer.data_augment()`)

(*)新增了 `class ReplayTree` , 目前支持对价值进行回溯, 初步测试发现value_loss会变得很小

6. trainer.py

网络训练模块, 从replay_buffer中获取batch对网络进行**权值更新**, 以及定义**损失函数**, 管理**学习率**等. 也是ray的一个actor.
训练时使用多条自对弈线程, 一条训练线程即可.

主要函数有 `Trainer.continuous_update_weights()`：持续更新网络权值

7. shared_storage.py

共享存储. 保存检查点(checkpoint)信息, 包含网络权值、当前对弈总局数、总训练步数、学习率等信息, 所有线程共享. 也是单独的一个actor, 不消耗GPU资源, 但会占用一定的内存.

检查点保存在 `SharedStorage.current_checkpoint` 中, 是一个字典, 该字典在main.py中定义.

主要函数:
`SharedStorage.save_checkpoint()`：保存检查点
`SharedStorage.get_checkpoint()`：获取检查点
`SharedStorage.get_info(keys)`：获取checkpoint中某些键对应的值
`SharedStorage.set_info(keys)`：设置checkpoint中某些键对应的值

8. main.py

主函数

未完待续...