QQ浏览器2021AI算法大赛-赛道二-算法文档

队伍名： PKU-DAIR **成员：** 姜淮钧、沈彧、黎洋 **学校：** 北京大学

2021年10月17日

团队介绍

PKU-DAIR团队的三位成员来自北京大学崔斌教授DAIR实验室的AutoML项目组。团队研究方向包括超参数优化（HPO）、神经网络结构搜索（NAS）、自动化机器学习系统（AutoML System）等。团队不仅在国际顶级会议上发表了多篇论文，如多精度集成贝叶斯优化算法MFES-HB（论文链接），为提高AutoML技术的易用性与可用性，团队还相继在GitHub开源了黑盒优化系统OpenBox（项目地址：https://github.com/PKU-DAIR/open-box）、自动化机器学习系统MindWare（项目地址：https://github.com/PKU-DAIR/mindware）等。

本次自动化超参数优化算法大赛，团队基于实验室自研开源黑盒优化系统OpenBox进行调参，系统内集成了最优秀的黑盒优化（超参数优化）方法。初赛时使用OpenBox系统中的并行贝叶斯优化（Bayesian optimization）算法，决赛在此基础上加入早停机制。下面将进行详细介绍。

初赛算法介绍

赛题理解

初赛赛题为经典的超参数优化问题，优化器需要在每轮以同步并行方式推荐5个超参数配置，共执行20轮推荐，即总共搜索100个配置。对每个配置均执行完整资源的验证，并且在比赛的问题抽象中，不同超参数的验证时间相同。根据现有研究，贝叶斯优化是超参数优化（黑盒优化）问题上state-of-the-art的方法，而且比赛场景中的超参数空间维度不超过6维，并非超高维问题，较适合贝叶斯优化方法，因此我们选定贝叶斯优化作为初赛的搜索算法。另外，问题中所有的超参数均为连续型（离散浮点型）超参数，这决定了我们的超参数空间定义方法、贝叶斯优化代理模型以及优化器的选择，接下来也将分别进行介绍。

算法核心技术——贝叶斯优化模块介绍

贝叶斯优化简介

超参数优化是典型的黑盒优化问题，即对于目标函数（超参数-奖励函数），具体表达式或导数信息是不可知的，只能通过尝试输入获取输出来推测目标函数的内部情况。

贝叶斯优化是解决黑盒优化问题的一个迭代式框架，优化流程包括如下步骤：

- 使用代理模型（surrogate model）对已有观测历史数据（尝试过的超参数和对应的奖励）进行建模拟合；

- 使用采集函数（acquisition function）评估搜索空间内的超参数，平衡探索与利用。对采集函数执行优化，找到验证价值最大（使采集函数值最大）的下一组超参数；
- 在目标函数上评估超参数，得到奖励；
- 将新评估的超参数和结果加入观测历史数据，重复以上步骤。

贝叶斯优化算法封装在OpenBox系统中，代码实现的主要流程如下：

```
# 使用贝叶斯优化得到超参数配置推荐
def get_suggestions(self, history_container, batch_size):
    # ...
    # 基于观测历史数据，训练贝叶斯优化代理模型
    self.surrogate_model.train(X, Y)
    # ...
    # 更新采集函数（使用EI函数时，要更新当前最优观测值）
    self.acquisition_function.update(eta=incumbent_value, ...)
    # 使用优化器优化采集函数，得到使采集函数值最大的一个（一组）超参数
    challengers = self.optimizer.maximize(...)
    # ...
    return batch_configs_list  # 依据并行算法，得到下一轮需要验证的超参数
```

超参数空间定义

首先，我们使用ConfigSpace库定义超参数空间。由于赛题中的超参数均为离散浮点型，并可近似为等间距分布，因此使用Int型定义超参数（UniformIntegerHyperparameter）（本质上和使用Float定义相同，但避免了超参数取值范围边缘可能出现不同间距的问题）。在ConfigSpace库中，Float型和Int型超参数均被视作连续型，在执行优化时会自动将参数范围缩放至[0, 1]。

初始化方法

贝叶斯优化需要一定数量的历史数据才能启动，我们使用了一种贪心法生成初始超参数配置。该方法从随机候选配置中，逐步挑选距离已有配置最远的配置加入初始配置集合。使用该方法进行初始化能更好地探索超参数空间，经测试效果稍好于完全随机初始化方法。初始化配置数设置为10个。该方法集成在OpenBox系统中，可通过init_strategy="random_explore_first"调用。

代理模型

贝叶斯优化中的代理模型（surrogate model）有多种选择，包括高斯过程（Gaussian process）、概率随机森林（probabilistic random forest）、Tree Parzen Estimator（TPE）等，其中高斯过程在连续超参数空间上（如数学问题）优化效果较好，概率随机森林在含有分类超参数的空间上优化效果较好。本次比赛只包含连续型超参数，经测试，高斯过程作为代理模型效果最好。高斯过程使用OpenBox系统默认的Matern5/2核，核超参数通过最大对数似然(maximize log likelihood)得到。

采集函数与优化

我们使用常用的Expected Improvement（EI）函数作为贝叶斯优化的采集函数（acquisition function）。在优化采集函数时，我们使用系统中的"random_scipy"优化器。该优化器在结合局部搜索与随机采样的基础上，使用L-BFGS-B算法对采集函数执行优化。测试表明，相较于单纯使用随机采样，该方法能对采集函数进行更为充分的优化，从而更大程度发挥GP模型和EI函数的潜能。

其他

传统的贝叶斯优化每轮只能推荐一个配置，因此设计并行推荐方法是一个值得考虑的问题。我们尝试了系统中实现的并行贝叶斯方法，包括"median_imputation"中位数插补法，即使用历史观察结果的中位数，填补并行batch中推荐配置的性能，重新训练代理模型并得到下一个并行推荐配置，以及"local_penalization"局部惩罚法，对并行已推荐配置在采集函数上施加局部惩罚，这两种方法的目的都是提高对超参数空间的探索性。不过经过测试，在本次比赛问题上这些方法的效果不佳，最终我们采用多次优化采集函数并去重的方式执行并行推荐，达到了较好的性能。

此外，为增大贝叶斯优化的探索性，保证算法收敛，我们设置每次推荐时使用随机搜索的概率为0.1。

代码实现

初赛代码仅需调用OpenBox系统中的并行贝叶斯优化器SyncBatchAdvisor，即可实现上述功能：

```
from openbox import SyncBatchAdvisor
self.advisor = SyncBatchAdvisor(
    config_space=self.config_space,
    batch_size=5,
    batch_strategy='reoptimization',
    initial_trials=10,
    init_strategy='random_explore_first',
    rand_prob=0.1,
    surrogate_type='gp',
    acq_type='ei',
    acq_optimizer_type='random_scipy',
    task_id='thpo',
    random_state=47,
)
```

每轮执行推荐时，调用advisor的get_suggestions接口：

```
def suggest(self, suggestion_history, n_suggestions):
    history_container = self.parse_suggestion_history(suggestion_history)
    next_configs = self.advisor.get_suggestions(n_suggestions, history_container)
    next_suggestions = [self.convert_config_to_parameter(conf) for conf in
next_configs]
    return next_suggestions
```

决赛算法介绍

赛题理解

决赛问题在初赛的基础上，对每个超参数配置提供14轮的多精度验证结果，供算法提前对性能可能不佳的配置验证过程执行早停。同时，总体优化预算时间减半，最多只能全量验证50个超参数配置，因此问题难度大大增加。如何设计好的早停算法，如何利用多精度验证数据是优化器设计的关键。

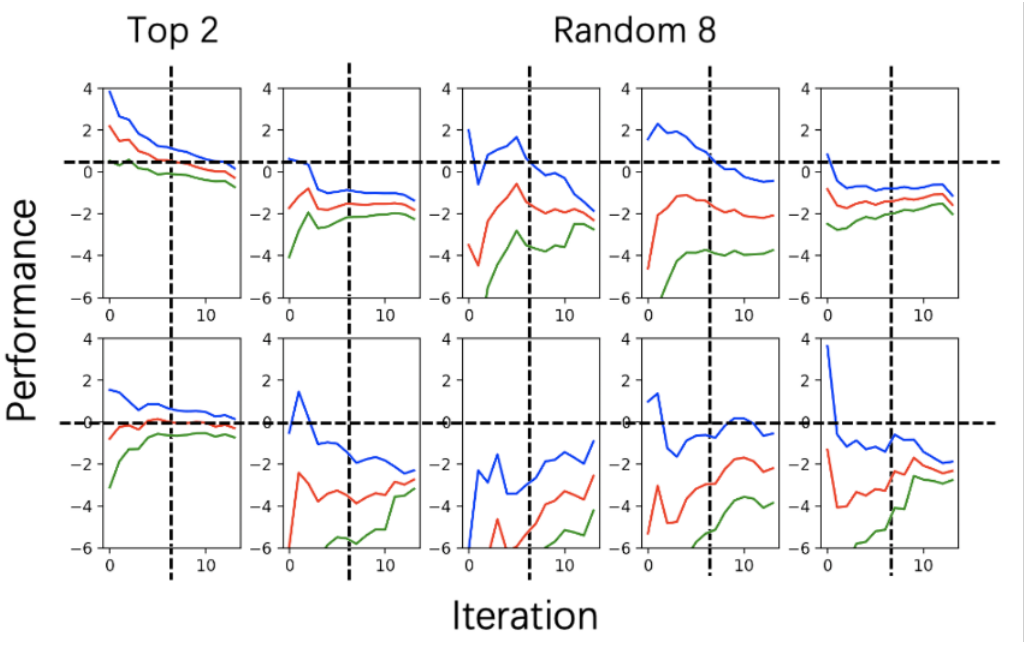
我们对本地公开的两个数据集进行了探索，发现了一些有趣的性质：

- 对于任意超参数配置，其第14轮的奖励均值位于前13轮置信区间内的概率为95%。
- 对于任意超参数配置，其前13轮中任意一轮的均值比第14轮均值大的概率为50%。
- 对于任意超参数配置，其14轮的置信区间是不断减小的，但均值曲线是任意波动的。

我们也对两两超参数配置间的关系进行了探索，比较了两两配置间前13轮的均值大小关系和第14轮的均值大小关系的一致性，发现：

- 在所有超参数配置之间，部分验证（1-13轮）和全量验证（14轮）均值大小关系一致的概率大于95%。
- 在空间中最终性能前1%的超参数配置之间，这种一致性大约在50%到70%之间。

下图为data-30空间中最终奖励排名前2的超参数和随机8个超参数的奖励-轮次关系：



我们在比赛开源代码仓库中提供了上述“数据探索”代码。

上述数据探索结果表明，根据前13轮的置信区间，我们可以推测第14轮奖励均值的位置。利用前13轮的均值大小关系，我们可以估计第14轮最终均值的大小关系，但是由于数据噪音的存在，排名靠前的超参数配置大小关系无法通过部分验证结果预估。由此我们设计了两种早停算法，分别是基于置信区间的早停和基于排名的早停，将在下一部分详细描述。

过于激进的早停策略在比赛中仍然存在问题。如果使用贝叶斯优化只对全量验证数据建模，由于总体优化预算时间很少，早停会减少可用于建模的数据量，使得模型不能得到充分训练。为解决这一问题，我们引入插值方法，增加模型可训练数据。

基于以上考量，最终我们的决赛算法在初赛贝叶斯优化算法的基础上，前期执行完整贝叶斯优化使模型得到较为充分的拟合，后期使用早停技术与插值法，加速超参数验证与搜索过程。下面将对早停模块做详细介绍。

算法核心技术——早停模块介绍

早停方法

由于超参数配置之间的部分验证轮次均值大小关系与最终均值大小关系存在一定的相关性，我们受异步多阶段早停算法ASHA的启发，设计了基于排名的早停算法：一个超参数如果到达需要判断早停的轮次，就计算其性能均值处于历史中同一轮次的超参数性能均值的排名，如果位于前$1/\eta$，则继续验证，否则执行早停。

依据95%置信区间的含义，我们还设计了另一种早停方法，即使用置信区间判断当前超参数配置是否仍有验证价值。如果某一时刻，当前验证超参数的置信区间上界差于已完全验证的性能前10名配置的均值，则代表至少有95%的可能其最终均值差于前10名的配置，故进行早停。使用本地数据验证，以空间中前50名的配置对前1000名的配置使用该方法进行早停，早停准确率在99%以上。

经过测试，结合贝叶斯优化时两种方法效果近似，我们最终选择使用基于排名的早停方法。无论是哪种方法，都需要设计执行早停的轮次。早停越早越激进，节省的验证时间越多，但是得到的数据置信度越低，后续执行插值时训练的模型就越不准确。为了权衡早停带来的时间收益和高精度验证带来的数据收益，我们选择只在第7轮（总共14轮）时判断每个配置是否应当早停。早停判断准则依据$\eta = 2$的ASHA算法，即如果当前配置均值性能处于已验证配置第7轮的后50%，就进行早停。

以下代码展示了基于排名的早停方法。首先统计各个早停轮次下已验证配置的性能并进行排序（比赛中我们使用早停轮次为第7轮），然后判断当前配置是否处于前$1/\eta$（比赛中为前1/2），否则执行早停：

```
# 基于排名的早停方法，prune_eta=2，prune_iters=[7]
def prune_mean_rank(self, iteration_number, running_suggestions,
suggestion_history):
    # 统计早停阶段上已验证配置的性能并排序
    bracket = dict()
    for n_iteration in self.hps['prune_iters']:
        bracket[n_iteration] = list()
    for suggestion in running_suggestions + suggestion_history:
        n_history = len(suggestion['reward'])
        for n_iteration in self.hps['prune_iters']:
            if n_history >= n_iteration:
                bracket[n_iteration].append(suggestion['reward'][n_iteration - 1]
['value'])
    for n_iteration in self.hps['prune_iters']:
        bracket[n_iteration].sort(reverse=True)  # maximize

    # 依据当前配置性能排名，决定是否早停
    stop_list = [False] * len(running_suggestions)
    for i, suggestion in enumerate(running_suggestions):
        n_history = len(suggestion['reward'])
        if n_history == CONFIDENCE_N_ITERATION:
            # 当前配置已完整验证，无需早停
            print('full observation. pass', i)
            continue
        if n_history not in self.hps['prune_iters']:
            # 当前配置不处于需要早停的阶段
            print('n_history: %d not in prune_iters: %s. pass %d.'
                  % (n_history, self.hps['prune_iters'], i))
            continue

```

```
            rank = bracket[n_history].index(suggestion['reward'][-1]['value'])
            total_cnt = len(bracket[n_history])
            # 判断当前配置性能是否处于前1/eta，否则早停
            if rank / total_cnt >= 1 / self.hps['prune_eta']:
                print('n_history: %d, rank: %d/%d, eta: 1/%s. PRUNE %d!'
                      % (n_history, rank, total_cnt, self.hps['prune_eta'], i))
                stop_list[i] = True
            else:
                print('n_history: %d, rank: %d/%d, eta: 1/%s. continue %d.'
                      % (n_history, rank, total_cnt, self.hps['prune_eta'], i))
        return stop_list
```

基于置信区间的早停方法可见我们的比赛开源代码库。

数据建模方法

对于贝叶斯优化的数据建模，我们尝试了多精度集成代理模型MFES-HB拟合多精度观测数据。该方法虽然能应对低精度噪声场景，但在决赛极其有限的优化时间限制内，可能无法快速排除噪声的干扰，导致效果不如仅使用最高精度数据建模。

我们最终选择只利用最高精度数据进行建模。为了弥补早停造成的高精度数据损失，我们引入插值方法，增加用于模型训练的数据量，具体来说，就是对早停的配置，设置一个完整验证时的性能均值，插入优化历史执行建模。对于插入值的选取，我们使用已完整验证配置的最终均值中位数进行插值。

以下为插值代码示例：

```
def set_impute_value(self, running_suggestions, suggestion_history):
    value_list = []
    for suggestion in running_suggestions + suggestion_history:
        n_history = len(suggestion['reward'])
        if n_history != CONFIDENCE_N_ITERATION:
            continue
        value_list.append(suggestion['reward'][-1]['value'])
    self.impute_value = np.median(value_list).item()
```

总结与思考

感谢QQ浏览器2021算法大赛为我们提供了宝贵的参赛机会。

通过探索数据、确定模型、尝试不同方法，在分数不断竞争上涨的比赛过程中，我们积累了实际的超参数优化经验，这是十分难得的。同时，比赛实践也打破了我们一些固有的观念，比如过去有论文工作指出，对于贝叶斯优化，高斯过程代理模型更适合优化黑盒数学问题，概率随机森林在大量机器学习超参数优化场景任务中表现更佳，而此次比赛告诉我们，超参数优化任务空间也可能有着较强的连续性，高斯过程模型也非常出色。

这次比赛提供的真实业务场景数据集，为我们开发与维护开源黑盒优化系统OpenBox、进一步保证其在通用场景下的性能有着重要的意义。同时，决赛的多精度优化场景也很新颖，使我们对超参数优化领域的认识更为深刻。

最后，再次感谢赛事组委会的辛勤付出，让我们拥有了一次丰富而又难忘的比赛体验。